Computational Techniques for Join Operations for Dynamic Programming Algorithms on Tree Decompositions

Rolf Plagmeijer

July 1, 2018

Abstract

Tree decompositions can be traversed with a dynamic programming algorithm in order to compute exact answers to various graph problems. A particular operation called a join operation is performed in these algorithms, and commonly this operation forms the bottleneck. In this master thesis, I will present a number of different results pertaining to these join operations. First, we will see that every 2×2 join operation can be computed in $O^*(2^k)$, where k represents treewidth. We will see that two $s \times s$ join operations, named the Fourier- and Möbius-Transform-based join operations, can be computed in $O*(s^k)$. We will introduce a new technique called filtering that computes join operations by computing a different but related join operation and doing only polynomially extra work. We will use this technique to prove a new result relating to an extension of Disjoint Set Sum.

1 Introduction

This is a master thesis about computational techniques for join operations for dynamic programming algorithms on tree decompositions. Tree decompositions are a type of structural decomposition for graphs that encode the tree-like structure of a graph. This data structure can be traversed with a dynamic programming algorithm in order to compute exact answers to various graph problems. A particular operation called a Join is used in these algorithms, and commonly these operations form the bottleneck of the algorithm. Join operations come in many shapes and sizes, but there are nonetheless certain things we can say about the time complexity of join operations.This master thesis will outline a couple of results that make it easier to determine efficient algorithms for computing join operations of various forms.

In Section 2, we will discuss the foundations for the theory behind the research. This includes a summary of tree decompositions, treewidth and an example dynamic programming algorithm. Section 3 delves into more detail about specific techniques that can be applied for computing join operations. In Section 4 we prove that every join operation of size 2×2 can be computed in $O^*(2^k)$, where k represents the treewidth of the tree decomposition we're using. In Sections 5 and 6 we prove that Fourier- and Möbius-Transform-based join operations of size $s \times s$ can

be computed in $O^*(s^k)$. In Section 7 we introduce a new technique called filtering that can compute join operations using a different related join operation and doing only polynomially extra work. Which join operations can be computed with this method is something that will be discussed in this section as well. We will also prove a new result relating to an extension of Disjoint Set Sum. Finally, Section 8 will offer conclusions and thoughts for further research.

2 Foundation

2.1 Tree decompositions

Let $G = (V(G), E(G))$ be an undirected graph. A tree decomposition $\mathcal{T}$ of G is defined as follows:

Tree Decomposition: A tree decomposition of a graph G is a pair $\mathcal{T} = (T, \{B_t\}_{t\in V(T)})$, where T is a tree in which every node t is assigned a vertex subset $B_t \subseteq V(G)$, called a bag, such that the following three conditions hold:

1. $\bigcup_{t\in V(T)} B_t = V(G)$. In other words, every vertex of G is in at least one bag.
2. For every $uv \in E(G)$, there exists a node t of T such that the bag B_t contains both u and v.
3. For every $v \in V(G)$, the set $T_v = \{x \in V(T) : v \in B_t\}$, i.e. the set of nodes whose corresponding bags contain v, induces a connected subtree of T.

The tree decomposition for a graph G is not unique. There always exists at least one valid Tree Decomposition, namely, the tree T with one node t, where $B_t = V(G)$. The usefulness of different tree decompositions can be quite varied however. Therefore, an important performance measure for tree decompositions, known as treewidth, is often used.

Treewidth: The treewidth of a tree decomposition $\mathcal{T} = (T, \{B_t\}_{t\in V(T)})$ of a graph G is defined as $\max_{t\in V(T)}|B_t|-1$.

In other words, the treewidth of a tree decomposition, often denoted by k, is the size of its largest bag minus 1. Tree decompositions and treewidth can, in general terms, be used to measure how 'tree-like' a certain graph is. Indeed, any tree will have a treewidth of 1.

These definitions for tree decompositions and treewidth were first introduced by Robertson and Seymour [1]. Computing a tree decomposition with minimal treewidth for a graph is unfortunately an $\mathcal{NP}$-hard problem [2], though there are some alternatives that can be used. Calculating a tree decomposition for a fixed treewidth k, should such a tree decomposition exist, is possible with linear time complexity [3], albeit with a high constant factor.

We can however obtain tree decompositions with small treewidth efficiently if the graph a tree decomposition is computed for belongs to certain graph classes. [4] There are also several upper bounds available that can work well if exact solutions aren't needed. [5].

2.2 Dynamic programming with tree decompositions

As mentioned, one of the reasons tree decompositions are used is that they allow for dynamic programming algorithms for solving various exact graph problems. These algorithms are exponential in treewidth k, the value of which is often much smaller than n, the value that is generally in the exponent of time complexities for these problems.

Before we can take a look at how these algorithms are set up however, we will introduce the concept of a nice tree decomposition, which will assist in the formulation of our algorithm.

Nice Tree Decomposition: A nice tree decomposition is a tree decomposition for which a node p is chosen as the root, and for which the following holds:

1. $B_p = \emptyset$, where p is the root of T.
2. $B_q = \emptyset$, for all leaves q of T.
3. Every non-leaf node of T is one of the following:
   - **Introduce node**: a node t with exactly one child d such that $B_t = B_d \cup \{v\}$ for some vertex $v \notin B_d$.
   - **Forget node**: a node t with exactly one child d such that $B_t = B_d \backslash \{v\}$ for some vertex $v \notin B_d$.
   - **Join node**: a node t with two children l, r such that $B_t = B_l = B_r$.

Nice tree decompositions were first introduced by Kloks. [6] Any tree decomposition can be changed into a nice tree decomposition without increasing its treewidth. This is left as an exercise to the reader.

Dynamic programming algorithms on tree decompositions make use of the structure of nice tree decompositions. Depending on the problem, the algorithm uses a state set S containing s states that a vertex in the graph can take on. The algorithm is performed using a bottom-up approach, starting at the leaf nodes of the decomposition. A table A_t is calculated for each node t using results from its child node(s). This table is indexed by so-called colorings; assignments of the states in S to all the vertices in B_t. The set of all colorings of a bag B is denoted by S^B. For each coloring $c \in S^B$, the table assigns a value in $\mathbb{Z}$. What this value represents or how it is calculated again depends on the problem. This is repeated until the root node p is reached, at which point the answer can be extracted.

A summary of a dynamic programming algorithm for the Minimum Dominating Set problem is given below. For a more detailed version of this algorithm see Exact Exponential-Time Algorithms for Domination Problems in Graphs. [9, section 11.2, p.208]

Let $\mathcal{T} = (T, \{B_t\}_{t \in V(T)})$ be a nice tree decomposition of a graph G. We will use the state set $S = \{1, 0_1, 0_0\}$, where 1 represents a vertex in the dominating set, 0_1 represents a dominated vertex not in the dominating set, and 0_0 represents an undominated vertex. The table corresponding to a node t in our decomposition $\mathcal{T}$ is denoted as $A_t(c)$, where c is a coloring of the vertices in the bag B_t using the states in S. The table entry $A_t(c)$, for node t and coloring c, stores the size of the smallest partial solution to Minimum Dominating Set for which:

1. The vertices in the bag B_t take on their color in c.
2. All vertices that have been forgotten by a forget node in the current branch are dominated.

For a vertex $u \in V(G)$, $c(u)$ is defined as the state the vertex u has in the coloring c. For a vertex $v \in V(G)$, we say that $N(v)$ is the set of all neighbors of v in G. We use the notation $c \times \theta$ in order to denote a coloring that ends in the state $\theta \in S$. Let t be an Introduce node that introduces a vertex v and has a child node d. We say that a coloring c_t of B_t matches a coloring c_d of B_d if:

1. For all $u \in B_d \backslash N(v)$: $c_t(u) = c_d(u)$.
2. For all $u \in B_d \cap N(v)$: either $c_t(u) = c_d(u) = 1$, or $c_t(u) = 0_1$ and $c_d(u) \in \{0_1, 0_0\}$.

With this, we can finally show the formulas used by our algorithm. These formulas are all in terms of a node t that may or may not have child nodes d, l, r:

- **Leaf node:** $A_t(\{1\}) = 1, A_t(0_1) = \infty, A_t(0_0) = 0$.
- **Forget node:** $A_t(c) = \min\{A_t(c \times \{1\}), A_d(c \times \{0_1\})\}$.
- **Introduce node:** (where v is the newly introduced vertex)
  - $A_t(c \times \{0_1\}) = \begin{cases} A_d(c) & \text{if } v \text{ has a neighbor with state 1 in } c \\ \infty & \text{otherwise} \end{cases}$
  - $A_t(c \times \{0_0\}) = \begin{cases} A_d(c) & \text{if } v \text{ has no neighbor with state 1 in } c \\ \infty & \text{otherwise} \end{cases}$
  - $A_t(c \times \{1\}) = \begin{cases} \infty & \text{if } c(u) = 0_0 \text{ for some } u \in N(v) \\ 1 + \min\{A_d(c')|c' \text{ matches } c\} & \text{otherwise} \end{cases}$

These operations scale with the size of B_t. Since the treewidth k is the largest size for a bag B_t, we use the term k in our time complexities. The running times of these operations are $O^*(3^k)$ for the Leaf and Forget nodes, and $O^*(4^k)$ for the Introduce node. See Exact Exponential-Time Algorithms for an explanation [9, section 11.2, p.210]

The only part of the algorithm that remains is the operation for Join nodes, which we call a join operation. For a Join node t, the join operation combines the tables from the two child nodes of t, l and r, into one table. The bags B_t, B_l and B_r all have exactly the same contents. Similar to the Introduce node, we will use a concept of colorings from these three bags matching. In this case, we will say that three colorings c_t, c_l and c_r of B_t, B_l and B_r respectively match if for each vertex $v \in B_t$:

- either $c_t(v) = c_l(v) = c_r(v) = 1$
- or $c_t(v) = c_l(v) = c_r(v) = 0_0$
- or $c_t(v) = 0_1$ while $c_l(v)$ and $c_r(v)$ are 0_1 or 0_0, but not both 0_0.

To put this into words: if a vertex in a coloring of B_t is either in our dominating set or undominated, then this must also be the case for any two colorings from the child nodes that we combine. However, if a vertex is not in the dominating set but is dominated, then we allow a couple more options. Such a vertex can be dominated by a vertex from the left side or a vertex from the right side of the tree decomposition, but it needs to be dominated from at least one of the two sides.

This now gives us the following formula for join nodes:

$$A_t(c_t) = \min_{c_t, c_l, c_r \text{ match}} A_l(c_l) + A_r(c_r) - \#_1(c_t)$$

The term $\#_1(c_t)$ in this formula is defined as the number of 1-states in the coloring c_t. For any combination of colorings we count the number of 1-states twice; once from the left side, and once from the right.

In order to compute tables for Join nodes, 5 combinations need to be considered, giving it a complexity of $O^*(5^k)$. This makes it the most complex type of node to compute in this algorithm.

3 Current techniques for computing join operations

In this section, we will examine a number of different ways join operations can be performed. Before we can go into that however, we will first need to provide these definitions:

State combination: Given a state set S with an ordering. The state combination between two states $p, q \in S$ is denoted by $j_{p,q}$ if $p \leq q$

The constraint of $a \leq b$ is important here since we want to make sure that our join operation is symmetrical. An asymmetrical join operation would treat the two children of a join node differently, which generally is not behaviour we want. The set $\mathcal{J}_S$ is the set of all state combinations for a state set S. It follows that $|\mathcal{J}_S|= \frac{1}{2}s(s+1)$.

Join operation: A join operation P is function that takes two tables $A_l, A_r : S^B \mapsto \mathbb{Z}$ and returns a table $A_t : S^B \mapsto \mathbb{Z}$.

Each join operation P has a mapping from $\mathcal{J}_S$ to $S \cup \{\times\}$, where $\times$ represents the empty combination. This mapping is what determines how the tables A_l and A_r are joined in order to form the resulting table A_t. If a state combination has the empty combination $\times$ assigned to it, then this means that that state combination is not allowed to be performed when performing join operation P. We abuse notation and write $P(j_{p,q}) = a$ with $j_{p,q} \in \mathcal{J}_S$ and $a \in S \cup \{\times\}$, denoting that P either combines states p and q into state a, or doesn't combine p and q at all.

Let us say that a join operation P combines two colorings c_l and c_r. How exactly the values $A_l(c_l)$ and $A_r(c_r)$ are combined depends on the problem at hand. For our algorithm of Minimum Dominating Set above, the way in which we combined these values was $A_l(c_l) + A_r(c_r) - \#_1(c_t)$. Generally however, the way in which we combine values is multiplication, i.e. $A_t(c_t) =$

	1	0_1	0_0
1	1		
0_1		0_1	0_1
0_0		0_1	0_0

Figure 1: *Join table for the join operation for* MINIMUM DOMINATING SET*. The empty cells in the table represents the empty combination* $\times$*.*

$A_l(c_l) \cdot A_r(c_r)$.

