



Factored Mappings as Knowledge Compilation for Symbolic Search

Bachelor Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence Research Group
<https://ai.dmi.unibas.ch/>

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Silvan Sievers

Leonhard Badenberg
leonhard.badenberg@stud.unibas.ch
2016-055-238

June 7, 2021

Acknowledgments

I want to thank my supervisor Dr. Silvan Sievers for his time and guidance. His advice and feedback was always helpful and his patience greatly appreciated. I also want to thank Prof. Dr. Malte Helmert for giving me the opportunity to do my bachelor thesis in the artificial intelligence research group and providing me with such an interesting topic. Furthermore I want to thank my family and friends for giving me nothing but encouragement and helpful advice.

Abstract

Symbolic search is an important approach to classical planning. Symbolic search uses search algorithms that process sets of states at a time. For this we need states to be represented by a compact data structure called knowledge compilations. Merge-and-shrink representations come a different field of planning, where they have been used to derive heuristic functions for state-space search. More generally they represent functions that map variable assignments to a set of values, as such we can regard them as a data structure we will call Factored Mappings.

In this thesis, we will investigate Factored Mappings (FMs) as a knowledge compilation language with the hope of using them for symbolic search. We will analyse the necessary transformations and queries for FMs, by defining the needed operations and a canonical representation of FMs, and showing that they run in polynomial time. We will then show that it is possible to use Factored Mappings as a knowledge compilation for symbolic search by defining a symbolic search algorithm for a finite-domain planning task that works with FMs.

Table of Contents

Acknowledgments	ii
Abstract	iii
1 Introduction	1
2 Background	3
2.1 Definitions	4
3 Canonicity	6
3.1 Requirements	6
3.2 Reducing Factored Mappings	10
4 Operations on Factored Mappings	13
4.1 Building Basic Factored Mappings	13
4.1.1 False	13
4.1.2 True	14
4.1.3 Atom	14
4.2 Boolean Tests	15
4.2.1 Includes	15
4.2.2 Equals	16
4.3 Set Operations	16
4.3.1 Union	17
4.3.2 Intersection	21
4.3.3 Complement	23
4.3.4 Difference	23
5 Symbolic Search	24
5.1 Converting Formulas into Factored Mappings	25
5.1.1 Formula	25
5.1.2 Singleton	26
5.2 The Apply Function	26
5.2.1 Reordering and Renaming	27
5.3 Symbolic Search Algorithm for Factored Mappings	28

Table of Contents	v
6 Conclusion	29
Bibliography	30
Declaration on Scientific Integrity	31

1

Introduction

Classical planning is an important part of artificial intelligence. It deals with finding a satisfying or optimal solution for a given planning task through the use of planning algorithms. Planning tasks compactly represent transition systems and are based on concepts from propositional logic. There are propositional and finite-domain planning tasks. One of "the big three" (Helmert and Röger [4]) approaches for classical planning algorithms is symbolic search. The idea of symbolic search is to use a search algorithm that can process sets of states at a time. For this, state sets have to be represented by more compact data structures. We want these data structures to be able to represent exponentially large state sets in polynomial size efficiently, so we can use search algorithms such as a symbolic breadth-first search on this data structure. There are different compilation approaches to represent knowledge bases as a compact data structure, called knowledge compilations. Knowledge compilations hope to deal with intractability of problems like general propositional reasoning (Darwiche and Marquis 2002 [3]). Some knowledge compilation languages like

- Negation Normal Form (NNF)
- Binary Decision Diagram (BDD)
- Disjunctive Normal Form (DNF)
- Conjunctive Normal Form (CNF)

and many others are already well researched on their advantages, applications, and the transformations they allow [3].

Binary decision diagrams (BDDs) are often used in classical planning as a knowledge compilation for symbolic search. In the Planning and Optimization lecture by Helmert and Röger (2020) [4], BDDs are established as reduced and ordered BDDs (Bryant 1985 [1]), to be canonical. To be able to practically use BDDs for a symbolic breadth-first search algorithm on a propositional planning task, it is important that all necessary transformations and queries are defined for BDDs and run in polynomial time.

Merge-and-shrink abstraction is a framework for deriving abstractions in factored state-space search problems (Helmert et al. 2015 [5]). Merge-and-shrink abstractions have been studied before as a generalization of pattern databases and have been primarily used as a heuristic for optimal planning. Recently the merge-and-shrink framework has also been studied as a "toolbox" of transformations on factored transition systems (Sievers and Helmert [6]). In this thesis, we investigate the merge-and-shrink representations as a knowledge compilation language called Factored Mappings (FMs).

Can FMs be used as a knowledge compilation for symbolic search? To see if this is possible we look at all properties necessary for BDDs to be used for a symbolic breadth-first search algorithm and check if they can be achieved in a similar way for FMs. BDDs have to be canonical and all necessary transformations and queries are defined and run in polynomial time. This means that we have to find a canonical representation for FMs and define all necessary operations in such a way that they run in polynomial time. This allows FMs to be used for a similar symbolic breadth-first search algorithm. We will first define a canonical representation of FMs and then look at a set of operations on FMs which are necessary to convert propositional formulas to FMs. Then, we will define an apply function that stores the states to which we can transition to using operations from our planning task in an FM. With this, we will be able to construct a symbolic breadth-first search algorithm that uses FMs as a knowledge compilation instead of BDDs. To construct the algorithm, we are going to use similar methods as shown for BDDs in chapter B8 of the Planning and Optimization lecture [4]. BDDs, however, have a distinct variable order to help achieve canonicity. This is different to FMs, where we have no strict variable order but a fixed underlying merge tree structure. This makes FMs much more similar to a Sentential Decision Diagram (SDD), which is a knowledge compilation that has their variable order hidden inside of a vtree (Darwiche 2011 [2]). Vtrees in SDDs show a strong resemblance to merge trees in FMs, however SDDs will not be further discussed in this thesis, since we will focus more on defining all transformations and queries necessary for symbolic search with FMs and their time complexity. One advantage of using FMs for symbolic search is that their state variables can have any finite domain and are easily usable on finite-domain (FDR) planning tasks on top of propositional planning tasks. The symbolic search algorithm we construct for FMs will therefore also work with FDR planning tasks.

2

Background

We will first give a bit of conceptual background and then show more relevant definitions that we will be using during the thesis.

Planning tasks In chapter A4 of the Planning and Optimization lecture by Helmert and Röger (2020) [4] planning tasks are introduced as compact representations of transition systems that are suitable as inputs for planning algorithms. A planning task is given by $\langle V, I, O, \gamma \rangle$ where

- V is a finite set of state variables,
- I is a valuation over V called the initial state,
- O is a finite set of operators over V , and
- γ is a formula over V called the goal.

We distinguish between propositional planning tasks, in which all state variables can be only true or false, and planning tasks in finite-domain representation (FDR), in which all state variables can take any value within a finite domain.

Binary Decision Diagrams Ordered binary decision diagrams (BDDs) are a data structure used to represent Boolean functions inside of acyclic graphs (Bryant 1985 [1]). They have a fixed variable order and can be used in a reduced form as a knowledge compilation for symbolic search, where variables represent states which can be set to true or false. Therefore they are well usable for propositional planning tasks.

Factored Mappings Factored Mappings (FMs), also called merge-and-shrink representations have been used in state-space search to represent functions that map states to numerical heuristic values. More generally we can use them to represent functions that map variable assignments to a set of values (Helmert et al. 2015 [5]). We regard FMs as a data structure with an underlying merge tree and we will show that they can be used as a knowledge compilation for symbolic search, where variables represent states which have a finite domain.

This means that if we can use FMs for symbolic search for propositional planning tasks, then we can use them for FDR planning tasks as well.

2.1 Definitions

Factored Mappings are directly based on the merge-and-shrink representations as seen in Helmert et al. (2015) [5] and on their treatment as Factored Mappings as seen in Sievers and Helmert [6]. The first six definitions follow directly from these papers.

Definition 1 (variable set). A variable set is a set $V = \{v_1, \dots, v_n\}$ of different variables with a finite domain. We write $dom(v)$ for the domain of $v \in V$, which can be an arbitrary finite set of values.¹

Definition 2 (assignment). A partial assignment α is an assignment of a subset $V' \subseteq V$, where V is a variable set. It maps every $v \in V'$ to some element $\alpha[v] \in dom(v)$. We write $vars(\alpha)$ instead of V' for the set of variables on which α is defined. A partial assignment with $vars(\alpha) = V$ is called an assignment.

Definition 3 (factored mappings). Factored Mappings (FMs) over a variable set V are inductively defined as follows. An FM σ has an associated finite value set $vals(\sigma) \neq \emptyset$ and an associated table σ^{tab} . σ is either atomic or a merge.

- If σ is atomic, then it has an associated variable $v \in V$. Its table is a partial function $\sigma^{tab} : dom(v) \mapsto vals(\sigma)$.
- If σ is a merge, then it has a left component FM σ_L and a right component FM σ_R . Its table is a partial function $\sigma^{tab} : vals(\sigma_L) \times vals(\sigma_R) \mapsto vals(\sigma)$.

Definition 4 (set of assignments). $A(V) = dom(v_1) \times \dots \times dom(v_n)$ is the set of all assignments α over V .

Definition 5 (represented function). Let σ be an FM over a variable set V . It represents the function $[\sigma] : A(V) \mapsto vals(\sigma)$ which is inductively defined as follows:

- If σ is atomic with associated variable v , then $[\sigma](\alpha) = \sigma^{tab}(\alpha[v])$.
- If σ is a merge, then $[\sigma](\alpha) = \sigma^{tab}([\sigma_L](\alpha), [\sigma_R](\alpha))$.

Definition 6 (size of an FM). The size of an FM σ , written as $|\sigma|$, and the size of its table, written as $|\sigma^{tab}|$, are defined inductively as follows:

- If σ is atomic with associated variable v , then $|\sigma^{tab}| = |\sigma| = |dom(v)|$.
- If σ is a merge, then $|\sigma^{tab}| = |vals(\sigma_L)| \cdot |vals(\sigma_R)|$ and $|\sigma| = |\sigma_L| + |\sigma_R| + |\sigma^{tab}|$

Remark. The underlying structure of FMs is given by merge trees. We will often use terminology for trees on FMs.

¹ It makes sense to assume $dom(v) \neq \emptyset$, else we would simply use the variable set $V \setminus \{v\}$.

Definition 7 (merge tree). A merge tree $\mathcal{T}(V)$ is a binary tree over a variable set V . We write $\mathcal{T}(\sigma)$ as the merge tree that is underlying to the FM σ .

Definition 8 (set of associated variables). Let V be a variable set. Every FM σ over V has a set of associated variables $\text{vars}(\sigma) \subseteq V$.²

- If σ is atomic with associated variable v , then $\text{vars}(\sigma) = \{v\}$
- If σ is a merge, then $\text{vars}(\sigma) = \text{vars}(\sigma_L) \cup \text{vars}(\sigma_R)$

In the merge tree $\mathcal{T}(V)$ every node u has a set of associated variables $\text{vars}(u) \subseteq V$.

- If u is a leaf node that refers to the singular variable v , then $\text{vars}(u) = \{v\}$
- If u is a parent node, then $\text{vars}(u) = \text{vars}(u_L) \cup \text{vars}(u_R)$, where u_L is the left child and u_R the right child of u .

Remark. We often write the set of associated variables of an FM σ in its name to clarify which FM component of σ the name stands for: $\text{vars}(\sigma_{vwx}) = \{v, w, x\}$. This helps to show which FMs matches with which corresponding node in a merge tree.

Definition 9 (corresponds). A node u of an underlying merge tree $\mathcal{T}(\sigma)$ *corresponds* to an FM σ if and only if $\text{vars}(u) = \text{vars}(\sigma)$. If two FMs σ and γ correspond to the same node such that $\text{vars}(\sigma) = \text{vars}(\gamma)$, then σ *corresponds* to γ .

Atomic FMs correspond to leaf nodes and merge FMs to inner nodes. We call the FM component of an FM σ that corresponds to the root node in $\mathcal{T}(\sigma)$ the *root FM* σ_{root} and its table function σ_{root}^{tab} the *root table*.

Definition 10 (orthogonal). An FM σ is called *orthogonal* if all atomic FMs of σ have different associated variables. Then all leaf nodes of the merge tree $\mathcal{T}(\sigma)$ corresponding to the atomic FMs of σ have different variables as well. $\mathcal{T}(\sigma)$ is also called *orthogonal*.

Definition 11 (full). An FM σ over V is called *full* if and only if $\text{vars}(\sigma) = V$. Then $\text{vars}(\sigma_{root}) = V$, where σ_{root} corresponds to the root of the merge tree $\mathcal{T}(\sigma)$. $\mathcal{T}(\sigma)$ is also called *full*.

Definition 12 (restriction on roots of FMs). The value set of the root of an FM σ is restricted to: $\text{vals}(\sigma_{root}) := \{0, 1\}$.

For our purposes we will use factored mappings to store variable assignments α . So a binary representation in the root table to show if α is stored in the FM suffices.

Definition 13 (assignments of an FM). Let σ be an FM over a variable set V . $A(\sigma) \subseteq A(V)$ is the set of all assignments α stored in σ . $\alpha \in A(\sigma)$ if and only if $[\sigma](\alpha) = 1$.

² Not to be mistaken with $\text{vals}(\sigma)$.

3

Canonicity

To use Factored Mappings in symbolic search we need to compare them for equality. For this it is important that FMs representing the same set of assignments are identical.

Definition 14 (identity). For two FMs σ and γ to be identical (written $\sigma = \gamma$), the following must be true:

- they have the same underlying merge tree: $\mathcal{T}(\sigma) = \mathcal{T}(\gamma)$
- for all σ_i and γ_i : if $\text{vars}(\sigma_i) = \text{vars}(\gamma_i)$ then $\sigma_i^{tab} = \gamma_i^{tab}$.

where σ_i and γ_i are FM components of σ and γ respectively.

Remark. We want to achieve a canonical representation of FMs that satisfies: $A(\sigma) = A(\gamma)$ if and only if $\sigma = \gamma$.

3.1 Requirements

For the representation of Factored Mappings to be canonical, we need it to fulfill two requirements. We will now introduce the requirements and then show why they are sufficient (and necessary) to satisfy the property described above.

Since we will be using FMs for symbolic search within the context of a specific planning task, all of our FMs will be over the same variable set V . We want them to have the same underlying merge tree $\mathcal{T}(V)$, so we can compare corresponding FMs. Because all FMs share the same $\mathcal{T}(V)$, they need to be full. It also does not make sense for any FM to have more than one atomic FM per variable, because we cannot assign different values to the same variable within one assignment α .

Requirement 1. *All FMs defined over some variable set V must have the same underlying merge tree $\mathcal{T}(V)$. Additionally, all FMs must be full and orthogonal, and therefore $\mathcal{T}(V)$ is full and orthogonal as well.*

Since tables of FM components, that are not the root table, are filled with arbitrary values that represent different entries, we need to ensure that they are filled consistently.

Requirement 2. We want the tables σ_i^{tab} for all non-root components σ_i of any FM σ to map to the values $vals(\sigma_i) = \{0, 1, \dots, n-1\}$, where

$$n = \begin{cases} |vals(\sigma_{iL}) \times vals(\sigma_{iR})| & \text{If } \sigma_i \text{ is a merge} \\ |dom(v_i)| & \text{If } \sigma_i \text{ is atomic with associated variable } v_i \end{cases}$$

We use a fixed order that always takes the smallest unused value to fill the tables from left to right, top to bottom. The root FM σ_{root} will still only map to 0 and 1 (see Definition 12). Its table function σ_{root}^{tab} is given by the condition that $[\sigma](\alpha) = 1$ if and only if $\alpha \in A(\sigma)$ (see Definition 13).

Later we will consider reducing table entries and allow mappings to already used values, but for now we stick with this naive mapping to n different values in order.

These two requirements are already enough to achieve the canonical representation of FMs that we wanted.

Theorem 1. Let σ and γ be two FMs that follow Requirements 1 and 2. Then $A(\sigma) = A(\gamma)$ if and only if $\sigma = \gamma$.

Proof. First we show that if $\sigma = \gamma$ then $A(\sigma) = A(\gamma)$: From the definition of $\sigma = \gamma$ (see Definition 14) we know that they have the same merge tree and all table functions are equal. That means that all entries in all tables are identical including the root table. It is trivial to see that the same assignments have to be stored in σ and γ . Therefore it follows directly from $\sigma = \gamma$ that $A(\sigma) = A(\gamma)$.

Now we show that if $A(\sigma) = A(\gamma)$ then $\sigma = \gamma$: From Requirement 1 we know that $\mathcal{T}(\sigma) = \mathcal{T}(\gamma)$. Because the merge trees are orthogonal and full we know that for all variables $v \in V$, both FMs have exactly one atomic component referring to v . Requirement 2 fills the tables of all atomic components with the same entries in the fixed value order from 0 to $|dom(v)| - 1$. Therefore all atomic FMs that refer to the same variable have identical entries. We can see that merging already identical FM tables will create a parent table that is spanned by the same values. For both FMs this parent table will also be filled with the same entries given by the value order of Requirement 2. By induction, this holds for all tables up to the root table, where it no longer holds. There σ_{root}^{tab} and γ_{root}^{tab} are still spanned by the same values. Except in the root table the entries are given in such a way that $[\sigma](\alpha) = 1$ if $\alpha \in A(\sigma)$ and $[\gamma](\alpha) = 1$ if $\alpha \in A(\gamma)$. Because all other components in σ and γ are identical, the entry at $\sigma_{root}^{tab}([\sigma_L](\alpha), [\sigma_R](\alpha))$ and $\gamma_{root}^{tab}([\gamma_L](\alpha), [\gamma_R](\alpha))$ is at the same index. And because $A(\sigma) = A(\gamma)$ this entry in σ_{root}^{tab} and γ_{root}^{tab} must either be mapped to 1 if $\alpha \in A(\sigma)$, or to 0. Therefore all entries in all tables of σ and γ are equal. This, with the fact that $\mathcal{T}(\sigma) = \mathcal{T}(\gamma)$, implies $\sigma = \gamma$ by Definition 14. \square

We can show that these two requirements are also necessary, if we look at what happens if one is not fulfilled for σ or γ . If Requirement 1 is not fulfilled and $\mathcal{T}(\sigma) \neq \mathcal{T}(\gamma)$, then Definition 14 directly tells us that σ and γ can never be identical. Let σ and γ be two FMs, where σ follows Requirement 2 but γ differs from σ in one value of a component

FM, therefore γ does not fulfill Requirement 2. Let the value in which γ differs from σ be an otherwise unused value. Since values can be arbitrary, as long as they represent different entries, and the root tables of both FMs are still identical, they still store the same assignments. Therefore $A(\sigma) = A(\gamma)$, but because σ and γ differ in an entry, $\sigma \neq \gamma$.

If we look at the current canonical representation of FMs we are using, we can see that all FMs will only ever differ inside of the root table. Filling every component table maximally by giving each entry a different value, takes an exponential amount of space, because of the repeated quadratic growth of each merge σ_m , for which we need space for $X \cdot Y$ different entries, where $X = \text{vals}(\sigma_{mL})$ and $Y = \text{vals}(\sigma_{mR})$. This is due to the naive way of filling the table entries from Requirement 2. But it is not necessary for all entries to be differentiated, because they sometimes get mapped to the same values in the parent table. Therefore, we revisit Requirement 2 so that we can reduce FMs later on to save space and get rid of redundant table entries.

Instead of filling all entries naively with all different values, we want to allow a value to be used for multiple entries if they will map to equivalent rows/columns. Multiple rows/columns are equivalent to each other if the entries at the same position have the same value. Because the FM does not have to differentiate between them, they can be reduced to just one row/column. In the root table we already allow multiple entries to be mapped to the same value. There, we can already have multiple equivalent rows/columns. Reducing them to one by using the same value in its children for the table entries whose old values were the indices for the equivalent rows/columns. This can lead to new equivalent rows/columns in the tables of the children, which then can be reduced themselves. This can be done for all merge FMs recursively and will not change the assignments stored.

Requirement 3. *The tables σ_i^{tab} for all components σ_i of any FM σ map to the values $\text{vals}(\sigma_i) = \{0, 1, \dots\}$. We fill the tables from left to right, top to bottom by using the smallest unused value if the parent table has to differentiate between them, or using the same value as the already filled entry that leads to an identical row/column in the parent table. As in Requirement 2, the root FM σ_{root} will still only map to 0 and 1 and its table function σ_{root}^{tab} is given by the condition that $[\sigma](\alpha) = 1$ if and only if $\alpha \in A(\sigma)$.*

Table 3.1 is an example for different tables σ^{tab} , where σ is a merge of σ_L and σ_R . $\text{vals}(\sigma_L) = \{0, 1, 2\}$ and $\text{vals}(\sigma_R) = \{0, 1\}$. $\sigma^{tab} : \{0, 1, 2\} \times \{0, 1\} \mapsto \text{vals}(\sigma) = \{0, 1, \dots, n - 1\}$ with $n \leq 3 \cdot 2$:

σ^{tab}	0	1	σ^{tab}	0	1	σ^{tab}	0	1
0	0	1	0	0	1	0	0	1
1	2	3	1	0	2	1	0	3
2	4	5	2	1	3	2	2	1
R2 and R3			Only R3			Neither		

Table 3.1: The first table fulfills Requirements 2 and 3, the second table just Requirement 3 and the third table fulfills neither.

Definition 15 (reduced). If the size of $vals(\sigma_i)$ for each component σ_i in σ is minimal among all possible FMs with the same represented function, then σ is called a *reduced* FM.