Each join operation can be visualized by using a join table. The join table for the join operation for MINIMUM DOMINATING SET shown in the previous section is displayed in Figure 1.

With this, we can start exploring some computation techniques for join operations. All of these techniques will be performed on a join node t with child nodes l and r.

3.1 Standard method

The first technique for computing join operations that we will discuss is the standard method. The computation that we used for MINIMUM DOMINATING SET in the previous section is an example of the standard method. When computing a join operation P using the standard method, the states used for the problem are simply combined with each other in the way prescribed by P.

The complexity of computing a join operation P using the standard method can be readily determined. Namely, for $p = \sum_{j_{a,b} \in \mathcal{J}_S} 2^{[a=b]}[P(j_{a,b}) \neq \times]$, the complexity of computing P with the standard method is $O^*(p^k)$. Terms with square bracket notation, such as $[a = b]$, are equal to 1 if the statement inside the square brackets is true, and 0 otherwise. More simply, p is equal to the amount of non-empty cells in the join table for P. Given this, computing the MINIMUM DOMINATING SET join operation with the standard method would have a time complexity of $O^*(5^k)$, which is the same time complexity we arrived at in the previous section.

3.2 State change

The state change method involves using a second state set different from the one the join operation normally uses in order to compute the table for the join node. Denoting the original state set by S_1 and the second state set by S_2, the state change method is performed as follows. First, the tables A_l and A_r are transformed so that they use the new state set S_2. The join operation is performed using the new state changed tables to create a single table using the state set S_2. This table is then state changed back to S_1 to get the result we initially wanted.

In order to examine this technique, we will look at a different algorithm for MINIMUM DOMINATING SET. The idea of using a different set of state than the usual states in order to solve MINIMUM DOMINATING SET was first introduced in a paper by Alber et al.[8], with an algorithm with a complexity of $O^*(4^k)$. Van Rooij, Bodlaender and Rosmanith were the first to use the State Change method, and with it managed to construct an algorithm that has a

complexity of $O^*(3^k)$.[10] This is the algorithm that will be discussed in this section.

Recall that the state set that we used for this problem in the previous section was $S_1 = \{1, 0_1, 0_0\}$. The state set that we will change to is $S_2 = \{1, 0_?, 0_0\}$. The states 1 and 0_0 are shared between both state sets and carry the same meaning in both. The new state, $0_?$, is a state that represents a vertex that is not in the Dominating Set, but may be dominated or not. It is essentially a state which means that a vertex with that state would have either the state 0_1 or 0_0 if that vertex were to be represented in S_1.

Before we tackle the actual changing of states, we must first change something about how we try to solve this problem. Namely, in order to apply state change in this case we must have our algorithm solve for the number of MINIMUM DOMINATING SETS instead of just its size. In other words, the tables construed at each node in the decomposition tree will be indexed by a coloring c and a size κ with $0 \leq \kappa \leq n$, meaning they are now represented as $A_t(c, \kappa)$. We do this because otherwise, it is impossibly to say with certainty whether a vertex represented with state set S_2 is dominated or not. These tables hold strictly more information, which will allow us to carry out the state change without losing information

In order to transform a table $A_t(c, \kappa)$ from one state set to another, we use $|B_t|$ steps. Each step, one of the vertices in all the colorings will be changed, and the values will be updated accordingly. We continue this until all vertices have been converted.

We will first tackle the conversion from S_1 to S_2. For any step of the algorithm, we denote by c_1 the part of the coloring that has not yet been changed and therefore uses the state set S_1, and by c_2 we denote the part of the coloring that has already been changed and therefore uses the state set S_2. The formula for changing a coloring to use the state $0_?$ then becomes:

$$A_t(c_1 \times \{0_?\} \times c_2, \kappa) = A_t(c_1 \times \{0_0\} \times c_2, \kappa) + A_t(c_1 \times \{0_1\} \times c_2, \kappa)$$

Entries with states 1 or 0_0 are kept unchanged. If we instead change from S_2 to S_1, then we get the following formula for changing back to state 0_1:

$$A_t(c_2 \times \{0_1\} \times c_1, \kappa) = A_t(c_2 \times \{0_?\} \times c_1, \kappa) - A_t(c_2 \times \{0_0\} \times c_1, \kappa)$$

This completes the explanation for performing a state change. As mentioned, the algorithm contains a total of $|B_t|$ steps, since each step, one of the vertices in the bag B_t is transformed. Each step contains $O(|A_t|)$ operations, one for each entry in the table. This brings the total computation time to $O(|B_t||A_t|)$, meaning that we get a general time complexity of $O(nk3^k)$ for performing the state change, which is we can also write as $O^*(3^k)$.

Now that we know how to apply the state change, we can look at the full algorithm. In particular, we will look at the Join node. For the computation of the Leaf, Introduce and Forget nodes, please refer to Exact-Exponential Time Algorithms. [9, section 11.3, p219-220]

We start with two tables, A_l and A_r for the nodes l and r that get joined into node t. The tables A_l and A_r, that use state set S_1 will be transformed to use state set S_2 using the above method. Then comes the actual joining of the two tables. In Figure 2, the join tables for combining two tables using state set S_2 is shown.

	1	$0_?$	0_0
1	1		
$0_?$		$0_?$	
0_0			0_0

Figure 2: *Join table for the join operation of* MINIMUM DOMINATING SET *with state set* $\{1, 0_?, 0_0\}$.

In order to compute the join operation now, we will simply be using the standard method. This means that the complexity of joining A_l and A_r using state set S_2 is $O^*(3^k)$, which is significantly less than our previous method. In particular, we can determine the values for specific entries in the table using the following formula:

$$A_x(c, \kappa) = \sum_{\kappa_l + \kappa_r = \kappa + \#_1(c)} A_l(c, \kappa_l) \cdot A_r(c, \kappa_r)$$

Since for this state set, colorings match one-to-one, we can multiply the two values from either side for each coloring. Since these values represent the amount of partial solutions on either side, and the shared vertices between the two (i.e. the ones in the bag B_t) have the same coloring, we can simply multiply these two values together to obtain our new value. The constraint on the summand is used in order to make sure that the correct combinations of solutions are assigned to the correct values of κ, which determines the size of a solution. Since the vertices in the bag B_t are going to be present in solutions in both the left and right sides of a decomposition, they will be counted twice in any combination, and thus we need to account for this, which is done by the term $\#_1(c)$. Alltogether, this comes down to a time complexity of $O(n^2 3^k)$.

After this, we simply change our table back to the original state set, and we can continue on with the rest of the dynamic programming algorithm. The time complexity of performing the join operation for MINIMUM DOMINATING SET in this manner is $O^*(3^k)$, since this is the time complexity for the two state changes and for the joining of the state changed tables.

3.3 Counting tricks

We will now examine an instance of a counting trick which may be required in order to compute certain join operations after a state change. In this case, the counting trick involves remembering certain properties of a coloring when applying a state change. These properties can then be used to compute the join operation.

This time we will consider a new problem, PERFECT MATCHINGS. In particular, we will be counting the number of perfect matchings in a graph. For this problem, we will be using a state set consisting of two states: $S_1 = \{0, 1\}$. The state 0 represents a vertex that is not matched yet, and the state 1 represents a vertex that is matched. The join table for this problem with this state set is shown in Figure 3. Combining the states 1 and 1 results in an invalid combination because every vertex must be matched to precisely one other vertex, thus we assign $j_{1,1}$ the empty combination $\times$.

The table for node t is indexed simply by a coloring c, and the values they hold represent the

	0	1
0	0	1
1	1	

	0	?
0	0	
?		$\not{?}$

Figure 3: *Join matrix for a join computation of Perfect Matching using the state sets $\{0, 1\}$ and $\{0, ?\}$. The symbol $\not{?}$ indicates that we need to be careful when performing the state combinating $j_{?,?}$ in S_2, since we need to take care of the situation where we combine two vertices that are already matched.*

amount of perfect matchings in which the vertices of B_t that have the state 0 are unmatched, and the vertices with state 1 are matched to a vertex that has been forgotten in a Forget node.

Using the standard method, computing this join operation has a complexity of $O^*(3^k)$. However, this can be improved by using a state change to the state set $S_2 = \{0, ?\}$. In this state set, the state ? represents a vertex that may be matched or not. The state change works in a similar manner to the state change we used for Minimum Dominating Set, so we will not cover that here.

Figure 3 shows the join table with the state set S_2. Only two combinations are legal in this join table, which would give way to an algorithm with complexity $O^*(2^k)$. While the states 0 and ? can be sensibly combined to the state ?, such a combination is not performed. With the current combinations in the join table, all possible combinations from the original state set $\{0, 1\}$ are made, so no information is lost. In addition, the current combinations are all we need in order to convert back to the original state set of $\{0, 1\}$. The combinations between the states 0 and ? would therefore not add any information, nor are they necessary to perform the reverse state change, so they are no performed.

There is a slight complication in this join table however, namely the state combination $j_{?,?}$ is assigned the state $\not{?}$. This is because a problem exists with simply combining two ? states. Namely, we had previously established that combining two 1 states is not allowed, but simply combining two ? states would not take this restriction into account. We will be using a counting trick in order to deal with this problem.

For the computations for the Leaf, Introduce and Forget nodes, we again refer to [9][section 11.4, p.222-225]. Instead of immediately applying a state change to the tables, we will first adjust them slightly. For a table $A_d(c)$, with $d \in \{l, r\}$, we will compute the table $A'_d(c, i)$, where i represents the number of vertices in B_d with state 1. We then define:

$$A'_d(c, i) = \begin{cases} A_d(c) & \text{if } \#_1(c) = i \\ 0 & \text{otherwise} \end{cases}$$

We then apply the state change to A'_l and A'_r instead of the original tables. The state changed tables, while they do not use the state 1 in their state set, still retain information from the previous representation via the index i. This is what will allow us to compute the join operation while also making sure that two vertices with state 1 do not get matched together. The formula we will use to compute this join operation is:

$$A'_t(c,i) = \sum_{i_l+i_r=i} A'_l(c,i_l) \cdot A'_r(c,i_r)$$

After this operation, our table $A'_t(c,i)$ now contains the total number of partial solutions such that vertices with state 0 are unmatched, and vertices with state ? are matched either twice, once or not at all.

We can apply a state change again in order to revert the table A'_t back to state set S_1. Due to allowing the ? state in the previous representation to represent a vertex that is matched either twice, once or not at all, the 1 state now represents a vertex that is matched either twice or once. However, we can use the index i to determine the correct entries for the table A_t. Namely, the entry $A'_t(c, \#_1(c))$ holds the correct value of partial matchings in which no vertex is matched twice.

In this way, we can use a counting trick to compute the join operation for counting perfect matchings with a complexity of $O^*(2^k)$.

4 The 2 × 2 join operations

In this section, we will try to classify the join operations of size 2×2, using the general state set $S = \{0, 1\}$. We would like to determine the time complexities of these 2×2 join operations, with particular interest for the join operations that can be computed in $O^*(2^k)$. The following result is therefore quite a nice one.

Theorem 4.1. *All* 2×2 *join operations can be computed in* $O^*(2^k)$*.*

A join operation of size 2×2 has 3 possible state combinations, $\mathcal{J}_S = \{j_{0,0}, j_{0,1}, j_{1,1}\}$. The values we can assign are those in the set $S \cup \{\times\}$, where $\times$ represents the empty combination. Therefore, the amount of different 2×2 join operations would seemingly be $3^3 = 27$.

However, there are some equivalences that can be established between join operations. In particular, the states assigned to the state combinations by the join operation do not carry any specific meaning, apart from the empty combination $\times$. Other than that though, the states 0 and 1 don't differ from each other, so it stands to reason that a join operation where the states 0 and 1 are switched would be calculated in the same manner as the original join operation.

In fact, we can be a bit more specific in how we switch the states 0 and 1 in a join operation. We can choose to switch the states only for the input colorings or only for the output colorings. In both cases we can also calculate these new join operations by using the calculation for the original join operations by inverting the colorings in the tables before and after the join operation is performed respectively. The equivalence classes formed by this equivalence relation all have a size of at most 4, such as the one shown in Figure 4.

This equivalence relation allow us to reduce the amount of join operations that we need to check the time complexity of. The amount of equivalence classes these equivalence relations give rise to is 10. A representative of each equivalence class is shown in Figure 5.