FMs that follow Requirement 3 have a minimal value set and are therefore always reduced. If they also follow Requirement 1, we call them *canonical and reduced*.

Example Fig. 3.1 shows an example of a not yet reduced FM σ with variable set $V = \{v, w, x\}$ and $dom(v) = dom(w) = dom(x) = \{0, 1, 2\}$. Let $A(\sigma) = \{\alpha\} = \{\{v \mapsto 0, w \mapsto 1, x \mapsto 2\}\}$ be the set of assignments stored in σ . Given the underlying merge tree $\mathcal{T}(V)$, we can draw the tables of the components and root of σ following the initial Requirement 2:

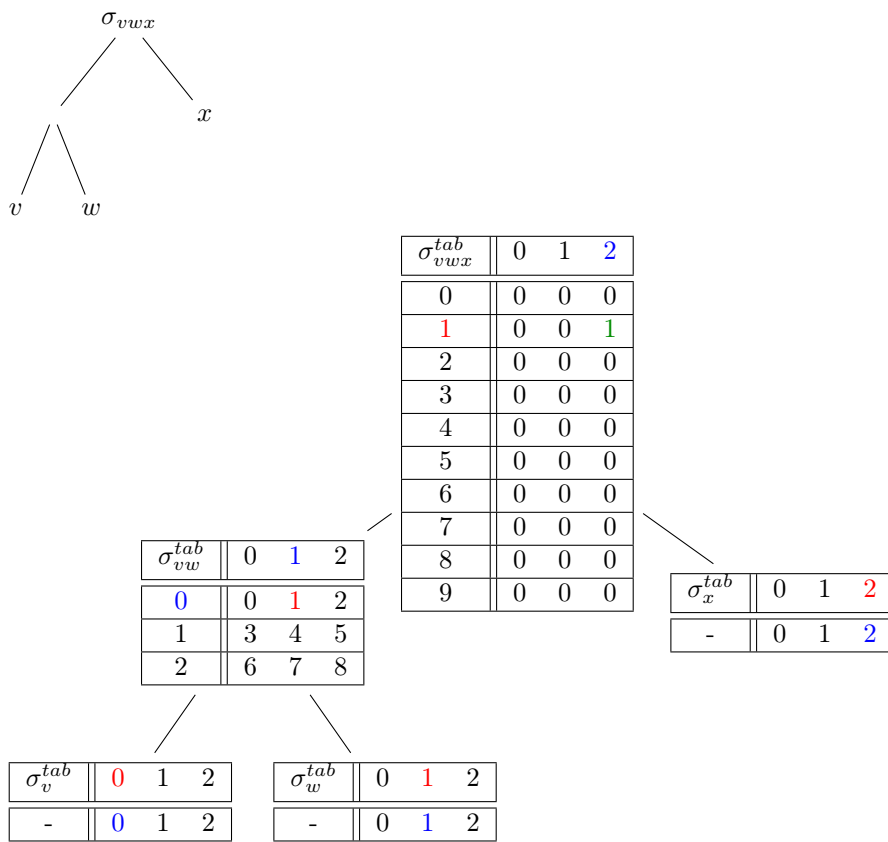


Figure 3.1: Colored entries represent the stored assignment α . Entries in σ_i are in the same color as the index of their row/column in the parent table. The merge tree $\mathcal{T}(V)$ is shown in the top left.

We can see that there are many rows (and two columns) in σ_{vwx}^{tab} where each entry is a 0. We want to reduce them to just one row (and one column), to get rid of these redundant entries.

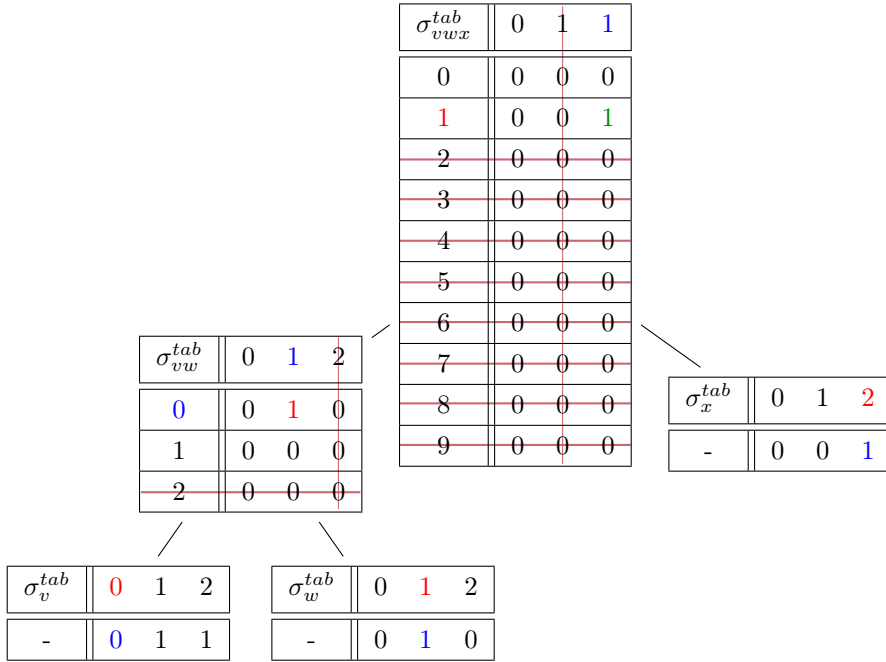
3.2 Reducing Factored Mappings

To reduce a Factored Mapping σ , we have to ensure that $|vals(\sigma_i)|$ for each component σ_i in σ cannot be further decreased without changing $A(\sigma)$. For this we will map entries in σ_i to the same value whenever the parent FM does not have to differentiate between them, because they lead to equivalent rows/columns. To transform σ into the reduced σ' that follows Requirement 3 we can use Algorithm 1.

Time complexity of REDUCE(σ) REDUCE(σ) calls COMPUTEROWPARTITIONING(σ) which goes over every row in σ and partitions them into lists of equivalent rows. For this it has to check every entry in σ once, which are $|\sigma^{tab}| = |vals(\sigma_L)| \cdot |vals(\sigma_R)|$ checks. Then, REDUCE(σ) goes through all $|vals(\sigma_L)|$ rows preparing their renaming and adding one per equivalentRows to the new mapping $\tilde{\sigma}^{tab}$. APPLYRENAMING is called, which goes through all $|\sigma_L^{tab}|$ entries once to apply the renaming. This leads to $\mathcal{O}(|\sigma^{tab}| + |vals(\sigma_L)| + |\sigma_L^{tab}|)$ operations for the rows of a merge node. For columns this is done analogously. Therefore we have $\mathcal{O}(2 \cdot (T^{max} + T^{max} + T^{max})) = \mathcal{O}(T^{max})$ operations per merge node, where T^{max} is the maximum over all table sizes of σ (Helmert et al. 2015 [5]). Let $n = |V|$ be the number of all variables in V , then $\mathcal{T}(\sigma)$ has $n - 1$ merge nodes. REDUCE(σ) therefore has a time complexity of $\mathcal{O}((n - 1) \cdot T^{max}) = \mathcal{O}(n \cdot T^{max})$.

Remark. The size of σ can be estimated by adding T^{max} for all $2n - 1$ nodes: $|\sigma| = \mathcal{O}((2n - 1) \cdot T^{max}) = \mathcal{O}(n \cdot T^{max})$ (Helmert et al. 2015 [5]). This means that REDUCE(σ) is linear in the size of the FM.

Example (continued) We revisit our example from before with variable set $V = \{v, w, x\}$, $dom(v) = dom(w) = dom(x) = \{0, 1, 2\}$ and $A(\sigma) = \{\{v \mapsto 0, w \mapsto 1, x \mapsto 2\}\}$. We see how the unnecessary rows and columns of σ disappear in its reduced form:

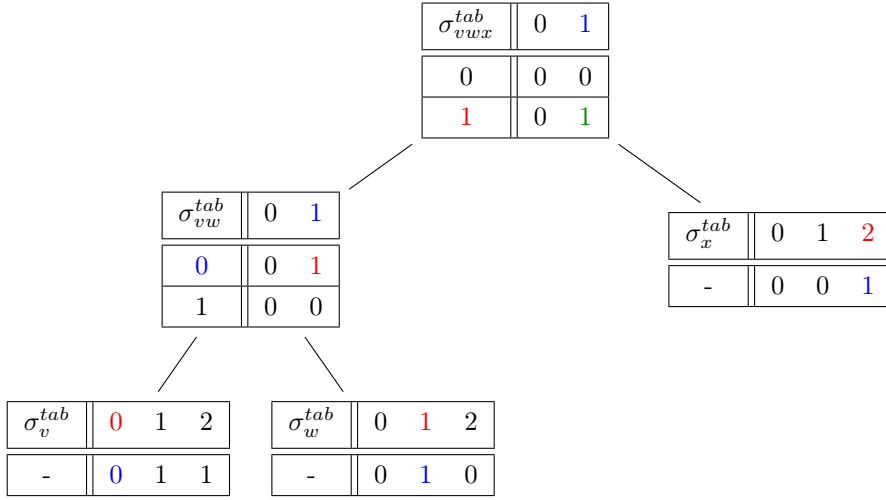


Algorithm 1 Reducing an FM σ to fit Requirement 3.

```

1: function COMPUTEROWPARTITIONING( $\sigma$ )
2:    $rowPartitioning \leftarrow []$ 
3:    $visited$  is a list of size  $|vals(\sigma_L)|$  with each entry set to false
4:   for row index  $i \in vals(\sigma_L)$  and not  $visited[i]$  do
5:      $equivalentRows \leftarrow []$ 
6:     append  $i$  to  $equivalentRows$ 
7:      $visited[i] \leftarrow true$ 
8:     for row index  $j \in vals(\sigma_L)$  with  $j > i$  and not  $visited[j]$  do
9:       if  $\sigma^{tab}[i][y] = \sigma^{tab}[j][y] \forall y$  then
10:        append  $j$  to  $equivalentRows$ 
11:         $visited[j] \leftarrow true$ 
12:     append  $equivalentRows$  to  $rowPartitioning$ 
13:   return  $rowPartitioning$ 
14: function COMPUTECOLUMNPARTITIONING( $\sigma$ )
15:    $columnPartitioning \leftarrow []$ 
16:    $visited$  is a list of size  $|vals(\sigma_R)|$  with each entry set to false
17:   for column index  $i \in vals(\sigma_R)$  and not  $visited[i]$  do
18:      $equivalentColumns \leftarrow []$ 
19:     append  $i$  to  $equivalentColumns$ 
20:      $visited[i] \leftarrow true$ 
21:     for column index  $j \in vals(\sigma_R)$  with  $j > i$  and not  $visited[j]$  do
22:       if  $\sigma^{tab}[x][i] = \sigma^{tab}[x][j] \forall x$  then
23:        append  $j$  to  $equivalentColumns$ 
24:         $visited[j] \leftarrow true$ 
25:     append  $equivalentColumns$  to  $columnPartitioning$ 
26:   return  $columnPartitioning$ 
27: function APPLYRENAMING( $\sigma, renaming$ )
28:   for  $entry \in \sigma^{tab}$  do
29:      $entry \leftarrow renaming[entry]$ 
30: function REDUCE( $\sigma$ )
31:    $rowPartitioning \leftarrow COMPUTEROWPARTITIONING(\sigma)$ 
32:    $\tilde{\sigma}^{tab} \leftarrow [[]]$ 
33:    $newRowIndex \leftarrow 0$ 
34:    $renaming$  is a list of size  $|vals(\sigma_L)|$  with each entry set to 0
35:   for  $equivalentRows \in rowPartitioning$  do
36:     for all  $y \in vals(\sigma_R)$  do
37:       append  $\sigma^{tab}[equivalentRows[0]][y]$  to  $\tilde{\sigma}^{tab}$ 
38:     for row index  $i \in equivalentRows$  do
39:        $renaming[i] \leftarrow newRowIndex$ 
40:      $newRowIndex \leftarrow newRowIndex + 1$ 
41:    $\sigma^{tab} \leftarrow \tilde{\sigma}^{tab}$ 
42:   APPLYRENAMING( $\sigma_L, renaming$ )
43:    $columnPartitioning \leftarrow COMPUTECOLUMNPARTITIONING(\sigma)$ 
44:    $\tilde{\sigma}^{tab} \leftarrow [[]]$ 
45:    $newColumnIndex \leftarrow 0$ 
46:    $renaming$  is a list of size  $|vals(\sigma_R)|$  with each entry set to 0
47:   for  $equivalentColumns \in columnPartitioning$  do
48:     for all  $x \in vals(\sigma_L)$  do
49:       append  $\sigma^{tab}[x][equivalentColumns[0]]$  to  $\tilde{\sigma}^{tab}$ 
50:     for column index  $i \in equivalentColumns$  do
51:        $renaming[i] \leftarrow newColumnIndex$ 
52:      $newColumnIndex \leftarrow newColumnIndex + 1$ 
53:    $\sigma^{tab} \leftarrow \tilde{\sigma}^{tab}$ 
54:   APPLYRENAMING( $\sigma_R, renaming$ )
55:   if  $\sigma_L$  is not atomic then REDUCE( $\sigma_L$ )
56:   if  $\sigma_R$  is not atomic then REDUCE( $\sigma_R$ )

```



All tables now store a minimal amount of different variables within a given table. σ is now reduced. We will usually already work with canonical and reduced FMs and try to construct them directly in that form. We only have to reduce FMs when we create redundancies within them through an operation.

Canonical and reduced FMs still have the canonicity property that we want.

Theorem 2. *Let σ and γ be two FMs that follow Requirements 1 and 3. Then, $A(\sigma) = A(\gamma)$ if and only if $\sigma = \gamma$.*

Proof. If $\sigma = \gamma$ then $A(\sigma) = A(\gamma)$ still follows directly. We know that FMs that follow Requirement 2 instead of Requirement 3 have the property that if $A(\sigma) = A(\gamma)$ then $\sigma = \gamma$. We need to show that this property is not lost by reducing them. This is true if σ always reduces to the same σ' . Because $\text{reduce}(\sigma)$ is a deterministic algorithm we know that given the same σ as input it always reduces to the same σ' . Therefore $A(\sigma) = A(\gamma)$ if and only if $\sigma = \gamma$ also holds when only working with canonical and reduced FMs. \square

We remark that we have two canonical representations of FMs that satisfy: $A(\sigma) = A(\gamma)$ if and only if $\sigma = \gamma$. The first one uses Requirement 2 to have a maximal amount of different values stored. The second one uses Requirement 3 to have a minimal amount of different values stored. For space saving purposes we will choose the latter canonical representation and only work with canonical and reduced FMs. Any future FM is assumed to be canonical and reduced unless specified otherwise.

4

Operations on Factored Mappings

We will now take a look at different operations for creating, querying and transforming Factored Mappings. These operations will be used to convert propositional formulas to FMs later (Section 5.1). First, we will look at how to create basic FMs that represent logical atoms, then we will see how to perform Boolean tests as queries on FMs, and finally we will see how to apply logical connectives by using FM transformations.

For all operations on Factored Mappings we always assume that all involved FMs are canonical and reduced as described above.

4.1 Building Basic Factored Mappings

The first operations we look at are not operations on Factored Mappings but operations to build some basic Factored Mappings directly. These operations will be used to represent logical atoms when converting propositional formulas later (Section 5.1).

The FMs we create have the underlying merge tree $\mathcal{T}(V)$, which can be understood as a global variable and can be derived from the the variable set V .

4.1.1 False

We want to create a Factored Mapping representing the empty set. The empty FM σ_F with $A(\sigma_F) = \{\}$ is created by filling all atomic leaf tables of σ_F with the same value. For Requirement 3 this value has to be 0. This leads to all merge tables in σ_F only having one entry, that is also 0. The root table will map its one entry to 0, so that no assignments are stored.

Algorithm 2 Creating the empty FM σ_F

```

1: function FALSE( $V$ )
2:   for node  $u \in \mathcal{T}(V)$  do
3:      $\sigma_u^{tab} \leftarrow [][ ]$ 
4:     if  $u$  is a leaf then
5:       for  $y \in |dom(vars(u))|$  do
6:          $\sigma_u^{tab}[y] \leftarrow 0$ 
7:        $\sigma_u \leftarrow$  atomic FM with the variable  $vars(u)$  with table  $\sigma_u^{tab}$ 
8:     else
9:        $\sigma_u^{tab}[0][0] \leftarrow 0$ 
10:       $\sigma_u \leftarrow$  merge FM with components  $\sigma_{uL}$  and  $\sigma_{uR}$  and with table  $\sigma_u^{tab}$ 
11:    if  $u$  is the root node then
12:      return  $\sigma_u$ 

```

4.1.2 True

We can create a Factored Mapping σ_T that stores all assignments with the same process as for the empty FM σ_F . The only difference is in the root table where we want to map the one entry to 1 instead of 0. This way the full FM σ_T stores every assignment $A(\sigma_T) = A(V)$. Swapping the root table entries like this is a process that we will also use for the complement later (Section 4.3.3). This can be seen intuitively because $\overline{A(\sigma_F)} = A(V) \setminus A(\sigma_F) = A(V) \setminus \{\} = A(V) = A(\sigma_T)$.

Algorithm 3 Creating σ_T as the FM that stores all assignments

```

1: function TRUE( $V$ )
2:    $\sigma_T \leftarrow$  FALSE( $V$ ) ▷ see Algorithm 2
3:    $\sigma_T^{tab}[0][0] \leftarrow 1$ 
4:   return  $\sigma_T$ 

```

Time complexity FALSE(V) and TRUE(V) both go through n leaf nodes with associated variable v and create an atomic FM with $|dom(v)|$ entries, where $n = |V|$. The other $n - 1$ node tables are filled with just one entry. Therefore, their time complexity is $\mathcal{O}(n \cdot D^{max} + (n - 1)) = \mathcal{O}(n \cdot D^{max})$, where $D^{max} = \max_{v \in V} |dom(v)|$ (Helmert et al. 2015 [5]).

4.1.3 Atom

We will now create a Factored Mapping σ that stores all assignments in which a variable v is set to the specific value $c \in dom(v)$. For this we fill all atomic tables σ_w^{tab} where $w \neq v$ with zeroes as we did for the empty FM. σ_v^{tab} has entries with two different values, one that is used for the entry at the index of the c and one that is used for all other entries. The entry $\sigma_v^{tab}(c)$ will map to 0 if $c = \min_{i \in dom(v)} i$, and to 1 otherwise. Due to Requirement 3 all other entries will map to 1 if $\sigma_v^{tab}(c) = 0$ and to 0 if $\sigma_v^{tab}(c) = 1$. Merge tables in σ have either one zero entry or two entries (0 and 1) if one of its children had two different values. This means that the only 1 we find in the root table comes from the partial assignment of $v \mapsto c$. The partial assignment of the other variables is meaningless because they all get mapped to the same value 0. Therefore $A(\sigma) = \{\alpha \mid \alpha[v] = c\}$.

Algorithm 4 Creating the atom FM σ that stores all α with $\alpha[v] = c$

```

1: function ATOM( $V, v, c$ )
2:   for node  $u \in \mathcal{T}(V)$  do
3:      $\sigma_u^{tab} \leftarrow [ ] [ ]$ 
4:     if  $u$  is a leaf then
5:       for  $y \in |dom(vars(u))|$  do
6:          $\sigma_u^{tab}[y] \leftarrow \begin{cases} 1 & \text{if } vars(u) = \{v\} \text{ and } c = \min_{i \in dom(v)} i \\ 0 & \text{otherwise} \end{cases}$ 
7:       if  $vars(u) = \{v\}$  then
8:          $\sigma_u^{tab}[c] \leftarrow \begin{cases} 0 & \text{if } c = \min_{i \in dom(v)} i \\ 1 & \text{otherwise} \end{cases}$ 
9:        $\sigma_u \leftarrow$  atomic FM with the variable  $vars(u)$  with table  $\sigma_u^{tab}$ 
10:    else
11:      for  $x \in vals(\sigma_{uL})$  do
12:        for  $y \in vals(\sigma_{uR})$  do
13:          if  $x = 0$  and  $y = 0$  then
14:             $\sigma_u^{tab}[x][y] \leftarrow 0$ 
15:          else
16:             $\sigma_u^{tab}[x][y] \leftarrow 1$ 
17:       $\sigma_u \leftarrow$  merge FM with components  $\sigma_{uL}$  and  $\sigma_{uR}$  and with table  $\sigma_u^{tab}$ 
18:    if  $u$  is the root node then
19:      return  $\sigma_u$ 

```

Time complexity of ATOM(V, v, c) As for computing the empty FM with FALSE(V), ATOM(V, v, c) goes through n leaf nodes and creates an atomic FM with $|dom(w)|$ entries, where $n = |V|$ and w is the associated variable of a leaf node. The other $n - 1$ node tables are filled by going through a maximum of 1×2 or 2×1 entries and map them to a maximum of two different entries (0 and 1). Therefore the time complexity of ATOM(V, v, c) is $\mathcal{O}(n \cdot D^{max} + (n - 1) \cdot 2) = \mathcal{O}(n \cdot D^{max})$, where $D^{max} = \max_{w \in V} |dom(w)|$.

4.2 Boolean Tests

We will look at how to perform the following queries on Factored Mappings:

- Is an assignment α included in an FM σ ?
- Are two FMs σ and γ equal to each other?

These operations will be used to perform Boolean tests on FMs when converting propositional formulas later (Section 5.1).