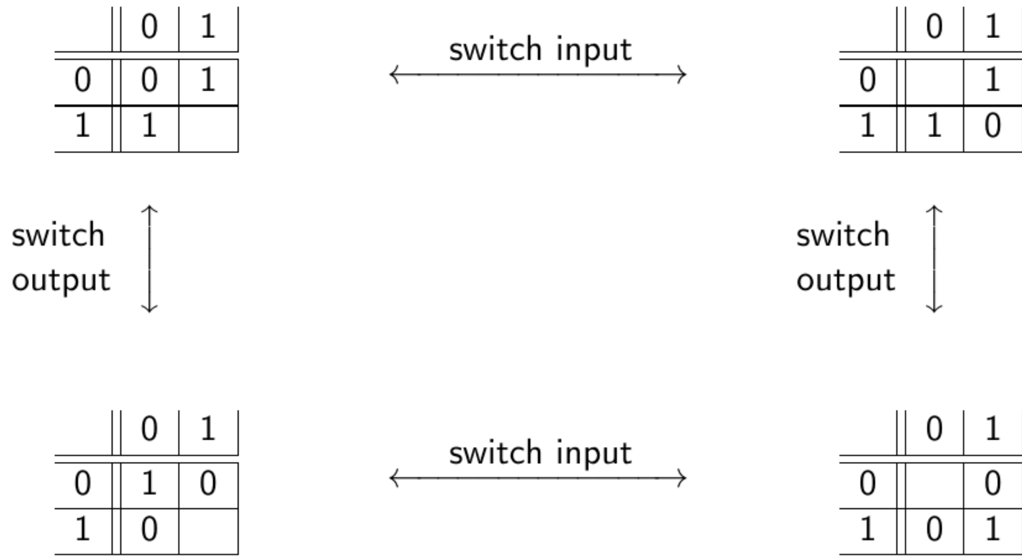

Figure 4: *One of the equivalence classes for 2×2 join operations*

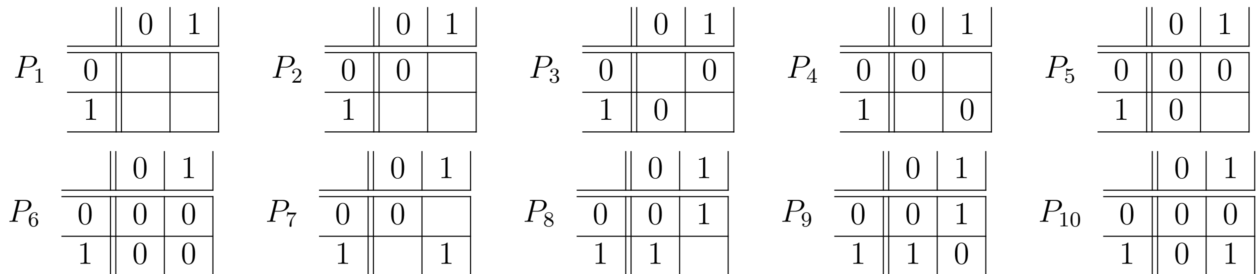

Figure 5: *The representatives of the ten equivalence classes of 2×2 join operations*

	?	1
?	?	
1		1

	?	1
?	?	
1		

Figure 6: *Join operations P_5 and P_6 after being state changed to state set $\{?\,,1\}$.*

	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

Figure 7: *Join table for 3×3 Fourier-Transform-based join operation with $S = \mathbb{Z}_3$.*

Thus, we only need to find out the time complexity for these 10 different join operations. It can be noted immediately that join tables P_1, P_2, P_3, P_4 and P_7 can be computed in $O^*(2^k)$. Since only two combinations are non-empty in these tables, the time complexity for performing these join operations is trivially $O^*(2^k)$, as discussed in Section 3.1. For join operation P_8, we can refer to Section 3.3, where this operation was shown to have a time complexity of $O^*(2^k)$.

The state change technique discussed in Section 3.2 can be used to perform join operations P_5 and P_6 in time complexity $O^*(2^k)$. Namely, we perform a state change from the state set $\{0, 1\}$ to the state set $\{?\,,1\}$, where the state ? implies that a vertex can have either state 0 or 1. The join tables for P_5 and P_6 using this new state set are shown in Figure 6. Both join operations only have 2 non-empty combinations, and thus join operations P_5 and P_6 can be computed in $O^*(2^k)$ as well.

This leaves join operations P_9 and P_{10}. The general-form $s \times s$ versions of these join operations are proven to have time complexity $O^*(s^k)$ in Sections 5 and 6, and thus the 2×2 versions can be computed in $O^*(2^k)$. With this, we have shown that every 2×2 join operation can be computed with time complexity $O^*(2^k)$. □

5 Fourier-Transform-based join operation

The $s \times s$ Fourier-Transform-based join operation, denoted by FT_s, is one of the join operations that can be performed in $O^*(s^k)$ time complexity, for $s \geq 2$. The $s \times s$ Fourier-Transform-based join operation combines two states set of size s in such a way that, if the states were numbered from 0 to $s - 1$, the resulting state would be the sum of those states taken modulo s. In other words, $FT_s(j_{p,q}) = p + q \pmod{s}$ where $S = \mathbb{Z}_s$, the cyclic group of size s, and $j_{p,q} \in \mathcal{J}_S$. The join table for the 3×3 Fourier-Transform-based join is given Figure 7.

The standard way to calculate this transform described in Section 3.1 would result in a time complexity of $O^*(s^{2k})$, but using an s-dimensional Fourier Transform, this time complexity can be lowered to $O^*(s^k)$. This method of computing the Fourier-Transform-based join operation was first introduced by Cygan et al. [11], but only FT_2 and FT_4 were actually proven to have time complexities of $O^*(2^k)$ and $O^*(4^k)$ respectively.

The Fourier Transform used here is basically a state change which changes our states to states in the Fourier Domain. In the following formulas, A will represent the table we are state changing and B will be the current bag. We use $\mathbb{Z}_s$ to denote the state set, meaning that $\mathbb{Z}_s^B$ is the set of all colorings. For a vertex $v \in B$ and a coloring $x \in \mathbb{Z}_s^B$, we retain our definition of $x(v)$ to refer to the color the vertex v takes on in coloring x. However, for these state change definitions, we also use x_b for $0 \leq b \leq |B|-1$ to refer to the b-th vertex in our coloring x. We define the inner product $x \cdot y = \sum_{v \in B} x(v) \cdot y(v)$ for $x, y \in \mathbb{Z}_s^B$. The term ϵ is the s-th root of unity, meaning that $\epsilon^s = 1$.

Just like with a state change, we will be transforming each vertex in our bag separately, resulting in $|B|$ different steps. For $x \in \mathbb{Z}_s^B$ and $0 \leq b \leq |B|-1$, we define:

$$\tilde{A}_{b+1}(x_1, \ldots, x_{b+1}, y_{b+2}, \ldots, y_{|B|}) = \sum_{y_{b+1} \in \mathbb{Z}_s} \tilde{A}_b(x_1, \ldots, x_b, y_{b+1}, \ldots, y_{|B|}) \epsilon^{x_{b+1} \cdot y_{b+1}}$$

This recursive definition of a table basically performs a Fourier Transform on each vertex of the current bag. The table $\tilde{A}_{b+1}$ represents a table with colorings that are made up of new states for the first $b+1$ vertices, and old states for the other vertices. In our definition, we use the coloring x to denote a coloring in the new states and the coloring y to denote a coloring in the old states. The table $\tilde{A}_0$ contains only old states and is therefore equal to A. The table $\tilde{A}_{|B|}$ will contain only new vertices, and thus this is the table after the state change is fully performed and we denote it by $\tilde{A}$. The table $\tilde{A}$ can also be written as:

$$\tilde{A}(x) = \sum_{y \in \mathbb{Z}_s^B} A(y) \cdot \epsilon^{x \cdot y} \tag{1}$$

The inverse state change is very similar, but has a couple of slight changes. The inverse state change is basically an inverse Fourier Transform, and is also performed on each vertex in succession. We denote this state change by $\bar{A}$, and the recursive table $\bar{A}_{b+1}$ for $0 \leq b \leq |B|-1$ is defined as:

$$\bar{A}_{b+1}(x_1, \ldots, x_{b+1}, y_{b+2}, \ldots, y_{|B|}) = \frac{1}{s} \sum_{y_{b+1} \in \mathbb{Z}_s} \bar{A}_b(x_1, \ldots, x_b, y_{b+1}, \ldots, y_{|B|}) \epsilon^{-x_{b+1} \cdot y_{b+1}}$$

Similar to $\tilde{A}$, we define $\bar{A}_0 = A$ and $\bar{A}_{|B|} = \bar{A}$. We also now know that $\bar{A}$ is equal to:

$$\bar{A}(x) = \frac{1}{s^{|B|}} \sum_{y \in \mathbb{Z}_s^B} A(y) \cdot \epsilon^{-x \cdot y} \tag{2}$$

Given b such that $0 \leq b \leq |B|-1$ and $\tilde{A}_b$ has already been calculated. For all entries in $\tilde{A}_b$, which are at most s^k, a sum with s summands is calculated, which results in s^{k+1} steps, giving

a general time complexity of $O(s^k)$. This is repeated for k vertices in our bag, which results in a final time complexity of $O(ks^k)$ for calculating $\tilde{A}$. The same argument holds for calculating $\bar{A}$, so it also has a time complexity of $O(ks^k)$.

This concludes the explanation of the state changes that we will used in our proofs. We will first prove the following lemma, which will be used later:

Lemma 5.1. *For $y \in \mathbb{Z}_s$,*

$$\sum_{x \in \mathbb{Z}_s} (\epsilon^y)^x = \begin{cases} s & \text{when } y = 0 \\ 0 & \text{when } y \neq 0 \end{cases}$$

where ϵ is the s-th root of unity.

When $y = 0$, the value of ϵ^y is 1. Then we get the sum:

$$\sum_{x \in \mathbb{Z}_s} 1 \text{ when } y = 0$$

which is equal to the amount of terms being summed, which in this case is $|\mathbb{Z}_s| = s$.

When $y \neq 0$, we can instead use a well-known identity for geometric sequences. Namely:

$$\sum_{i=0}^{k} r^i = \frac{1 - r^k}{1 - r} \text{ when } r \neq 1$$

which implies that:

$$\sum_{x \in \mathbb{Z}_s} (\epsilon^y)^x = \frac{1 - (\epsilon^y)^s}{1 - \epsilon^y} \text{ when } \epsilon^y \neq 1$$

Since we know that $y \neq 0$, we know that $\epsilon^y \neq 1$ as required. We also know that $(\epsilon^y)^s = (\epsilon^s)^y = 1^y = 1$ since ϵ is the s-th root of unity. Therefore, we can now determine that:

$$\sum_{x \in \mathbb{Z}_s} (\epsilon^y)^x = \frac{1 - 1}{1 - \epsilon^y} = 0 \text{ when } y \neq 0$$

which proves the lemma. □

Now we can use this lemma to first show that our two formulas defined in equations (1) and (2) are inverses.