4.2.1 Includes

To see if an assignment α is stored in a Factored Mapping σ we have to check if $\alpha \in A(\sigma)$, which is case if and only if $[\sigma](\alpha) = 1$ (see Definition 13). The function $[\sigma]$ is directly represented by σ and the reason for the FM structure we use (see Definition 5). To check if $[\sigma](\alpha) = 1$ we need look up the value of every atomic table in σ at the entry for $\alpha[v]$, where

v is its associated variable and pass it on. Then, we use that value together with the value that is passed on from the other child as indices for the entry of the parent table, for which we look up its value and pass it on. We do this recursively up until the root node, where we can check if the entry at the stored values is equal to 1 or not.

Algorithm 5 Checking if α is included in $A(\sigma)$

```

1: function GETVALUE( $\sigma, \alpha$ )
2:   if  $\sigma$  is a merge FM then
3:      $x \leftarrow$  GETVALUE( $\sigma_L, \alpha$ )
4:      $y \leftarrow$  GETVALUE( $\sigma_R, \alpha$ )
5:     return  $\sigma^{tab}(x, y)$ 
6:   else
7:     return  $\sigma^{tab}(\alpha[vars(\sigma)])$ 
8: function INCLUDES( $\sigma, \alpha$ )
9:   if GETVALUE( $\sigma, \alpha$ ) = 1 then
10:    return True
11:  else
12:    return False

```

Time complexity of INCLUDES(α, σ) INCLUDES(α, σ) calls GETVALUE(α, σ), which recursively visits every component of σ once. We can directly return the entry of each component without having to go through all entries. Therefore the time complexity is given by $\mathcal{O}(2n - 1) = \mathcal{O}(n)$, where $n = |V|$.

4.2.2 Equals

For two Factored Mappings σ and γ that share the same merge tree $\mathcal{T}(V)$ to be equal, all entries of corresponding FM components must have the same tables (see Definition 14). Therefore, we need to go through the whole FMs σ and γ and check every entry for equality. This can be done using Algorithm 6.

Time complexity of EQUALS(σ, γ) It is easy to see that EQUALS(σ, γ) goes through all entries of all tables in both FMs exactly once to compare them. It is clear that the time complexity is basically the size of both FMs added. Therefore the time complexity is given by $\mathcal{O}((2n - 1) \cdot T_\sigma^{max} + (2n - 1) \cdot T_\gamma^{max}) = \mathcal{O}(n(T_\sigma^{max} + T_\gamma^{max}))$, where $n = |V|$ and T_σ^{max} and T_γ^{max} is the maximum over all table sizes of σ and γ respectively.

4.3 Set Operations

We will now see how we can implement the set operations union, intersection, complement and difference for Factored Mappings. These FM transformations will be used to apply logical connectives when converting propositional formulas later (Section 5.1).

Algorithm 6 Testing if $\sigma = \gamma$

```

1: function EQUALS( $\sigma, \gamma$ )
2:   if  $\sigma$  and  $\gamma$  are merge FMs then
3:     if  $|vals(\sigma_L)| \neq |vals(\gamma_L)|$  then
4:       return False
5:     if  $|vals(\sigma_R)| \neq |vals(\gamma_R)|$  then
6:       return False
7:     for  $x \in vals(\sigma_L)$  do
8:       for  $y \in vals(\sigma_R)$  do
9:         if  $\sigma^{tab}(x, y) \neq \gamma^{tab}(x, y)$  then
10:          return False
11:      $leftTreeEquals \leftarrow$  EQUALS( $\sigma_L, \gamma_L$ )
12:      $rightTreeEquals \leftarrow$  EQUALS( $\sigma_R, \gamma_R$ )
13:     return  $leftTreeEquals$  and  $rightTreeEquals$ 
14:   else
15:     for entry  $i \in dom(vars(\sigma))$  do
16:       if  $\sigma^{tab}(i) \neq \gamma^{tab}(i)$  then
17:         return False
18:     return True

```

4.3.1 Union

Creating a union of two Factored Mappings σ and γ into one FM δ efficiently requires a smarter approach than to manually figure out which assignments are stored in $A(\sigma)$ and $A(\gamma)$, then creating a maximally filled FM with the union of the assignments, and finally reducing the many redundancies. The simplest way to create a union is to directly compute the combination for each component table pair σ_i^{tab} and γ_i^{tab} , with $vars(\sigma_i) = vars(\gamma_i)$, into the component table δ_i^{tab} with $vars(\sigma_i) = vars(\delta_i)$. This combination δ_i has to keep all information of σ_i and γ_i . We differentiate between two cases:

1. σ_i and γ_i are atomic.
2. σ_i and γ_i are merges.

We will look at how the union for these two cases is created. Creating the union for such a table will leave us with 2-dimensional entries that we will get rid off when combining the tables together to form a union of two complete FMs.

Combining leaves For the union of two corresponding atomic leaf node tables σ_l^{tab} and γ_l^{tab} of an atomic FM with associated variable v , we use a function $\text{COMBINELEAVES}(\sigma_l^{tab}, \gamma_l^{tab})$. COMBINELEAVES takes two atomic FM tables and creates a new table δ_l^{tab} with $vals(\delta_l) = vals(\sigma_l) \times vals(\gamma_l)$. Its cells are filled by combining 1-dimensional entries in σ_l^{tab} and γ_l^{tab} to 2-dimensional ones:

$$\delta_l^{tab}(x) \mapsto (\sigma_l^{tab}(x), \gamma_l^{tab}(x))$$

Combining merges For the union of two corresponding merge node tables σ_m^{tab} and γ_m^{tab} , we use a function $\text{COMBINEMERGES}(\sigma_m^{tab}, \gamma_m^{tab}, \delta_{m_L}, \delta_{m_R})$. COMBINEMERGES needs

σ_l^{tab}	0	1	2	3	γ_l^{tab}	0	1	2	3
-	0	1	0	2	-	0	1	2	0

δ_l^{tab}	0	1	2	3
-	(0,0)	(1,1)	(0,2)	(2,0)

Figure 4.1: Two leaf node tables σ_l^{tab} and γ_l^{tab} get combined to one δ_l^{tab} with 2-dimensional values.

two merge tables and the already combined δ_{m_L} and δ_{m_R} ³ and creates a new table δ_m^{tab} , which is spanned by the 2-dimensional values in $vals(\delta_{m_L})$ and $vals(\delta_{m_R})$ ⁴ with $vals(\delta_m) = vals(\sigma_m) \times vals(\gamma_m)$. Its cells are filled by combining the 1-dimensional entries in σ_m^{tab} and γ_m^{tab} to 2-dimensional ones given by the 2-dimensional indices of the cell:

$$\delta_m^{tab}((x_1, x_2), (y_1, y_2)) \mapsto (\sigma_m^{tab}(x_1, y_1), \gamma_m^{tab}(x_2, y_2))$$

σ_m^{tab}	0	1	2	γ_m^{tab}	0	1	2
0	0	1	2	0	0	1	0
1	3	4	5	1	0	1	2
2	0	2	3	2	1	2	3

δ_m^{tab}	(0,0)	(1,1)	(0,2)	(2,0)
(0,0)	(0,0)	(1,1)	(0,0)	(2,0)
(1,1)	(3,0)	(4,1)	(3,2)	(5,0)
(2,2)	(0,1)	(2,2)	(0,3)	(3,1)

$\delta_{m_L}^{tab}$	0	1	2	3
-	(0,0)	(1,1)	(2,2)	(0,0)

$\delta_{m_R}^{tab}$	0	1	2	3
-	(0,0)	(1,1)	(0,2)	(2,0)

Figure 4.2: Two merge node tables σ_m^{tab} and γ_m^{tab} get combined to one δ_m^{tab} with 2-dimensional values. δ_m^{tab} is spanned by its already combined children $\delta_{m_L}^{tab}$ and $\delta_{m_R}^{tab}$ with 2-dimensional values. The colors show where in the tables of σ_m^{tab} and γ_m^{tab} the value for an entry in δ_m^{tab} can be found.

³ δ_{m_L} and δ_{m_R} have to be the result of COMBINELEAVES or COMBINEMERGES of the corresponding children of σ_m and γ_m .

⁴ This also means that δ_m^{tab} can have a higher dimension than σ_m^{tab} and γ_m^{tab} (more on this later).

Algorithm 7 Creating δ as the union of two FMs σ and γ

```

1: function RENAMING2D( $\sigma$ )
2:   renaming is a list of size  $|vals(\sigma)|$  with each entry set to 0
3:   newValue  $\leftarrow$  0
4:   for value  $(x, y) \in vals(\sigma)$  do
5:     renaming $[(x, y)] \leftarrow newValue$ 
6:     newValue  $\leftarrow newValue + 1$ 
7:   APPLYRENAMING( $\sigma, renaming$ ) ▷ see Algorithm 1
8: function COMBINE( $\sigma, \gamma$ )
9:   if  $\sigma$  and  $\gamma$  are merge FMs then
10:     $\delta_L \leftarrow COMBINE(\sigma_L, \gamma_L)$ 
11:     $\delta_R \leftarrow COMBINE(\sigma_R, \gamma_R)$ 
12:     $\delta^{tab} \leftarrow COMBINEMERGES(\sigma, \gamma, \delta_L, \delta_R)$ 
13:     $\delta \leftarrow$  merge FM with components  $\delta_L$  and  $\delta_R$  and with table  $\delta^{tab}$ 
14:    RENAMING2D( $\delta_L$ )
15:    RENAMING2D( $\delta_R$ )
16:   else
17:     $\delta^{tab} \leftarrow COMBINELEAVES(\sigma, \gamma)$ 
18:     $\delta \leftarrow$  atomic FM with the variable  $v = vars(\sigma) = vars(\gamma)$  with table  $\delta^{tab}$ 
19:   return  $\delta$ 
20: function UNION( $\sigma, \gamma$ )
21:    $\delta \leftarrow COMBINE(\sigma, \gamma)$ 
22:   for entry  $\in \delta^{tab}$  with value  $(x, y)$  do
23:     if  $x = 1$  or  $y = 1$  then
24:       entry  $\leftarrow 1$ 
25:     else
26:       entry  $\leftarrow 0$ 
27:   REDUCE( $\delta$ ) ▷ see Algorithm 1
28:   return  $\delta$ 

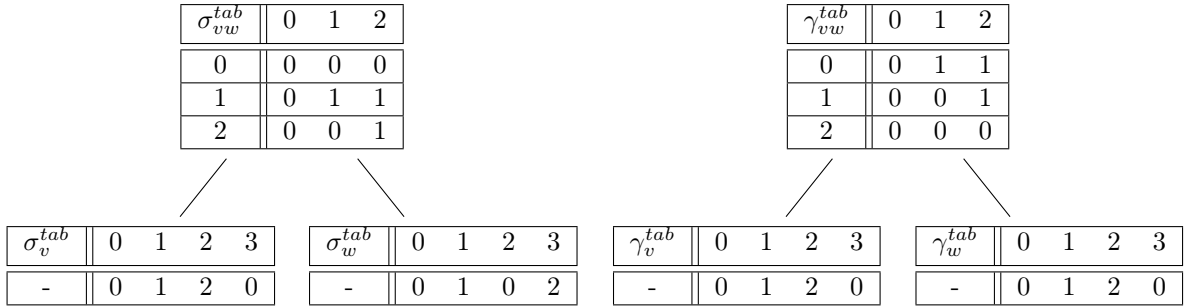
```