Theorem 5.2. $\tilde{\tilde{A}} = A$

Our first steps will be to work out $\tilde{\tilde{A}}$. For $x \in \mathbb{Z}_s^B$:

$$\begin{aligned}
\tilde{\tilde{A}}(x) =& \overline{\sum_{y \in \mathbb{Z}_s^B} A(y)\epsilon^{x \cdot y}} \\
=& \frac{1}{s^{|B|}} \sum_{z \in \mathbb{Z}_s^B} \left(\sum_{y \in \mathbb{Z}_s^B} A(y)\epsilon^{z \cdot y} \right) \epsilon^{-x \cdot z} \\
=& \frac{1}{s^{|B|}} \sum_{y \in \mathbb{Z}_s^B} A(y) \sum_{z \in \mathbb{Z}_s^B} \left(\epsilon^{(y-x)}\right)^z
\end{aligned}$$

The steps above are simple rewritings of $\tilde{\tilde{A}}$ using equations (1) and (2).

$$= \frac{1}{s^{|B|}} \sum_{\substack{y \in \mathbb{Z}_s^B \\ y = x}} A(y) \sum_{z \in \mathbb{Z}_s^B} \left(\epsilon^{(y-x)}\right)^z + \frac{1}{s^{|B|}} \sum_{\substack{y \in \mathbb{Z}_s^B \\ y \neq x}} A(y) \sum_{z \in \mathbb{Z}_s^B} \left(\epsilon^{(y-x)}\right)^z$$

Here, we split the equation into two parts: one where the sum over y has the requirement that $y = x$, and one where it has the opposite requirement $y \neq x$. These two requirements cover every possibility for y, so it is clear that it is equal to what came before it. When $y = x$, the exponent $y - x$ will be equal to 0, whereas when $y \neq x$, the exponent will not be equal to 0. Having written the formula in this way allows us to then apply Lemma 5.1:

$$\begin{aligned}
=& \frac{1}{s^{|B|}} \sum_{\substack{y \in \mathbb{Z}_s^B \\ y = x}} A(y) \cdot n^{|B|} + \frac{1}{s^{|B|}} \sum_{\substack{y \in \mathbb{Z}_s^B \\ y \neq x}} A(y) \cdot 0 \\
=& \frac{1}{s^{|B|}} s^{|B|} \sum_{\substack{y \in \mathbb{Z}_s^B \\ y = x}} A(y) \\
=& \sum_{\substack{y \in \mathbb{Z}_s^B \\ y = x}} A(y) \\
=& A(x)
\end{aligned}$$

which proves that $\tilde{\tilde{A}} = A$. □

Now we will try to prove that these state changes allow us to compute the Fourier-Transform-based join operation:

Theorem 5.3. *The $s \times s$ Fourier-Transform-based join can be computed in $O^*(s^k)$.*

The idea behind this way of calculating the Fourier-Transform-based join is the perform the state change defined in $\tilde{A}$, perform a pointwise multiplication on the resulting colorings, and then perform the state change back defined in $\bar{A}$. Thus, we plan to work out $\overline{\tilde{A}_l(x) \cdot \tilde{A}_r(x)}$ for $x \in \mathbb{Z}_s^B$.

$$\begin{aligned}
\overline{\tilde{A}_l(x) \cdot \tilde{A}_r(x)} &= \frac{1}{s^{|B|}} \sum_{z \in \mathbb{Z}_s^B} \tilde{A}_l(z) \cdot \tilde{A}_r(z) \cdot \epsilon^{-z \cdot x} \\
&= \frac{1}{s^{|B|}} \sum_{z \in \mathbb{Z}_s^B} \left(\sum_{y_1 \in \mathbb{Z}_s^B} f_L(y_1) \cdot \epsilon^{z \cdot y_1} \right) \cdot \left(\sum_{y_2 \in \mathbb{Z}_s^B} f_R(y_2) \epsilon^{z \cdot y_2} \right) \cdot \epsilon^{-z \cdot x} \\
&= \frac{1}{s^{|B|}} \sum_{y_1, y_2 \in \mathbb{Z}_s^B} A_l(y_1) \cdot A_r(t_2) \sum_{z \in \mathbb{Z}_s^B} \epsilon^{z \cdot (y_1 + y_2 - x)}
\end{aligned}$$

All the above steps can be obtained by rearranging and using the definitions in equations (1) and (2).

$$= \frac{1}{s^{|B|}} \sum_{y_1, y_2 \in \mathbb{Z}_s^B} A_l(y_1) \cdot A_r(y_2) \prod_{v \in B} \left(\sum_{z(v) \in \mathbb{Z}_s} \left(\epsilon^{y_1(v) + y_2(v) - x(v)} \right)^{z(v)} \right)$$

In the above step, we simply work in terms of the individual states in a coloring, rather than the full coloring itself.

We know from Lemma 5.1 that when $y_1(v) + y_2(v) - x(v) = 0$ in the nested sum above, the value of that sum is equal to s, and otherwise it is equal to 0. This can be expressed with the following formula.

$$= \frac{1}{s^{|B|}} \sum_{y_1, y_2 \in \mathbb{Z}_s^B} A_l(y_1) \cdot A_r(y_2) \prod_{v \in B} \left(s \cdot [y_1(v) + y_2(v) - x(v) = 0] \right)$$

Again, the notation in the term $[y_1(b) + y_2(b) - x(b) = 0]$ simply means that if the statement inside the square brackets is true, then the value of of that term is equal to 1, and otherwise it is equal to 0.

$$= \frac{1}{s^{|B|}} \sum_{y_1, y_2 \in \mathbb{Z}_s^B} A_l(y_1) \cdot A_r(y_2) \cdot s^{|B|} \cdot [y_1 + y_2 - x = 0]$$
$$= \frac{1}{s^{|B|}} s^{|B|} \sum_{y_1, y_2 \in \mathbb{Z}_s^B} A_l(y_1) \cdot A_r(y_2) \cdot [y_1 + y_2 = x]$$
$$= \sum_{\substack{y_1, y_2 \in \mathbb{Z}_s^B \\ y_1 + y_2 = x}} A_l(y_1) \cdot A_r(y_2)$$

The above steps are simple rewritings. Bringing everything together, we arrive at:

$$\overline{(\tilde{A}_l(x) \cdot \tilde{A}_r(x))} = \sum_{\substack{y_1, y_2 \in \mathbb{Z}_s^B \\ y_1 + y_2 = x}} A_l(y_1) \cdot A_r(y_2)$$

We can see that the right-hand side here is exactly the Fourier-Transform-based join operation. For a given coloring x, the sum on the right-hand side only allows precisely those pairs of colorings in $\mathbb{Z}_s^B$ that are equal to x when added together in $\mathbb{Z}_s$.

Now we only need to determine the time complexity of this algorithm. Joining the table with changed states takes s^k steps (the amount of entries in the two tables, since we simply perform an entry-wise multiplication). The two state changes have already been shown to have $O(ks^k)$ time complexity, thus our final time complexity is $O(ks^k)$. □

6 Möbius-Transform-based join operation

The $s \times s$ Möbius-Transform-based join operation, denoted by MT_s, is another join operation that can be performed in $O^*(s^k)$ time complexity, for $n \geq 2$. The Möbius-Transform-based join effectively combines two states in such a way that resulting state is equal the highest value between the two combined states. In other words, $MT_s(j_{p,q}) = \max(p, q)$, where $S = \mathbb{Z}_s$, the cyclic group of size s, and $j_{p,q} \in \mathcal{J}_S$.

We perform the Möbius-Transform-based join operation by performing a state change. This state change allows for the Möbius-Transform-based join operation to be computed in $O^*(s^k)$, as opposed to the standard algorithm which would give a time complexity of $O^*(s^{2k})$.

We will begin by defining the two state sets S_1 and S_2. The state set S_1 simply has s states enumerated from 0 to $s-1$. The state set S_2 instead has s states denoted by $a_{\leq}$ for $a \in \mathbb{Z}_s - 0$, and has one extra state of 0. The state 0 in S_2 is equivalent to the state 0 in S_1, but a state of the form $a_{\leq}$ indicates that the vertex in question could have any value from 0 to a. Both state sets are represented by $\mathbb{Z}_s$ in the following proofs.

The join tables for the 3×3 Möbius-Transform-based join operation in both state sets is shown in Figure 8, in both S_1 and S_2 .

	0	1	2
0	0	1	2
1	1	1	2
2	2	2	2

	0	$1_\leq$	$2_\leq$
0	0		
$1_\leq$		$1_\leq$	
$2_\leq$			$2_\leq$

Figure 8: *3×3 join table for Möbius-Transform-based join operation, using both S_1 and S_2.*

Now we will look at how the actual state changes will work. As is usual for state changes, we will perform the state change by iterating over the table we're state changing one time for each vertex in our bag B. In the change from S_1 to S_2, changing to a state of the form a_{leq} requires us to consider all states from S_1 lower than or equal to a in value. For $x \in \mathbb{Z}_s^B$ and $0 \leq b \leq |B|-1$, we define:

$$\tilde{A}_{b+1}(x_1, \ldots, x_{b+1}, y_{b+2}, \ldots, y_{|B|}) = \sum_{\substack{y_{b+1} \in \mathbb{Z}_s \\ y_b+1 \leq x_b+1}} \tilde{A}_b(x_1, \ldots, x_b, y_{b+1}, \ldots, y_{|B|})$$

The definition of x_b is the same as in Section 5. We say that $x \leq y$ if, for all $v \in B$, $x(v) \leq y(v)$. We also define $x-1$ as the coloring where $(x-1)(v) = x(v) - 1$. In the event that $x(v) = 0$, we especially define the value of $(x-1)(v) = -1$ in order to avoid complications later on. We define $\tilde{A}_0$ to be equal to A, which is the table we are performing the state change on. The table $\tilde{A}_{|B|}$ is therefore the table after the state change is complete. Using the above formula, we obtain that the value of $\tilde{A}_{|B|}$, which we denote by $\tilde{A}$, is:

$$\tilde{A}(x) = \sum_{\substack{y \in \mathbb{Z}_s^B \\ y \leq x}} A(y) \tag{3}$$

We will use a similar approach to do the reverse state change. In this case, we want to change from S_2 to S_1. We can obtain a state a from S_1 by calculating the difference $a_\leq - (a-1)_\leq$. This leads us to the following recursive formula:

$$\bar{A}_{b+1}(x_1, \ldots, x_{b+1}, y_{b+2}, \ldots, y_{|B|}) = \sum_{\substack{y_{b+1} \in \mathbb{Z}_s \\ (x-1)_{b+1} \leq y_{b+1} \leq x_{b+1} \\ \mathbf{0} \leq y_b}} \bar{A}_b(x_1, \ldots, x_b, y_{b+1}, \ldots, y_{|B|})(-1)^{[y_{b+1}=(x-1)_{b+1}]}$$

We again define $\bar{A}_0$ to be equal to A. The symbol $\mathbf{0}$ refers to the coloring that has the state 0 for every vertex. The term $[y_b = (x-1)_b]$ is equal to 1 if and only if the equation inside the square brackets is true, and is equal to 0 otherwise. The final table after the state change is again given by $\bar{A}_{|B|}$, which we denote by $\bar{A}$, and is equal to:

$$\bar{A}(x) = \sum_{\substack{y \in \mathbb{Z}_s^B \\ (x-1) \leq y \leq x \\ \mathbf{0} \leq y}} A(y)(-1)^{\sum_{v \in B}[y(v)=(x-1)(v)]} \tag{4}$$

The time complexity calculation for $\tilde{A}$ and $\bar{A}$ are almost identical to those in Section 5, but we will repeat it here. Given b such that $0 \leq b \leq |B|-1$ and $\tilde{A}_b$ has already been calculated. For all entries in $\tilde{A}_b$ and $\bar{A}_b$, which are at most s^k, a sum is calculated. For $\tilde{A}_b$, the sum has at most s summands, and for $\bar{A}_b$, the sum has 2 summands. In both cases, this results in a time complexity of $O(s^k)$. This is repeated for k vertices in our bag, which results in a final time complexity of $O(ks^k)$ for calculating $\tilde{A}$ and $\bar{A}$.

We will again first prove a lemma which will end up being quite useful.

Lemma 6.1. *For* $x, z \in \mathbb{Z}_s^B$,

$$\sum_{\substack{y \in \mathbb{Z}_s^B \\ (x-1) \leq y \leq x \\ z \leq y}} (-1)^{\sum_{v \in B}[y(v)=(x-1)(v)]} = \begin{cases} 1 & \text{when } z = x \\ 0 & \text{when } z \neq x \end{cases}$$

When z is equal to x the above sum is equal to 1. This is because of the requirement placed on y that $z \leq y$. Since $z = x$, this also means that $x \leq y$. Due to this, none of the vertices in the coloring y can possibly have the same state as that of the coloring $x - 1$. This causes the term $\sum_{v \in B}[y(v) = (x-1)(v)]$ to equal 0, which causes the summand to take on the value 1. The sum also has the requirement that $y \leq x$, which when combined with $x \leq y$ shows that y can only take on one value, namely that of the coloring x. This means that the summand 1 is only counted once, causing the final value of this sum to be 1.

When $z \leq x$ and $z \neq x$, we need to do a bit more work. We first split the sum up into two sums.

$$\sum_{\substack{y \in \mathbb{Z}_s^B \\ (x-1) \leq y \leq x \\ z \leq y}} (-1)^{\sum_{v \in B}[y(v)=(x-1)(v)]} = \sum_{i=0}^{\sum_{v \in B}[z(v) \leq (x-1)(v)]} \sum_{\substack{y \in \mathbb{Z}_s^B \\ (x-1) \leq y \leq x \\ z \leq y \\ \sum_{v \in B}[y(v)=(x-1)(v)] = \sum_{v \in B}[z(v) \leq (x-1)(v)] - i}} (-1)^{\sum_{v \in B}[y(v)=(x-1)(v)]}$$

The sum is split up so that the inner sum now only looks at colorings y such that the term $\sum_{v \in B}[y(v) = (x-1)(v)]$ is equal to the value of $\sum_{v \in B}[z(v) = (x-1)(v)] - i$, which is determined by the outer sum and runs from $\sum_{v \in B}[z(v) \leq (x-1)(v)]$ to 0. Therefore we can be certain that this step has not changed the value of the term we were rewriting. We have to use the statement $z(v) \leq (x-1)(v)$ rather than $z(v) = (x-1)(v)$ because, unlike the coloring y, there is no constraint that $x - 1 \leq z$.