We will look at how Algorithm 7 works: $COMBINE(\sigma, \gamma)$ performs a post-order traversal of $\mathcal{T}(V)$. It therefore starts at the atomic FMs, where it combines all values from the atomic leaf tables of σ_l and γ_l into 2-dimensional ones, keeping all information. Those tuples are then given to the parent and are only renamed afterwards to fit Requirement 3. Combining merge tables always ends in 2-dimensional entries as well. We allow merge tables δ_m^{tab} to increase in dimension, compared to the dimensions of the original components, so that they can store all relevant combinations from σ_m and γ_m . Since merges also recursively give the tuples to the parent before renaming them, no information about the assignments is lost. In the root table δ^{tab} we end up with entries that have 2-dimensional values (x, y) , where x stores if the assignment was in σ and y if it was in γ . $UNION(\sigma, \gamma)$ then maps those entries to 1 if one of the FMs stored the assignment ($x = 1$ or $y = 1$), or 0 otherwise. Therefore, $A(\delta) = A(\sigma) \cup A(\gamma)$, which means that δ is an FM that stores exactly the union of σ and γ . Because δ could map entries that we differentiated before to the same value in the root table and might create new redundancies, we need to reduce δ to ensure that it is canonical and reduced.

Time complexity of $UNION(\sigma, \gamma)$ $UNION(\sigma, \gamma)$ calls $COMBINE(\sigma, \gamma)$ which goes through every node i in $\mathcal{T}(V)$. For leaf nodes it goes through all entries in σ_l^{tab} and γ_l^{tab} once

to create δ_l^{tab} . Those are $|\sigma_l^{tab}| + |\gamma_l^{tab}|$ entries per leaf node to go through. For merge nodes, this is different since they can increase in dimension. Since δ_m^{tab} is spanned by $vals(\delta_{m_L})$ and $vals(\delta_{m_R})$, which have the 2-dimensional values of $vals(\sigma_{m_L}) \times vals(\gamma_{m_L})$ and $vals(\sigma_{m_R}) \times vals(\gamma_{m_R})$, we see that $|\delta_m^{tab}| = |vals(\delta_{m_L})| \cdot |vals(\delta_{m_R})| = |vals(\sigma_{m_L}) \times vals(\gamma_{m_L})| \cdot |vals(\sigma_{m_R}) \times vals(\gamma_{m_R})|$. This means that $\text{COMBINE}(\sigma, \gamma)$ has to go through $|\delta_m^{tab}|$ entries per merge node. For every non-root FM their parent calls $\text{RENAMING2D}(\delta_i)$, which goes through $|vals(\delta_i)|$ entries, corresponding to rows or columns in the parent FM, to prepare their renaming. Then, APPLYRENAMING is called, which goes through all $|\delta_i^{tab}|$ entries once. This leads to $\mathcal{O}(|\sigma_l^{tab}| + |\gamma_l^{tab}| + |vals(\delta_l)| + |\delta_l^{tab}|)$ operations per leaf node, $\mathcal{O}(|\delta_m^{tab}| + |vals(\delta_m)| + |\delta_m^{tab}|)$ operations per non-root merge node, and $\mathcal{O}(|\delta_m^{tab}|)$ for the merge node. Let $n = |V|$ be the number of variables, $D^{max} = \max_{v \in V} |dom(v)|$ be the maximum over the domains, and T_σ^{max} and T_γ^{max} be the maximum over all table sizes of σ and γ respectively. We can see that $|\delta_m^{tab}| \leq T_\sigma^{max} \cdot T_\gamma^{max}$. Then, $\text{COMBINE}(\sigma, \gamma)$ has $\mathcal{O}(n(4 \cdot D^{max}) + (n-2)(3 \cdot T_\sigma^{max} \cdot T_\gamma^{max}) + T_\sigma^{max} \cdot T_\gamma^{max}) = \mathcal{O}(n(D^{max} + T_\sigma^{max} \cdot T_\gamma^{max}))$ operations. Because $T_\sigma^{max} \geq D^{max}$, we see that $\text{COMBINE}(\sigma, \gamma)$ has the time complexity of $\mathcal{O}(n \cdot T_\sigma^{max} \cdot T_\gamma^{max})$. Lastly, $\text{UNION}(\sigma, \gamma)$ calls $\text{REDUCE}(\delta)$ which has the time complexity of $\mathcal{O}(n \cdot T_\delta^{max}) = \mathcal{O}(n \cdot T_\sigma^{max} \cdot T_\gamma^{max})$. This leads to the combined time complexity for $\text{UNION}(\sigma, \gamma)$ of $\mathcal{O}(n \cdot T_\sigma^{max} \cdot T_\gamma^{max} + n \cdot T_\sigma^{max} \cdot T_\gamma^{max}) = \mathcal{O}(n \cdot T_\sigma^{max} \cdot T_\gamma^{max})$.

Example Let's see $\text{UNION}(\sigma, \gamma)$ on an example. Let $V = \{v, w\}$ be a variable set, and $dom(v) = dom(w) = \{0, 1, 2, 3\}$ be the domains of the variables. Let σ be the FM for the assignments $A(\sigma) = \{\{v \mapsto 1, w \mapsto 1\}, \{v \mapsto 1, w \mapsto 3\}, \{v \mapsto 2, w \mapsto 3\}\}$. Let γ be the FM for the assignments $A(\gamma) = \{\{v \mapsto 0, w \mapsto 1\}, \{v \mapsto 3, w \mapsto 1\}, \{v \mapsto 0, w \mapsto 2\}, \{v \mapsto 3, w \mapsto 2\}, \{v \mapsto 1, w \mapsto 2\}\}$. We will now compute the union of σ and γ into a new FM δ :

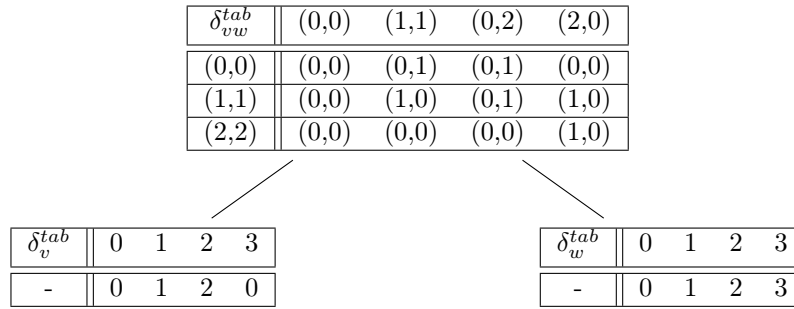


First $\text{UNION}(\sigma, \gamma)$ calls $\text{COMBINE}(\sigma_{vw}, \gamma_{vw})$, which will call itself with $\delta_v = \text{COMBINE}(\sigma_v, \gamma_v)$ and $\delta_w = \text{COMBINE}(\sigma_w, \gamma_w)$:

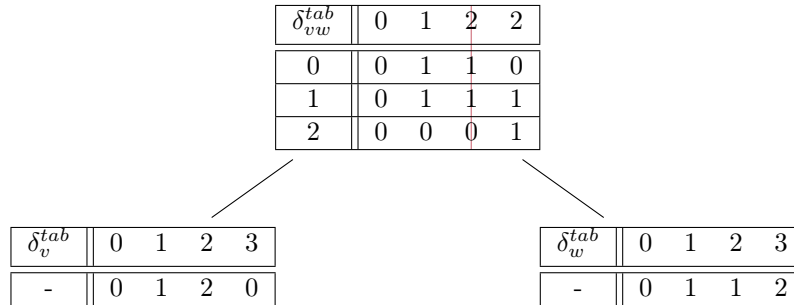
δ_v^{tab}	0	1	2	3
-	(0,0)	(1,1)	(2,2)	(0,0)

δ_w^{tab}	0	1	2	3
-	(0,0)	(1,1)	(0,2)	(2,0)

Then, $\text{COMBINE}(\sigma_{vw}, \gamma_{vw})$ creates the merge δ_{vw} , filling its table using $\text{COMBINEMERGES}(\sigma_{vw}^{tab}, \gamma_{vw}^{tab}, \delta_v, \delta_w)$, and renaming the entries in δ_v^{tab} and δ_w^{tab} :



UNION(σ, γ) then remaps the entries of δ_{vw}^{tab} to 0 or 1 and reduces the FM if necessary:



This indeed gives us a new canonical FM δ that represents the union of the assignments from σ and γ : $A(\delta) = \{ \{v \mapsto 1, w \mapsto 1\}, \{v \mapsto 1, w \mapsto 3\}, \{v \mapsto 2, w \mapsto 3\}, \{v \mapsto 0, w \mapsto 1\}, \{v \mapsto 3, w \mapsto 1\}, \{v \mapsto 1, w \mapsto 2\} \}$.

4.3.2 Intersection

To create an intersection of two Factored Mappings σ and γ into one FM δ we can use a similar approach as for creating a union. We can use the same combine function as for the union to get an FM that keeps all information about the assignments by combining $A(\sigma)$ and $A(\gamma)$. The root table δ^{tab} will still have entries with 2-dimensional values (x, y) , where x stores if the assignment was in σ and y if it was in γ . The difference between the intersection and the union comes in the remapping of those entries. The 2-dimensional values (x, y) , where x stores if the assignment was in σ and y if it was in γ , have to be mapped differently than before. For the intersection, we want $A(\delta)$ to be equal to $A(\sigma) \cap A(\gamma)$. Therefore, we map an entry in δ^{tab} to 1 if both of the FMs stored the assignment ($x = 1$ and $y = 1$), or 0 otherwise. This means that δ will be an FM that stores exactly the intersection of σ and γ . Because δ could still map entries that we differentiated before to the same value in the root table and might create new redundancies, we need to reduce δ to ensure that it is canonical and reduced. This reduction can lead to drastically different looking FMs for the union and the intersection, as can be seen in the example on page 22.

Algorithm 8 Creating δ as the intersection of two FMs σ and γ

```

1: function INTERSECTION( $\sigma, \gamma$ )
2:    $\delta \leftarrow$  COMBINE( $\sigma, \gamma$ ) ▷ see Algorithm 7
3:   for  $entry \in \delta^{tab}$  with value  $(x, y)$  do
4:     if  $x = 1$  and  $y = 1$  then
5:        $entry \leftarrow 1$ 
6:     else
7:        $entry \leftarrow 0$ 
8:   REDUCE( $\delta$ ) ▷ see Algorithm 1
9:   return  $\delta$ 

```

Time complexity of INTERSECTION(σ, γ) We can see that the time complexity of INTERSECTION(σ, γ) is the same as for UNION(σ, γ), since they only differ in their mapping of the root table. Therefore, INTERSECTION(σ, γ) also has the time complexity of $\mathcal{O}(n \cdot T_\sigma^{max} \cdot T_\gamma^{max})$, where $n = |V|$ and T_σ^{max} , and T_γ^{max} is the maximum over all table sizes of σ and γ respectively.