Now let's say that in the inner sum, the value $\sum_{v \in B}[z(v) \leq (x-1)(v)] - i$ is equal to some value α. Then the nested sum is equal to $(-1)^\alpha$ multiplied by the amount of different colorings y that conform to the requirements set in the sum's subscript. Due to the requirement of $(x-1) \leq y \leq x$, we know that each $y(v)$ has to be equal to either $(x-1)(v)$ or $x(v)$. Since $z \leq y$, the vertices v where $y(v)$ can be equal to either $(x-1)(v)$ or $x(v)$ are precisely the vertices for which $z(v) \leq (x-1)(v)$. Thus the amount of vertices v where $y(v)$ can end up being equal to $(x-1)(v)$ is equal to $\sum_{v \in B}[z(v) \leq (x-1)(v)]$.

As mentioned, for each of these vertices the state $y(v)$ has to be equal to either $(x-1)(v)$ or $x(v)$. The amount of colorings y that conform to the requirements set by the sum is precisely the amount of subsets of size α that can be made out of the $\sum_{v\in B}[z(v) \leq (x-1)(v)]$ vertices. This number can be found using the binomial coefficient. Specifically, it's equal to:

$$\binom{\sum_{v\in B}[z(v) \leq (x-1)(v)]}{\alpha} = \binom{\sum_{v\in B}[z(v) \leq (x-1)(v)]}{\sum_{v\in B}[z(v) \leq (x-1)(v) - i} = \binom{\sum_{v\in B}[z(v) \leq (x-1)(v)]}{i}$$

Which we can then plug back into the equation we left off at:

$$= \sum_{i=0}^{\sum_{v\in B}[z(v)\leq(x-1)(v)]} \binom{\sum_{v\in B}[z(v) \leq (x-1)(v)]}{i} (-1)^{\sum_{v\in B}[z(v)\leq(x-1)(v)]-i}$$

For $c, d \in \mathbb{R}$ it holds that $1^c = 1$ and $(-1)^{c-d} = (-1)^{d-c}$. Using this, we can rewrite this formula as follows:

$$= \sum_{i=0}^{\sum_{v\in B}[z(v)\leq(x-1)(v)]} \binom{\sum_{v\in B}[z(v) \leq (x-1)(v)]}{i} (-1)^{i-\sum_{v\in B}[z(v)\leq(x-1)(v)]} (1)^{\sum_{v\in B}[z(v)\leq(x-1)(v)]}$$

And due to Newton's Binomium, this is equal to:

$$= (-1+1)^{\sum_{v\in B}[z(v)\leq(x-1)(v)]} = 0$$

It should be noted that since $z \leq x$ and $z \neq x$, the term $\sum_{v\in B}[z(v) \leq (x-1)(v)]$ will not equal 0. Therefore the above statement is always defined, and we have proven the Lemma. □

Using this, we can show that the two equations in (3) and (4) are inverses of each other in the following theorem:

Theorem 6.2. $\bar{\tilde{A}} = A$

We will start by trying to rewrite $\bar{\tilde{A}}$ in terms of the definitions in (3) and (4).

$$\begin{aligned}
\bar{\tilde{A}} &= \sum_{\substack{y\in\mathbb{Z}_s^B \\ (x-1)\leq y\leq x \\ \mathbf{0}\leq y}} \tilde{A}(y)(-1)^{\sum_{v\in B}[y(v)=(x-1)(v)]} \\
&= \sum_{\substack{y\in\mathbb{Z}_s^B \\ (x-1)\leq y\leq x \\ \mathbf{0}\leq y}} \left(\sum_{\substack{z\in\mathbb{Z}_s^B \\ z\leq y}} A(z) \right) (-1)^{\sum_{v\in B}[y(v)=(x-1)(v)]}
\end{aligned}$$

Then we switch the order of the two nested sums:

$$= \sum_{\substack{z \in \mathbb{Z}_s^B \\ z \leq x}} A(z) \sum_{\substack{y \in \mathbb{Z}_s^B \\ (x-1) \leq y \leq x \\ z \leq y}} (-1)^{\sum_{v \in B} [y(v) = (x-1)(v)]}$$

The nested sum above looks a lot like something that we can use Lemma 6 for, but we will first need to split it in two sums as follows:

$$\begin{aligned} &= \sum_{\substack{z \in \mathbb{Z}_s^B \\ z = x}} A(z) \sum_{\substack{y \in \mathbb{Z}_s^B \\ (x-1) \leq y \leq x \\ z \leq y}} (-1)^{\sum_{v \in B} [y(v) = (x-1)(v)]} \\ &+ \sum_{\substack{z \in \mathbb{Z}_s^B \\ z \leq x \\ z \neq x}} A(z) \sum_{\substack{y \in \mathbb{Z}_s^B \\ (x-1) \leq y \leq x \\ z \leq y}} (-1)^{\sum_{v \in B} [y(v) = (x-1)(v)]} \end{aligned}$$

As you can see, we have split the sum into precisely the two cases described in Lemma , which we can use to arrive at:

$$\begin{aligned} &= \sum_{\substack{z \in \mathbb{Z}_s^B \\ z = x}} A(z) \cdot 1 + \sum_{\substack{z \in \mathbb{Z}_s^B \\ z \leq x \\ z \neq x}} A(z) \cdot 0 \\ &= \sum_{\substack{z \in \mathbb{Z}_s^B \\ z = x}} A(z) \\ &= f(x) \end{aligned}$$

which proves the lemma. □

And now we can continue with the following Theorem.

Theorem 6.3. *The Möbius-Transform-based join can be computed in $O^*(s^k)$.*

Given a coloring $x \in \mathbb{Z}_s^B$. We will work out the result of the state change with (3) and (4). This comes down to working out the value of $\overline{\tilde{A}_l(x) \cdot \tilde{A}_r(x)}$. We can rewrite this as follows:

$$
\begin{aligned}
\overline{(\tilde{A}_l(x)\cdot\tilde{A}_r(x))} &= \overline{\left(\sum_{\substack{z_1\in\mathbb{Z}_s^B\\ z_1\le x}} A_l(z_1)\right)\cdot\left(\sum_{\substack{z_2\in\mathbb{Z}_s^B\\ z_2\le x}} A_r(z_2)\right)} \\
&= \overline{\sum_{\substack{z_1,z_2\in\mathbb{Z}_s^B\\ z_1,z_2\le x}} A_l(z_1)\cdot A_r(z_2)} \\
&= \sum_{\substack{y\in\mathbb{Z}_s^B\\ (x-1)\le y\le x\\ \mathbf{0}\le y}} \left(\sum_{\substack{z_1,z_2\in\mathbb{Z}_s^B\\ z_1,z_2\le y}} A_l(z_1)\cdot A_r(y_2)\right)(-1)^{\sum_{v\in B}[y(v)=(x-1)(v)]} \\
&= \sum_{\substack{y\in\mathbb{Z}_s^B\\ (x-1)\le y\le x\\ \mathbf{0}\le y}} (-1)^{\sum_{v\in B}[y(v)=(x-1)(v)]}\cdot \sum_{\substack{z_1,z_2\in\mathbb{Z}_s^B\\ z_1,z_2\le z}} A_l(z_1)\cdot A_r(y_2)
\end{aligned}
$$

This follows from definitions (3) and (4).

$$
= \sum_{\substack{z_1,z_2\in\mathbb{Z}_s^B\\ z_1,z_2\le x}} A_l(z_1)\cdot A_r(z_2)\cdot \sum_{\substack{y\in\mathbb{Z}_s^B\\ (x-1)\le y\le x\\ z_1,z_2\le y}} (-1)^{\sum_{v\in B}[y(v)=(x-1)(v)]}
$$

Here we switch the order of the two sums, and rewrite the subscripts accordingly.

$$
\begin{aligned}
= &\sum_{\substack{z_1,z_2\in\mathbb{Z}_s^B\\ z_1,z_2\le x\\ \mathrm{cmax}(z_1,z_2)=x}} A_l(z_1)\cdot A_r(z_2)\cdot \sum_{\substack{y\in\mathbb{Z}_s^B\\ (x-1)\le y\le x\\ \mathrm{cmax}(z_1,z_2)\le y}} (-1)^{\sum_{v\in B}[y(v)=(x-1)(v)]} \\
+ &\sum_{\substack{z_1,z_2\in\mathbb{Z}_s^B\\ z_1,z_2\le x\\ \mathrm{cmax}(z_1,z_2)\le x\\ \mathrm{cmax}(z_1,z_2)\ne x}} A_l(z_1)\cdot A_r(z_2)\cdot \sum_{\substack{y\in\mathbb{Z}_s^B\\ (x-1)\le y\le x\\ \mathrm{cmax}(z_1,z_2)\le y}} (-1)^{\sum_{v\in B}[y(v)=(x-1)(v)]}
\end{aligned}
$$

We define $\mathrm{cmax}(z_1, z_2) \in \mathbb{Z}_s^B$ as the component-wise maximum of z_1 and z_2, or more formally, $\mathrm{cmax}(z_1, z_2)(v) = \max(z_1(v), z_2(v))$ for all $v \in B$. In this step, we split up the sum into two parts: one where $\mathrm{cmax}(z_1, z_2)$ is equal to x, and another where it is not. A note to make is that the requirement of $\mathrm{cmax}(z_1, z_2) \le y$ is equivalent to the requirement that $z_1, z_2 \le y$. We rewrote the requirement in this manner so that it would be easier to apply Lemma 6. Since the sum is now split into two parts corresponding to the two cases in Lemma 6, the following holds:

$$= \sum_{\substack{z_1,z_2 \in \mathbb{Z}_s^B \\ z_1,z_2 \leq x \\ \mathrm{cmax}(z_1,z_2)=x}} A_l(z_1) \cdot A_r(z_2) \cdot 1 + \sum_{\substack{z_1,z_2 \in \mathbb{Z}_s^B \\ z_1,z_2 \leq x \\ \mathrm{cmax}(z_1,z_2) \leq x \\ \mathrm{cmax}(z_1,z_2) \neq x}} A_l(z_1) \cdot A_r(z_2) \cdot 0$$

$$= \sum_{\substack{z_1,z_2 \in \mathbb{Z}_s^B \\ z_1,z_2 \leq x \\ \mathrm{cmax}(z_1,z_2)=x}} A_l(z_1) \cdot A_r(z_2)$$

Bringing everything together, we can see that we have proven that:

$$\overline{\tilde{A}l(x) \cdot \tilde{A}_r(x)} = \sum_{\substack{z_1,z_2 \in \mathbb{Z}_s^B \\ z_1,z_2 \leq x \\ \mathrm{cmax}(z_1,z_2)=x}} A_l(z_1) \cdot A_r(z_2)$$

Meaning that our state changes end up giving a result that is equal to sum of all products between pairs of two colorings for which the component-wise maximum is equal to x. This is exactly the behaviour that the Möbius-Transform-based join is supposed to give.

In this algorithm, we perform two state changes that both have time complexity $O(ks^k)$, and in between the two state changes we perform a component-wise multiplication of time complexity $O(s^k)$. These parts combine to give us an algorithm with time complexity $O^*(s^k)$. □

7 Filtering

The next step that we are going to cover is a result that has some wide application, namely filtering. Filtering is a technique that can be used in order to compute a join operation Q by using a different join operation P while only doing polynomially extra work. This is done by excluding results from the table obtained after performing join operation P in order to get the table that performing join operation Q would have given. In addition, it is not necessary to know how P is computed in order to use this technique.

In this section, we will first explain the filtering algorithm. Then we will look more closely at determining when a join operation Q can in fact be performed using join operation P with the filtering algorithm. Lastly, we will show an example of this technique being put to use and prove a result relating to an extension of Disjoin Set Sum.

7.1 The filtering algorithm

Before we start on the algorithm, we will establish some additional definitions. If a join operation P uses s states, then we use the cyclic group $\mathbb{Z}_s$ to represent the state set S used by P.

	0	1	2
0	0	1	2
1	1	1	2
2	2	2	2

	0	1	2
0	0		2
1		1	
2	2		2

Figure 9: *The join table on the left represents a full join operation. The join table on the right represents a suboperation of the operation represented by the join table on the left.*

We say that a join operation P is full if $P(j) \neq \times$ for all $j \in \mathcal{J}_S$. We say that a join operation Q is a suboperation of a join operation P, denoted by $Q \subseteq P$, if for all $j \in \mathcal{J}_S$, either $Q(j) = P(j)$ or $Q(j) = \times$. These two definitions are visualized in Figure 9.