Example Let's look at the same example that we used for the union again. Let $V = \{v, w\}$ be a variable set, and $dom(v) = dom(w) = \{0, 1, 2, 3\}$ be the domains of the variables. Let σ be the FM for the assignments $A(\sigma) = \{\{v \mapsto 1, w \mapsto 1\}, \{v \mapsto 1, w \mapsto 3\}, \{v \mapsto 2, w \mapsto 3\}\}$. Let γ be the FM for the assignments $A(\gamma) = \{\{v \mapsto 0, w \mapsto 1\}, \{v \mapsto 3, w \mapsto 1\}, \{v \mapsto 0, w \mapsto 2\}, \{v \mapsto 3, w \mapsto 2\}, \{v \mapsto 1, w \mapsto 2\}\}$. We will now compute the intersection of σ and γ into a new FM δ :

δ_{vw}^{tab}	(0,0)	(1,1)	(0,2)	(2,0)
(0,0)	(0,0)	(0,1)	(0,1)	(0,0)
(1,1)	(0,0)	(1,0)	(0,1)	(1,0)
(2,2)	(0,0)	(0,0)	(0,0)	(1,0)

δ_v^{tab}	0	1	2	3
-	0	1	2	0

δ_w^{tab}	0	1	2	3
-	0	1	2	3

This is still the same δ that was created by COMBINE(σ, γ) as in the union. But INTERSECTION(σ, γ) will now remap the entries in δ_{vw}^{tab} differently than the union did:

δ_{vw}^{tab}	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0

δ_v^{tab}	0	1	2	3
-	0	0	0	0

δ_w^{tab}	0	1	2	3
-	0	0	0	0

Lastly, REDUCE(δ) will give us the intersection δ , which is the empty FM because σ and γ do not share any assignments: $A(\delta) = \{\}$.

4.3.3 Complement

Creating the complement $\bar{\sigma}$ of a Factored Mapping σ is pretty simple. We want $A(\bar{\sigma})$ to be equal to $\overline{A(\sigma)}$, where $\overline{A(\sigma)} = A(V) \setminus A(\sigma)$. For this, we need $\bar{\sigma}$ to store all assignments that are not stored in σ . To achieve this, we only have to change the root table and swap all zeroes and ones. Since this swap does not add any new redundancies, $\bar{\sigma}$ will still be canonical and reduced, and store exactly the complement of σ .

Algorithm 9 Creating the complement of an FM σ

```

1: function COMPLEMENT( $\sigma$ )
2:   for  $entry \in \sigma^{tab}$  do
3:     if  $entry = 0$  then
4:        $entry \leftarrow 1$ 
5:     else
6:        $entry \leftarrow 0$ 
7:   return  $\sigma$ 

```

Time complexity of COMPLEMENT(σ) COMPLEMENT(σ) only goes through all entries in the root table once. Its time complexity is given by $\mathcal{O}(|\sigma^{tab}|) = \mathcal{O}(T^{max})$, where T^{max} is the maximum over all table sizes of σ .

4.3.4 Difference

Creating a Factored Mapping δ as the set difference of two FMs σ and γ requires $A(\delta)$ to be equal to $A(\sigma) \setminus A(\gamma)$. Because

$$A(\sigma) \setminus A(\gamma) = (A(\sigma) \cap A(V)) \setminus A(\gamma) = A(\sigma) \cap (A(V) \setminus A(\gamma)) = A(\sigma) \cap \overline{A(\gamma)} = A(\sigma) \cap A(\bar{\gamma}),^5$$

we can create δ by using the already known intersection and complement operations.

Algorithm 10 Creating δ as the set difference of two FMs σ and γ

```

1: function DIFFERENCE( $\sigma, \gamma$ )
2:    $\bar{\gamma} \leftarrow$  COMPLEMENT( $\gamma$ ) ▷ see Algorithm 9
3:    $\delta \leftarrow$  INTERSECTION( $\sigma, \bar{\gamma}$ ) ▷ see Algorithm 8
4:   return  $\delta$ 

```

COMPLEMENT(γ) returns the FM $\bar{\gamma}$ with $A(\bar{\gamma}) = \overline{A(\gamma)}$. INTERSECTION($\sigma, \bar{\gamma}$) returns the FM δ with $A(\delta) = A(\sigma) \cap A(\bar{\gamma}) = A(\sigma) \setminus A(\gamma)$. Because our operations return canonical and reduced FMs δ is a canonical and reduced FM that stores exactly the set difference of σ and γ .

Time complexity of DIFFERENCE(σ, γ) COMPLEMENT(γ) has time complexity $\mathcal{O}(T_\gamma^{max})$ and INTERSECTION($\sigma, \bar{\gamma}$) has time complexity $\mathcal{O}(n \cdot T_\sigma^{max} \cdot T_\gamma^{max})$, where $n = |V|$ and T_σ^{max} , and T_γ^{max} is the maximum over all table sizes of σ and γ respectively. This leads to the time complexity of DIFFERENCE(σ, γ) of $\mathcal{O}(T_\gamma^{max} + n \cdot T_\sigma^{max} \cdot T_\gamma^{max}) = \mathcal{O}(n \cdot T_\sigma^{max} \cdot T_\gamma^{max})$.

⁵ Follows from the associative property of the intersection of sets.

5

Symbolic Search

To consider Factored Mappings as a knowledge compilation for symbolic search we will try to find a reasonable symbolic search algorithm that uses FMs as a knowledge compilation. For this, we will closely follow the algorithm from chapter B8 of the Planning and Optimization lecture by Helmert and Röger (2020) [4]. We will call this breadth-first progression search algorithm for a propositional planning tasks $\langle V, I, O, \gamma \rangle$ $\text{BFSPROGRESSION}(V, I, O, \gamma)$.

Algorithm 11 Progression Breadth-first Search

```
1: function  $\text{BFSPROGRESSION}(V, I, O, \gamma)$ 
2:    $goalStates \leftarrow \text{MODELS}(\gamma)$ 
3:    $reached_0 \leftarrow \{I\}$ 
4:    $i \leftarrow 0$ 
5:   loop
6:     if  $reached_i \cap goalStates \neq \emptyset$  then
7:       return solution found
8:      $reached_{i+1} \leftarrow reached_i \cup \text{APPLY}(reached_i, O)$ 
9:     if  $reached_{i+1} = reached_i$  then
10:      return no solution exists
11:     $i \leftarrow i + 1$ 
```

In the lecture this algorithm uses binary decision diagrams (BDDs) as a knowledge compilation to represent sets of states S . To be a reasonable algorithm it requires that the operations $\text{MODELS}(\gamma)$, $\{I\}$, \cap , $\neq \emptyset$, \cup , $\text{APPLY}(reached_i, O)$, and $=$ are efficiently implemented for BDDs, which means that their time complexity has to be polynomial. We want to see how we can use FMs as a knowledge compilation for this algorithm in a similar way. An FM σ will represent the set of states S as a set of assignments $A(\sigma)$. Since variables in FMs can have any finite domain, we can use the same $\text{BFSPROGRESSION}(V, I, O, \gamma)$ approach for planning tasks in finite domain representation (FDR) as well.

BFSPROGRESSION The algorithm saves the initial state in $reached$, which will be an FM representing the set of states S that are reachable from our initial state I . Then it appends all states that can be reached from $reached$ using the operations O to $reached$. This is done by having $reached_{i+1}$ be the union of $reached_i$, and $\text{APPLY}(reached_i, O)$, which is an FM representing all newly reachable states when applying any operation from O . This is

repeated until the goal state is reachable, or we have no new states that can be reached. In $\text{APPLY}(\text{reached}_i, O)$, we will need to somehow store transitions $s \xrightarrow{O} s'$. We will store these transitions by storing their state pairs $\langle s, s' \rangle$ inside of an FM. To do this, we are going to need two FM variables v and v' for every state variable $v \in V$. Since we always assume the same variable set and merge tree of FMs for the operations, all FMs inside this algorithm will have to be over the variable set \tilde{V} . Where $\tilde{V} = V \cup V'$ and every $v' \in V'$ is a primed copy of v with the same domain $\text{dom}(v) = \text{dom}(v')$. The fixed merge tree $\mathcal{T}(\tilde{V})$ of the FMs ensures that any FM σ over \tilde{V} has a left subtree σ_L only over V , and a right subtree σ_R only over V' . This merge tree structure is chosen like this so we will always have an empty FM over V' as the right subtree σ_R outside of $\text{APPLY}(\text{reached}_i, O)$, where we only need the state variables from V and do not care about the variables from V' used for the state pairs $\langle s, s' \rangle$.

5.1 Converting Formulas into Factored Mappings

5.1.1 Formula

All of the already visited operations allow us to convert propositional formulas ϕ into Factored Mappings σ , representing the models of ϕ . The computation from propositional formulas to FMs is called $\text{FORMULA}(\phi)$. A list of individual logic connectives for which we can use FM operations follows. Let $n = |V|$ be the number of variables, $D^{\max} = \max_{v \in V} |\text{dom}(v)|$ be the maximum over the domains, and T_σ^{\max} and T_γ^{\max} be the maximum over all table sizes of σ and γ respectively:

- For \perp we can use $\text{FALSE}(V)$: $\mathcal{O}(n \cdot D^{\max})$
- For \top we can use $\text{TRUE}(V)$: $\mathcal{O}(n \cdot D^{\max})$
- For v we can use $\text{ATOM}(V, v, c)$: $\mathcal{O}(n \cdot D^{\max})$
- For $I \models \phi?$ we can use $\text{INCLUDES}(\sigma, \alpha)$: $\mathcal{O}(n)$
- For $\phi \equiv \psi?$ we can use $\text{EQUALS}(\sigma, \gamma)$: $\mathcal{O}(n(T_\sigma^{\max} + T_\gamma^{\max}))$
- For $(\phi \vee \psi)$ we can use $\text{UNION}(\sigma, \gamma)$: $\mathcal{O}(n \cdot T_\sigma^{\max} \cdot T_\gamma^{\max})$
- For $(\phi \wedge \psi)$ we can use $\text{INTERSECTION}(\sigma, \gamma)$: $\mathcal{O}(n \cdot T_\sigma^{\max} \cdot T_\gamma^{\max})$
- For $\neg\phi$ we can use $\text{COMPLEMENT}(\sigma)$: $\mathcal{O}(T_\sigma^{\max})$
- For $(\phi \wedge \neg\psi)$ we can use $\text{DIFFERENCE}(\sigma, \gamma)$: $\mathcal{O}(n \cdot T_\sigma^{\max} \cdot T_\gamma^{\max})$
- For $(\phi \rightarrow \psi)$ we can use $\text{UNION}(\text{COMPLEMENT}(\sigma), \gamma)$: $\mathcal{O}(n \cdot T_\sigma^{\max} \cdot T_\gamma^{\max})$
- For $(\phi \leftrightarrow \psi)$ we can use $\text{UNION}(\text{INTERSECTION}(\sigma, \gamma), \text{INTERSECTION}(\text{COMPLEMENT}(\sigma), \text{COMPLEMENT}(\gamma)))$: $\mathcal{O}(n \cdot T_\sigma^{\max} \cdot T_\gamma^{\max})$

Since all our operations take polynomial time, each individual logic connective also takes polynomial time. In the Planning and Optimization lecture [4], it is discussed that converting

a full formula of length m into BDDs can take $\mathcal{O}(2^m)$ time. The same is true for FMs, because converting a formula with m nested connectives can lead to m multiplications of polynomial time, which in the worst case results in the exponential time complexity of $\mathcal{O}(2^m)$ for $\text{FORMULA}(\phi)$.