We define the terms τ_m and υ_m for $m \in \mathbb{Z}_S$ as the number of occurrences of m in a coloring, where τ_m is used for colorings from the input tables of P and υ_m is used for colorings from the table outputted by P. A filter rule is a linear function $f(\tau_0, \tau_1, \ldots, \tau_{s-1}, \upsilon_0, \upsilon_1, \ldots, \upsilon_{s-1})$. We also define the vector space $F_s \subseteq \mathbb{R}^{2s}$ spanned by unit vectors with the values $\tau_0, \tau_1, \ldots, \tau_{s-1}$, $\upsilon_0, \upsilon_1, \ldots, \upsilon_{s-1}$ to be the filter rule vector space that contains all possible filter rules. We can link each vector in F_s to a filter rule f by simply summing all the coordinates of the vector.

We start the algorithm off with two tables $A_l, A_r : \mathbb{Z}_s^B \to \mathbb{Z}$ to join. The join operations that we will use are P and Q, with $Q \subseteq P$. We want to join A_l and A_r using P, and then use a filter rule $f \in F_s$ to filter our resulting table in order to get a table equivalent to that of joining A_l and A_r using Q.

The first step of our algorithm is to augment A_l and A_r a little bit in order to be able to successfully apply the filter rule later on. We define $\tau_f(\tau_0, \tau_1, \ldots, \tau_{s-1}) = f(\tau_0, \tau_1, \ldots, \tau_{s-1}, 0, \ldots, 0)$ and $\upsilon_f(\upsilon_0, \upsilon_1, \ldots, \upsilon_{s-1}) = f(0, \ldots, 0, \upsilon_0, \upsilon_1, \ldots, \upsilon_{s-1})$. For a given table A_d with $d \in \{l, r\}$, we define:

$$A'_d(c, \tau) = \begin{cases} A_d(c) & \text{if } \tau = \tau_f(\#_0(c), \#_1(c), \ldots, \#_{s-1}(c)) \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

This allows us to retain information about the colorings we are combining during the join operation. The values for τ_m for $m \in \mathbb{Z}_s$ range from 0 to k, which means that τ can take on at worst k^{s-1} different values. This is because of the extra constraint that $\tau_0 + \tau_1 + \cdots + \tau_{s-1} = k$, which has $s - 1$ degrees of freedom. We can therefore essentially view a table augmented in this manner as at most k^{s-1} copies of the original table. For $d = \{l, r, t\}$, we denote by $A'_d(*, \tau)$ the table A'_d which has a fixed value of τ.

The table A'_d is indexed by colorings c, of which there are s^k, and by all possible values for τ. Therefore, this step has a time complexity of $O(k^s s^k)$. In practice however, the amount of possible values τ can take on turns out to be significantly lower, which decreases the time complexity.

We then join the augmented tables A'_l and A'_r together using P in order to form the augmented table A'_t. We index the table A'_t by υ which can take on all the possible values that υ_f can take on. The next step is then:

$$A'_t(*, \upsilon) = \sum_{\tau_l + \tau_r = \upsilon} P(A'_l(*, \tau_l), A'_r(*, \tau_r)) \tag{6}$$

For this step, we have to iterate over all possible values for υ, of which there are at most k^{s-1}. For each of these steps, we calculate a sum with k^{s-1} summands. This number comes from the fact that $\tau_l + \tau_r = \upsilon$ has one degree of freedom in τ_l or τ_r, both of which can take on k^{s-1} different values. This results in $k^{2(s-1)}$ times that we perform the join operation P, the complexity of which we denote by $P(k)$. This results in a time complexity of $O(k^{2(s-1)}P(k))$.

In order to now obtain the joined table that we want, we have to select entries from A'_t in the following manner:

$$A_t(c) = A'_t(c, \upsilon) \text{ where } \upsilon = -\upsilon_f(\#_0(c), \#_1(c), \ldots, \#_{s-1}(c)) \tag{7}$$

We want to keep results precisely when $f = 0$, or equivalently when $\tau_f = -\upsilon_f$. When two colorings are joined such that $f = 0$, then we know that the state combinations we want to deny did not occur in joining these two colorings. The reasoning behind this will become clear in Section 7.2. This is why we need to augment our tables with τ prior to performing our join operation P. It allows us to retain the information from before the join that we need in order to apply our filter rule.

This step involves going through the full table A_t, which has size s^k, and performing a calculation that uses at most s steps, meaning that s^{k+1} steps may be required. This results in a complexity of $O(s^k)$ for this step.

After this step, the table A_t is exactly the table that we are supposed to get after joining tables A_l and A_r together with join operation Q. Between all of the steps in this algorithm, the most costly one was the joining of the two augmented tables, meaning that the time complexity of this algorithm is $O(k^{2(s-1)}P(k))$.

7.2 Filterable sets of state combinations and their filter rules

While the algorithm above is quite nice, we have yet to prove that it actually works. In this section we will not only show that the above algorithm works, but we will also show how to determine whether a set of state combinations can be filtered from a join operation, and how to determine a filter rule that will allow this.

We shall start with an $s \times s$ join operation P and a set of state combinations $J \subseteq \mathcal{J}_S$. We can construct a new join operation P_J by setting $P_J(j) = P(j)$ if $j \notin J$ and $P_J(j) = \times$ if $j \in J$. It is clear that $P_J \subseteq P$. We say that J can be filtered from P (or that J is filterable in P) if there exists a filter rule $f \in F_s$ such that if performing the algorithm in Section 7.1 with P and f results in a table equivalent to joining the two input tables with P_J. We want to have a way to determine whether J can be filtered from P, and if so, what its filter rule would be.

Since filter rules are made up of the terms τ_m and υ_m with $m \in \mathbb{Z}_s$, it would make sense to start off by trying to work with these terms. Define l_j as the amount of times the state combination $j \in \mathcal{J}_s$ is performed when executing a given join operation. Next, we will try to write filter rule terms τ_m and υ_m for all $m \in \mathbb{Z}_s$ in terms of l_j.

The terms τ_m for $m \in \mathbb{Z}_s$ are terms that represent the amount of m-states in a coloring before the join operation is performed. With this, we can form the following equation:

$$\tau_m = 2l_{j_{m,m}} + \sum_{\substack{a \in \mathbb{Z}_s \\ a < m}} l_{j_{a,m}} + \sum_{\substack{a \in \mathbb{Z}_s \\ a > m}} l_{j_{m,a}}$$

Recall that $j_{p,q}$ for $p, q \in S$ refers to the state combination that combines p and q. Every time that the state combination $j_{m,m}$ is performed, the original two colorings had $l_{j_{m,m}}$ different vertices each that had the state m, resulting in at least $2l_{j_{m,m}}$ m-states in the colorings before the join operation. A similar argument for the joins in which only one side has the state m gives us the above formula. Important to note is that this equation doesn't change with different join operations.

The terms υ_m for $m \in \mathbb{Z}_n$ are instead terms that represent the amount of m states in a coloring after the join operation has been performed. The equation for this is quite straightforward as well; if $P(j) = m$, then that means that when joining two colorings together, the amount of times the state combination j has to be performed is the amount that gets contributed to υ_m. In other words:

$$\upsilon_m = \sum_{j \in J} [P(j) = m] \cdot l_j$$

We re-use the same square bracket notation as from Sections 5 and 6. These equations do change with different join operations, since these equations explicitely use the join operation P.

The introduction of the terms l_j and rewriting the filter rule terms in these equations is not without reason. Since we want to filter the state combinations in J from our resulting join table, what we essentially want is for $l_j = 0$ for all $j \in J$. In writing our filter rule terms as equations in terms of l_j, we can tackle them as a system of linear equations which allows us to see what we can do in order to get the required l_j terms set to 0.

As with any system of linear equations, we can write the system in the form of a matrix, which we will call the filtering matrix for join operation P and denote by M_P. The filtering matrix M_P has two parts: a non-augmented part denoted by M'_P containing the columns corresponding to the l_j terms, and an augmented part denoted by M''_P containing columns corresponding to the filter rule terms $\tau_0, \tau_1, \ldots, \tau_{s-1}, \upsilon_0, \upsilon_1, \ldots, \upsilon_{s-1}$. Initially, M''_P is simply I_{2s}, the $2s \times 2s$ identity matrix. The filtering matrix M_P for a general join operation P is shown in Figure 10. We note that we don't write M''_P in matrix form in order to save space.

The dimensions of M_P can be determined quite easily. The filtering matrix M_P has precisely one row for each of the terms τ_m and υ_m with $m \in \mathbb{Z}_s$, of which there are $2s$. Each column in M'_P corresponds to one of the l_j terms in our original equations, meaning the amount of

$$
\left(\begin{array}{cccc|c}
2 & 1 & \cdots & 0 & \tau_0 \\
0 & 1 & \cdots & 0 & \tau_1 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \cdots & 2 & \tau_{s-1} \\
[P(j_{0,0})=0] & [P(j_{0,1})=0] & \cdots & [P(j_{s-1,s-1})=0] & \upsilon_0 \\
[P(j_{0,0})=1] & [P(j_{0,1})=1] & \cdots & [P(j_{s-1,s-1})=1] & \upsilon_1 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
[P(j_{0,0})=s-1] & [P(j_{0,1})=s-1] & \cdots & [P(j_{s-1,s-1})=s-1] & \upsilon_{s-1}
\end{array}\right)
$$

Figure 10: *Filtering matrix M_P for a general join operation P. M'_P is on the left side, and M''_P is on the right side. Note that we do not write M''_P in matrix form in order to save space.*

$$
\left(\begin{array}{cccccc|c}
2 & 1 & 1 & 0 & 0 & 0 & \tau_0 \\
0 & 1 & 0 & 2 & 1 & 0 & \tau_1 \\
0 & 0 & 1 & 0 & 1 & 2 & \tau_2 \\
1 & 0 & 0 & 0 & 1 & 0 & \upsilon_0 \\
0 & 1 & 0 & 0 & 0 & 1 & \upsilon_1 \\
0 & 0 & 1 & 1 & 0 & 0 & \upsilon_2
\end{array}\right)
$$

Figure 11: *Filtering matrix M_{FT_3} for the 3×3 Fourier-Transform-based join operation. Intuitively, the first s coordinates of a column vector in M'_{FT_3} correspond to the state combination $j \in \mathcal{J}_{\mathbb{Z}_3}$ that column vector refers to, and the last n coordinates correspond to the value of $FT_3(j)$. If we look at the third column vector from the left for example, we see that the coordinates corresponding to τ_0 and τ_2 both have a value 1, meaning that this column corresponds to the state combination $j_{0,2}$. This column vector also has a value of 1 on the coordinate corresponding to υ_2, meaning that $FT_3(j_{0,2}) = 2$, which is indeed true.*

columns in M'_P is equal to $|\mathcal{J}_S| = \frac{1}{2}s(s+1)$. This also implies that the amount of columns in M'_P is equal to or larger than the amount of rows when $s \geq 3$. Since M''_P is I_{2s}, it has $2s$ columns and therefore the dimensions $2s \times 2s$.

Since the filtering matrix M_P shown in Figure 10 is perhaps a little overwhelming, we also show the filtering matrix M_{FT_3} for the 3×3 Fourier-Transform-based join operation FT_3 in Figure 11.

We can make a couple of observations on our filtering matrix.

Lemma 7.1. *For a full join operation P, M'_P is not a full-rank matrix.*

The sum of the last s row vectors of M'_P is equal to a vector in which the coordinate corresponding to the term l_j takes on the value:

$$\sum_{m \in \mathbb{Z}_s} [P(j) = m]$$

The value of $[P(j) = m]$ can only be 1 or 0. Since P is a full join operation, we know that for all $j \in \mathcal{J}_S$, $P(j) \neq \times$ and thus $P(j) = m$ for some $m \in \mathbb{Z}_s$. Since only one term in the sum is equal to $[P(j) = m]$ for this m, we know that the value of this coordinate is equal to 1. Since the same reasoning holds for all coordinates, we know that the sum is equal to the **1** vector.

Similarly, the sum of the first s row vectors is also equal to a vector where each coordinate has the same value. If the coordinate corresponds to a term of the form $l_{j_{a,a}}$, then the equation with τ_a will contribute a value of 2 to the sum, and all the others will contribute 0. If the coordinate corresponds to a term of the form $l_{j_{a,b}}$ with $a \neq b$, then the equations for τ_a and τ_b will both contribute a value of 1 to the sum, and all the others 0. Thus, we see that the sum of the first s row vectors is equal to the **2** vector. The **1** and **2** vectors are linearly dependent, so we know that the rank of M'_P is not full. □

Lemma 7.2. *For a join operation P, M''_P is a full-rank matrix.*

M''_P is the identity matrix I_{2s}, so this is trivially true. □

Now that we have seen how a filtering matrix looks like and determined a couple of lemmas on filtering matrices, we are ready to move onto the next step, which is to put the non-augmented part of the filtering matrix, M'_P, into row-reduced echelon form. During this process, we also make sure to adjust the augmented park of the filtering matrix, M''_P, as necessary. While it isn't a necessity to put the filtering matrix in row-reduced echelon form, it does make the following steps easier to carry out.

What we end up with is a collection of rows in our matrix that on one side represent formulas with l_j for and on the other side contains formulas made up of the terms $\tau_0, \tau_1, \ldots, \tau_{s-1}$, $\upsilon_0, \upsilon_1, \ldots, \upsilon_{s-1}$. The terms l_j for $j \in \mathcal{J}_S$ are terms we cannot influence or determine during the course of the join operation. While we also can't influence of the variables τ_m and υ_m for $m \in \mathbb{Z}_s$, we can at least determine the values of these variables after the algorithm has completed and use those values in order to possibly determine values for l_j for $j \in \mathcal{J}_S$. Of course, for our purposes, we would like to be able to determine when $l_j = 0$ for $j \in J$., as that would mean we would have a way of using the variables τ_m and υ_m for $m \in \mathbb{Z}_s$ in order to find out which results don't use the joins that we want to filter.

So take a row from our filtering matrix. This doesn't have to specifically be a row from the row-reduced echelon form of the filtering matrix; any linear combination of rows from M_P will do. Since our matrix is based on a system of linear equations, we know this row will also represent an equation. This equation has the following form:

$$c_{j_1} l_{j_1} + c_{j_2} l_{j_2} + \cdots + c_{j_{|\mathcal{J}_s|}} l_{j_{|\mathcal{J}_s|}} = c'_{\tau_0}\tau_0 + \cdots + c'_{\tau_{s-1}}\tau_{s-1} + c'_{\upsilon_0}\upsilon_0 + \cdots + c'_{\upsilon_{s-1}}\upsilon_{s-1} \tag{8}$$

where the coefficients c and c' are defined as the coordinates of the row vector we took from M_P. The state combinations $j_i \in \mathcal{J}_s$ with $1 \leq i \leq |\mathcal{J}_s|$ are simply the state combinations in $\mathcal{J}_s$ in some arbitrary order.

This now leads into our main theorem of this paper.

Theorem 7.3. *Given a full join operation A and a set $J \subseteq \mathcal{J}_s$. The set J can be filtered from P if and only if we can make a row vector in M'_P using basic row operations such that $c_j > 0$ if $j \in J$ and $c_j = 0$ if $j \notin J$.*

Assume we can make such a row vector in the filtering matrix M_P. Then we can set up an equation in the form of equation 8. We can take the right-hand side of this equation to be our filter rule $f(\tau_0, \tau_1, \ldots, \tau_{s-1}, \upsilon_0, \upsilon_1, \ldots, \upsilon_{s-1})$. When we set the right-hand side equal to 0, as we do to the filter rule f in the algorithm, then we find that:

$$c_{j'_1} l_{j'_1} + \cdots + c_{j'_{|J|}} l_{j'_{|J|}} = 0$$

where $j'_i \in J$ and $c_{j'_i} > 0$ for all $1 \leq i \leq |J|$. Since all the $c_{j'}$ terms are positive and since the $l_{j'}$ terms cannot be negative, the only way the above equation can hold true is if $l_{j'} = 0$ for all $j' \in J$. This means that our filter rule f does indeed filter out all state combinations in J, and thus J can be filtered from P.

Assume that we cannot make a row vector as described in the theorem. Then we will have to show that we cannot always determine that $l_{j'} = 0$ for all $j' \in J$. In the event that $c_{j'} = 0$ for $j' \in J$, then $l_{j'}$ can take on any value in $\mathbb{N}$. If $c_j \neq 0$ for $j \notin J$, then make $c_j = 0$ via basic row operations. If this is not possible for all $j \notin J$, then we will have extra degrees of freedom in l_j, for all $j \notin \mathcal{J}_S \backslash J$ with $c_j \neq 0$, so we cannot determine that $l_{j'} = 0$ for all $j' \in J$. Therefore, we can assume $c_{j'}$ are non-zero if and only if $j' \in J$.

In that case, we know some of our non-zero c coefficients are positive, and the other non-zero c coefficients are negative. We denote the positive c coefficients by c_{+_s} and the negative c coefficients c_{-_t}, where s and t range between 1 and s_{max} and t_{max}, the total number of positive and negative c coefficients respectively. Call the sum of the positive c coefficients C_+, and the sum of the negative c coefficients C_-. We can then put all terms l_j with a positive c coefficient equal to $-C_-$ and all terms l_j with a negative c coefficient equal to C_+. These are both positive values. This then gives us:

$$\begin{aligned}
& c_{+_1} l_{+_1} + \ldots + c_{+_{s_{max}}} l_{+_{s_{max}}} + c_{-_1} l_{-_1} + \cdots + c_{-_{t_{max}}} l_{-_{t_{max}}} \\
=& - c_{+_1} C_- - \ldots - c_{+_{s_{max}}} C_- + c_{-_1} C_+ + \cdots + c_{-_{t_{max}}} C_+ \\
=& - C_-(c_{+_1} + \cdots + c_{+_{s_{max}}}) + C_+(c_{-_1} + \cdots + c_{-_{t_{max}}}) \\
=& - C_- \cdot C_+ + C_+ \cdot C_- \\
=& 0
\end{aligned}$$

In short, our equation has a degree of freedom left. This means that if we were to apply the filter rule in this case, we cannot actually be certain that all the state combinations that we wish to filter appear 0 times in our join. Since this reasoning applies to all possible equations we can make for J, we cannot filter J from P in this case. □

Having proven the main filtering theorem, we end this section with some corollaries.

Corollary 7.3.1. *For a full $s \times s$ join operation P and filterable sets of state combinations $J_1, J_2 \subseteq \mathcal{J}_S$ and filter rules f_1 and f_2 that filter J_1 and J_2 from P respectively. If $f_1 = f_2$, then $J_1 = J_2$.*

We show that if $J_1 \neq J_2$, then $f_1 \neq f_2$. Since J_1 and J_2 are both filterable, we can construct two row vectors complying to the requirements in Theorem 7.3 using basic row operations to

	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

	0	1	2
0	0	1	2
1	1	2	
2	2		

Figure 12: *3×3 join tables for join operations FT_3 and FT_3^*.*

the filtering matrix M_P. These same basic row operations give us the filter rules f_1 and f_2, which can be obtained from M''_P. Assume that $f_1 = f_2$. Then the row vectors in M''_P are clearly linearly dependent, meaning the rank of M''_P cannot be full. However, we know from Lemma 7.2 that the rank of M''_P is full, leading to a contradiction. □

Corollary 7.3.2. *For a full $s \times s$ join operation P, if $J_1, J_2 \subseteq \mathcal{J}_S$ are filterable, then $J_1 \cup J_2$ is also filterable*

We know from Theorem 7.3 that if J_1 and J_2 are filterable, we can construct two row vectors r_1 and r_2 from the rows in M_P using basic row operations such that the equations that come forth from r_1 and r_2 have $c_j > 0$ if $j \in J_1$ or $j \in J_2$, and $c_j = 0$ otherwise. Then the equation corresponding to the row vector $r_1 + r_2$ has $c_j > 0$ if $j \in J_1 \cup J_2$ and $c_j = 0$ otherwise, which by Theorem 7.3 makes $J_1 \cup J_2$ filterable. □

Corollary 7.3.3. *For any full $s \times s$ join operation P with $s \geq 3$, there will always be a $J \subseteq \mathcal{J}_S$ that cannot be filtered from P.*

Assume that we can filter all $J \subseteq \mathcal{J}_S$ from P. Then we can also filter all singleton sets of state combinations, i.e. $J \subseteq \mathcal{J}_n$ with $|J|= 1$. From Theorem 7.3, we know that, for every $j \in \mathcal{J}_S$, we must be able to use basic row operations on the filtering matrix M_P in order to get a row vector such that $c_j > 0$, and all the others coordinates are equal to 0. This gives us $|\mathcal{J}_S|= \frac{1}{2}s(s+1)$ row vectors which are all linearly independent. This implies that $\text{rank}(M'_P) \geq \frac{1}{2}s(s+1)$. We also know from Lemma 7.1 that $\text{rank}(M'_P) < 2s$. This implies that $\frac{1}{2}n(n+1) < 2s$, which is only true when $s < 3$. □

7.3 An example with Fourier-Transform-based join

In the final part of this section, we will give an example of how to calculate a join operation with this method. The join operation we will be trying to compute is the 3×3 Fourier-Transform-based join operation FT_3 with $J = \{j_{1,2}, j_{2,2}\}$ our set of state combinations to filter. This new join operation is denoted by FT_3^*. The join tables for FT_3 and FT_3^* are shown in Figure 12.

The reason why we chose specifically this join operation as an example is that this join operation already has a fairly clear filter rule that we could use. We can see that $FT_3^*(j_{p,q}) = \max(p+q, 2)$, and thus a filter rule we can use is:

$$g(\tau_0, \tau_1, \tau_2, \upsilon_0, \upsilon_1, \upsilon_2) = 0 \cdot \tau_0 + 1 \cdot \tau_1 + 2 \cdot \tau_2 - 0 \cdot 2 \cdot \upsilon_0 - 1 \cdot 2 \cdot \upsilon_1 - 2 \cdot 2 \cdot \upsilon_2$$

$$
\left(\begin{array}{cccccc|c}
1 & 0 & 0 & 0 & 0 & -1 & \frac{1}{3}\tau_0 - \frac{1}{3}\tau_2 + \frac{1}{3}\upsilon_0 - \frac{1}{3}\upsilon_1 \\
0 & 1 & 0 & 0 & 0 & 1 & \upsilon_1 \\
0 & 0 & 1 & 0 & 0 & 1 & \frac{1}{3}\tau_0 + \frac{2}{3}\tau_2 - \frac{2}{3}\upsilon_0 - \frac{1}{3}\upsilon_1 \\
0 & 0 & 0 & 1 & 0 & -1 & \frac{1}{6}\tau_0 + \frac{1}{2}\tau_1 - \frac{1}{6}\tau_2 - \frac{1}{3}\upsilon_0 - \frac{2}{3}\upsilon_1 \\
0 & 0 & 0 & 0 & 1 & 1 & -\frac{1}{3}\tau_0 + \frac{1}{3}\tau_2 + \frac{2}{3}\upsilon_0 + \frac{1}{3}\upsilon_1 \\
0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2}\tau_0 - \frac{1}{2}\tau_1 - \frac{1}{2}\tau_2 + \upsilon_0 + \upsilon_1 + \upsilon_2
\end{array}\right)
$$

Figure 13: *Filtering matrix M_{FT} in reduced row-echelon form*

We make gratuitous use of coefficients here in order to more clearly convey what is going on. First off, the sum $\tau_0+\tau_1+\tau_2$ will always be twice as large as the sum $\upsilon_0+\upsilon_1+\upsilon_2$ due to it being based on two colorings compared to one. The sum of two states in a legal state combination in FT_3^* is also its resulting state. This behaviour can be captured by multiplying each term τ_m by m and summing these together, and setting this equal to the sum created by summing the terms υ_m multiplied by $2m$, all for $m \in \mathbb{Z}_s$. This then results in the above filter rule.

Despite already having a filter rule, we will still go through the steps outlined in the previous section in order to find a filter rule for FT_3^*. We set up the filtering matrix M_{FT_3}, which was shown in Figure 11. Then we put M_{FT_3} in reduced row-echelon form, the result of which is shown in Figure 13.

Our goal now is to be able to apply Theorem 7.3, since this theorem directly allows us to determine whether or not we can filter J from FT_3. According to this theorem, we can filter J from FT_3 if and only if we can make a row vector in M'_{FT_3} using basic row operations such that $c_{j_{1,2}}, c_{j_{2,2}} > 0$, and $c_j = 0$ for $j \in \mathcal{J} - J$.

Thankfully, this is the case, and quite clearly so. The 5th row vector in M'_{FT_3}, which is equal to $(0,0,0,0,1,1)$, is already in this form. This means we can extract our filter rule f quite easily from the 5th row of M''_{FT}. Using Theorem 7.3, we have determined that it is possible to filter J from FT, by using the filter rule $f(\tau_0,\tau_1,\tau_2,\upsilon_0,\upsilon_1,\upsilon_2) = -\frac{1}{3}\tau_0 + \frac{1}{3}\tau_2 + \frac{2}{3}\upsilon_0 + \frac{1}{3}\upsilon_1$.

Interestingly though, the filter rule f is quite different from the filter rule g we had established earlier. However, we can see with a little bit of effort that these two end up resolving to be the same. First, we can multiply f by a constant 3, since we are setting f equal to 0 in our algorithm anyway. We can also use the identity $\tau_0+\tau_1+\tau_2-2\upsilon_0-2\upsilon_1-2\upsilon_2$ and add this onto f (which also appears in the 6th row vector in M''_{FT_3}). This gives us:

$$
\begin{aligned}
&3(-\frac{1}{3}\tau_0 + \frac{1}{3}\tau_2 + \frac{2}{3}\upsilon_0 + \frac{1}{3}\upsilon_1) + \tau_0 + \tau_1 + \tau_2 - 2\upsilon_0 - 2\upsilon_1 - 2\upsilon_2 \\
=&-\tau_0 + \tau_2 + 2\upsilon_0 + \upsilon_1 + \tau_0 + \tau_1 + \tau_2 - 2\upsilon_0 - 2\upsilon_1 - 2\upsilon_2 \\
=&\tau_1 + 2\tau_2 - 2\upsilon_1 - 4\upsilon_2 \\
=&g(\tau_0,\tau_1,\tau_2,\upsilon_0,\upsilon_1,\upsilon_2)
\end{aligned}
$$

and so we can see that the filter rules f and g give the same result.

We will now look at how to calculate FT_3^* with the algorithm in Section 7.1. We can choose to use either f or g, or any other filter rule that is a linear combination of these. We will be using $f' = 3f$. The time complexity of the algorithm is $O(k^{2(s-1)}P(k))$. In this case, the

join operation we are using in our algorithm is FT_3, which has been proven to have a time complexity of $O(k3^k)$. The value of s here is 3, so if we strictly apply the algorithm in Section 7.1, then we get a time complexity of $O(k^5 3^k)$.

We can improve this however. In the algorithm in Section 7.1, we augment our tables with a term τ, which takes on all possible values that $\tau_{f'}$ can take on. Worst-case, this means τ takes on $k^{2(s-1)}$ different values. However, for the filter rule we are using for this join operation, this number is quite a bit lower. We can see from $\tau_{f'}(\tau_0, \tau_1, \tau_2) = -\tau_0 + \tau_2$ that the values that $\tau_{f'}$, and therefore τ, can take on range from $-k$ to k. Similarly, the values $\upsilon_{f'}$, and therefore υ, can take on range from 0 to $2k$ since $\upsilon_{f'}(\upsilon_0, \upsilon_1, \upsilon_2) = 2\upsilon_0 + \upsilon_1$. Since the values we augment our tables with only have complexity $O(k)$, rather than $O(k^2)$, we can reduce the time complexity of our algorithm to $O(k^3 3^k)$.

Another reason why we choose this join operation can solve an extension of the DISJOINT SET SUM problem. This problem asks for k-tuples $\mathcal{A}, \mathcal{B} \subseteq \{0, 1, \ldots, s-1\}^k$, what the size of the set $\mathcal{A} \bigoplus \mathcal{B}$ is, where $\mathcal{A} \bigoplus \mathcal{B} = \{(A_1 + B_1, A_2 + B_2, \ldots, A_k + B_k) | A_i + B_i \leq s - 1 \text{ for all } 1 \leq i \leq k\}$, with A_i and B_i the i-th element of the k-tuple $\mathcal{A}$ and $\mathcal{B}$ respectively. We can solve this with the join operation FT_s^* by using it with two tables A_l and A_r, where $A_l(c) = A_r(c) = 1$ for all $c \in \mathbb{Z}_s^B$.

An algorithm to solve this extension in $O(k^3 s^k)$ was given by Cygan and Pilipczuk [12], which we have just matches. We will show that this problem can be solved in $O(k^2 s^k)$ for constant s in the theorem below.

Theorem 7.4. *The join operation FT_s^* can be computed in $O(k^2 s^{k+2})$, which is $O(k^2 s^k)$ when s is constant.*

This algorithm follows a structure similar to that of the algorithm in Section 7.1. We start with tables $A_l(c)$ and $A_r(c)$. We can extend the filter rule g from the previous explanation in order to fit with the general case join operation FT_s^*:

$$g(\tau_0, \tau_1, \ldots, \tau_{s-1}, \upsilon_0, \upsilon_1, \ldots, \upsilon_{s-1}) = \tau_1 + 2 \cdot \tau_2 + \cdots + (s-1) \cdot \tau_{s-1} + \upsilon_1 + 2 \cdot \upsilon_2 + \cdots + (s-1) \cdot \upsilon_{s-1}$$

For this filter rule g, the values that τ_g and υ_g can both take on values ranging from 0 to sk. We augment our tables $A_l(c)$ and $A_r(c)$ to $A'_l(c, \tau)$ and $A'_r(c, \tau)$ respectively just like in equation 5, which in this case has a time complexity of $O(sks^k) = O(ks^{k+1})$.

However, the step in equation 6 is performed differently. Instead of summing over possible combinations of τ_l and τ_r and performing the join operation FT_s^* for each of theses combinations, we instead start by performing FT_s^* on the full tables $A'_l(c, \tau)$ and $A'_r(c, \tau)$. In particular, we perform the state change defined in Section 5, equation 1 on our tables $A'_l(c, \tau)$ and $A'_r(c, \tau)$. These tables have ks^{k+1} entries, and the algorithm for state changing them iterates over the entire table k times, so this step has time complexity $O(k^2 s^{k+1})$.

Now we use the sum shown in equation 6 to join the state changed tables $A'_l(c, \tau)$ and $A'_r(c, \tau)$ together. This works as follows:

$$A'_t(*, \upsilon) = \sum_{\tau_l + \tau_r = \upsilon} A'_l(*, \tau_l) \cdot A'_r(*, \tau_r)$$

We iterate over sk possible values of υ and sk possible ways that $\tau_l + \tau_r$ can be equal to υ. Each of these iterations also performs an entry-wise multiplication on a table of size s^k (since τ_l and τ_r are fixed), just like in Theorem 5.3. This means that this step has a time complexity of $O(k^2 s^{k+2})$.

We state change our tables A'_l and A'_r back to A_l and A_r, which has a time complexity $O(ks^{k+1})$. Lastly, we perform the filtering in the same way as in equation 7, which has time complexity. $O(s^k)$. The total time complexity of our algorithm is therefore $O(k^2 s^{k+2})$, which is $O(k^2 s^k)$ when s is constant. □

8 Further research opportunities and conclusion

We end this thesis by discussing further research opportunities and concluding the thesis.

In Theorem 4.1, we gave an overview of the time complexities for the 2×2 join operations and proved that each 2×2 join operation has a time complexity of $O^*(2^k)$. There are further research opportunities into other sizes of join operations, with the 3×3 join operations likely to be the next step. Using the equivalence relation explained in Section 4, the amount of 3×3 join operations can be determined to be at most 160. The filtering technique can be used to great effect here as well, since given a join operation P with a certain complexity, then suboperations $Q \subseteq P$ that can be obtained by filtering P will have the same exponential complexity.

The filtering technique itself is also subject to further research. While the filtering technique is already a very useful technique when the join operation P used to compute $Q \subseteq P$ is not known, it has the potential to be even more useful once we can make certain assumptions about P and change the way that P is performed. We saw in Theorem 7.4 that as soon as we were able to change the order of the steps that we normally take in the filtering algorithm, that we were able to decrease the complexity of the algorithm by a factor k. Therefore, research into other general-form filtering algorithms that make more assumptions on how P is calculated might prove worthwhile.

There are further research opportunities for calculating components in join operations. We define a component in a join operation P with state set S as a subset $S' \subset S$ such that $j_{s',s} = \times$ for all $s' \in S'$ and $s \in S \backslash S'$. In Figure 14, a join operation using state set $S = \mathbb{Z}_6$ with components $\{0, 1, 2\}$ and $\{3, 4, 5\}$ is shown.

A useful result for join operations of this form would be for the components to be able to be calculated as separate join operation. This would then allow us to use other algorithms for computing the different components of a join operation. The join operation shown in Figure 14 could use the algorithm explained in Theorem 7.4 for both of its components, for example. This would lead to improvements in algorithms for [ρ-σ]-domination problems, which were introduced by Telle [7]. In particular, the join operation in Figure 14 is the same as the join

	0	1	2	3	4	5
0	0	1	2			
1	1	2				
2	2					
3				3	4	5
4				4	5	
5				5		

Figure 14: *Join table for a join operation using state set* $S = \mathbb{Z}_6$*. This join operation has components* $\{0, 1, 2\}$ *and* $\{3, 4, 5\}$*.*

operation for a $[\rho\text{-}\sigma]$-domination problem with $\rho = \sigma = \{0, 1, 2\}$. A decomposing theorem could also be useful when attempting to summarize the time complexities of all $s \times s$ join operations for $s > 2$.

Finally, we answer the research questions that we posed at the beginning of the thesis period.

'What properties must a join matrix have such that specific methods can be applied in the process of computing a join operation with this join matrix? What complexity can be achieved for join matrices of a specific form?' (1)

This was a fairly open-ended research question, and thus this thesis does not give a complete answer. However, our filtering theorem does provide a partial answer. In this theorem, we have shown and worked out a method for computing join matrices and shown under which circumstances this method can be applied.

'Can a classification of the 2×2 *join matrices be made?'* (2)

This research question has been fully answered. Yes, we can make a classification of the 2×2 join matrices, and we have shown that all these join matrices can be computed in $O^*(2^k)$.

'How do generalizations and join techniques scale to larger state sets?' (3)

While this question may not have been answered in any general way, we have always taken scaling into account in the techniques that we have developed, including the filtering technique

and the computation of the Fourier- and Möbius-Transform-based join operations. In all these cases, scaling ended up working fairly straightforwardly.

'Can we construct a generic theorem for join operations?' (4)

This question cannot be answered as of now, but we have definitely made certain advancements in this question as a result of this thesis. Filtering in particular allows for us to gain a clearer picture of which join operations can be computed with which time complexities. Our result on 2×2 join operations in Theorem 4.1 also contibutes to a generic theorem on join operations.

References

[1] Neil Robertson, Paul D. Seymour, *Graph minors. II. Algorithmic aspects of tree-width* in: Journal of Algorithms, Volume 7, Issue 3, Elsevier, 1994, pages 309–322.

[2] Stefan Arnborg, Derek G. Corneil, Andrzej Proskurowski, *Complexity of finding embeddings in a k-tree* in: Society for Industrial and Applied Mathematics Journal on Algebraic and Discrete Methods, Volume 8, Issue 2, SIAM, 1987, pages 277–284.

[3] Hans L. Bodlaender, *A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth* in: Society for Industrial and Applied Mathematics Journal on Computing, Volume 25, Issue 6, SIAM, 1996, pages 1305–1317.

[4] Hans L. Bodlaender, *A partial k-arboretum of graphs with bounded treewidth* in: Theoretical Computer Science, Volume 209, Issue 1–2, Elsevier, 1998, pages 1–45.

[5] Hans L. Bodlaender, Arie M.C.A. Koster, *Treewidth computations I. Upper bounds* in: Information and Computation, Volume 208, Issue 3, Elsevier, 2010, pages 259–275.

[6] Ton Kloks, *Treewidth, Computations and Approximations* in: Lecture Nores in Computer Science, Volume 842, Springer, 1994.

[7] Jan Arne Telle, Andrzej Proskurowski, *Algorithms for Vertex Partitioning Problems on Partial k-Trees* PhD Thesis, Department of Computer and Information Science, University of Oregon, Eugene, Oregon, USA, 1994.

[8] Jochen Alber, Rolf Niedermeier, *Improved Tree Decomposition Based Algorithms for Domination-like Problems* in: Lecture Notes in Computer Science, Volume 2286, Springer, 2002, pages 613–627.

[9] Johan M. M. van Rooij, *Exact Exponential-Time Algorithms for Domination Problems in Graphs* 2011.

[10] Johan M. M. van Rooij, Hans L. Bodlaender, Peter Rossmanith, *Dynamic Programming on Tree Decompositions Using Generalised Fast Subset Convolution* in: Lecture Notes in Computer Science, Springer, 2009, pages 566–577.

[11] Marek Cygan, Jesper Nederlof, Marcin Pilipczuk, Michał Pilipczuk, Johan van Rooij, Jakub Onufry Wojtaszczyk, *Solving connectivity problems parameterized by treewidth in single exponential time* 2011.

[12] Marek Cygan, Marcin Pilipczuk, *Exact and approximate bandwidth* in: Theoretical Computer Science, Volume 411, Issues 40–42, Elsevier, 2010, pages 3701–3713.