5.1.2 Singleton

We can convert the single assignment I into a Factored Mapping σ representing $\{I\}$. We do this by computing the conjunction of all literals in I . We use $\text{INTERSECTION}()$ to conjunct all atoms for the partial assignments $\alpha : v \mapsto c$ with $\alpha \in I$, which we can create using $\text{ATOM}(V, v, c)$. We call this computation $\text{SINGLETON}(I)$. I has a maximum of n literals, where $n = |V|$. Therefore $\text{SINGLETON}(I)$ will maximally create $n - 1$ intersections between n created atoms. This takes $\mathcal{O}((n - 1) \cdot n \cdot T_i^{\max} \cdot T_{i+1}^{\max} + n \cdot n \cdot D^{\max}) = \mathcal{O}(n^2 \cdot T_i^{\max} \cdot T_{i+1}^{\max} + n^2 \cdot D^{\max})$ time, where $D^{\max} = \max_{v \in V} |\text{dom}(v)|$ and T_i^{\max} is the maximum over all table sizes of an atoms i . Because $T_i^{\max} \geq D^{\max}$, we can see that $\text{SINGLETON}(I)$ has the time complexity of $\mathcal{O}(n^2 \cdot T_i^{\max} \cdot T_{i+1}^{\max})$.

5.2 The Apply Function

Lastly, we need the $\text{APPLY}(\text{reached}, O)$ operation that computes the set of states (as a Factored Mapping) that can be reached by applying some operator $o \in O$ in some state $s \in \text{reached}$, where O is the set of operators given by the planning task and reached is an FM representing a set of states. First, we need a way to store all possible transitions from our planning task in an FM. As in the Planning and Optimization lecture [4], we let $\tau_V(o)$ be the formula that describes all transitions $s \xrightarrow{o} s'$ of a single operator $o \in O$ in terms of the variables V describing s and V' describing s' . Then, we can create the formula $\bigvee_{o \in O} \tau_V(o)$ describing all state transitions of any operator in O . This formula can be converted to an FM $T_V(O)$ over \tilde{V} with $\text{FORMULA}(\bigvee_{o \in O} \tau_V(o))$. This conversion can take exponential time (as seen in Section 5.1.1), but only needs to be computed once per planning task. $T_V(O)$ is called the transition relation of the planning task and can be computed before calling the apply function. Since $\text{APPLY}(\text{reached}, O)$ is called inside a loop of Algorithm 11, it makes sense to compute the transition relation $T_V(O)$ outside of the loop and pass it as an input to the apply operation instead of O . We will therefore call $\text{APPLY}(\text{reached}, T_V(O))$ instead, which will then apply the transition relation to our reached states to see to which states we could transition to. Since these states will be stored inside the right subtree over V' , we need to reorder and rename them to be stored inside the left subtree over V , so we can use them in our new reached states as starting point for the next iteration. This is different to BDDs, where the lecture uses a RENAME and FORGET operation to achieve this. But since we want FMs to be always over the same variable set \tilde{V} , we need a different operation REORDER .

5.2.1 Reordering and Renaming

To reorder and rename a Factored Mapping σ over \tilde{V} with right subtree σ_R over V' and left subtree σ_L over V , we want to have the old primed variables as unprimed variables inside the left subtree σ_L and the right subtree σ_R over the primed variables as an empty FM. This way, FMs σ over \tilde{V} will always have σ_R as empty FM outside of the $\text{APPLY}(\text{reached}, T_V(O))$ operation. To do this, we use the algorithm $\text{REORDER}(\sigma)$, which sets the left subtree of σ as the old right subtree, renames every $v' \in V'$ to $v \in V$, and then sets the right subtree as empty FM over V' .

Algorithm 12 Reordering and renaming of σ over \tilde{V}

```

1: function REORDER( $\sigma$ )
2:    $\sigma_L \leftarrow \sigma_R$ 
3:   for atomic FM  $\sigma_{v'} \in \sigma_L$  with  $\text{vars}(\sigma_{v'}) = \{v'\}$  do ▷ where  $v' \in V'$ 
4:      $\text{vars}(\sigma_{v'}) \leftarrow \{v\}$  ▷ where  $v \in V$  is the unprimed version of  $v'$ 
5:    $\sigma_R \leftarrow \text{FALSE}(V')$  ▷ see Algorithm 2
6:   return  $\sigma$ 

```

Time complexity of REORDER(σ) $\text{REORDER}(\sigma)$ goes through all $n' = |V'|$ atomic FMs in the new σ_L and changes their associated variable. Then, it calls $\text{FALSE}(V')$ which has a time complexity of $\mathcal{O}(n' \cdot D^{\max})$, where $D^{\max} = \max_{v' \in V'} |\text{dom}(v')| = \max_{v \in V} |\text{dom}(v)|$. Because $n = |V| = |V'| = n$, $\text{REORDER}(\sigma)$ has the time complexity of $\mathcal{O}(n + n \cdot D^{\max}) = \mathcal{O}(n \cdot D^{\max})$.

Now, we can compute the apply operation using $\text{APPLY}(\text{reached}, T_V(O))$, which takes the transition relation, describing state pairs, and conjuncts them with the set of states that are already reached. This gives us an FM σ storing state pairs $\langle s, s' \rangle$, where s' is a successor of s and $s \in \text{reached}$. The states s are stored in σ_L over V and the states s' in σ_R over V' . Then, we reorder σ so its right subtree over V' is an empty FM and the states s' are now stored in terms of variables $v \in V$.

Algorithm 13 Computes the set of successors of reached using the transition relation $T_V(O)$

```

1: function APPLY( $\text{reached}, T_V(O)$ )
2:    $\sigma \leftarrow T_V(O)$ 
3:    $\sigma \leftarrow \text{INTERSECTION}(\sigma, \text{reached})$  ▷ see Algorithm 8
4:    $\sigma \leftarrow \text{REORDER}(\sigma)$  ▷ see Algorithm 12
5:   return  $\sigma$ 

```

Time complexity of APPLY($\text{reached}, T_V(O)$) $\text{APPLY}(\text{reached}, T_V(O))$ calls $\text{INTERSECTION}(\sigma, \text{reached})$ and $\text{REORDER}(\sigma)$ once. This takes $\mathcal{O}(n \cdot T_\sigma^{\max} \cdot T_{\text{reached}}^{\max} + n \cdot D^{\max})$ time, where $D^{\max} = \max_{v \in V} |\text{dom}(v)|$, and T_σ^{\max} and $T_{\text{reached}}^{\max}$ is the maximum over all table sizes of σ and reached respectively. Because $T_\sigma^{\max} \geq D^{\max}$, we can see that $\text{APPLY}(\text{reached}, T_V(O))$ has the time complexity of $\mathcal{O}(n \cdot T_\sigma^{\max} \cdot T_{\text{reached}}^{\max})$.

5.3 Symbolic Search Algorithm for Factored Mappings

Now, we have all the tools for a breadth-first progression search algorithm using Factored Mappings as a knowledge compilation. We will revisit $\text{BFSPROGRESSION}(V, I, O, \gamma)$ and use the implemented FM operations (just like the BDD operations in the Planning and Optimization lecture [4]) to create $\text{BFSPROGFINAL}(V, I, O, \gamma)$. $\text{BFSPROGFINAL}(V, I, O, \gamma)$ uses FMs as knowledge compilation and can thus be used not only for propositional planning tasks, but also for any FDR planning task $\langle V, I, O, \gamma \rangle$.

Algorithm 14 Progression breadth-first search for a FDR planning task using FMs

```

1: function BFSPROGFINAL( $V, I, O, \gamma$ )
2:    $T_V(O) \leftarrow \text{FORMULA}(\bigvee_{o \in O} \tau_V(o))$  ▷ see Section 5.1.1
3:    $goalStates \leftarrow \text{FORMULA}(\gamma)$  ▷ see Section 5.1.1
4:    $reached_0 \leftarrow \text{SINGLETON}(I)$  ▷ see Section 5.1.2
5:    $i \leftarrow 0$ 
6:   loop
7:     if EQUALS( $\text{INTERSECTION}(reached_i, goalStates), \text{FALSE}(\tilde{V})$ ) = False then
8:       return solution found ▷ see Algorithms 8,2 and 6
9:        $reached_{i+1} \leftarrow \text{UNION}(reached_i, \text{APPLY}(reached_i, T_V(O)))$  ▷ see Algorithms 7, 13
10:      if EQUALS( $reached_{i+1}, reached_i$ ) then ▷ see Algorithm 6
11:        return no solution exists
12:       $i \leftarrow i + 1$ 

```

All necessary operations are implemented efficiently with a polynomial time complexity. The only operations with a potentially exponential time complexity are $\text{FORMULA}(\bigvee_{o \in O} \tau_V(o))$ on line 2, $\text{FORMULA}(\gamma)$ on line 3, and $\text{SINGLETON}(I)$ on line 4. Because both FORMULA calls and the $\text{SINGLETON}(I)$ call only need to be computed once per planning task, this is a reasonable symbolic search algorithm for a propositional or FDR planning task. This means that Factored Mappings are well suited as a knowledge compilation for symbolic search.

6

Conclusion

In this thesis, we investigated Factored Mappings as a knowledge compilation language. We have seen that it is possible to use Factored Mappings as a knowledge compilation language for symbolic search.

We have done this by first looking for a canonical representation and have found that there are two canonical representations assuming a fixed merge tree and value order. The first one stores a maximum amount of different values and the second one stores a minimal amount.

We decided to use the latter for obvious space saving reasons.

We then looked at different operations for creating, querying, and transforming FMs. We saw that all of those operations could be performed in polynomial time.

Then we looked the missing operations that are necessary for a breadth-first symbolic search algorithm. We saw that converting a formula into an FM could take exponential time, but the only operations inside a symbolic search algorithm that have to convert a formula to an FM only get called once per planning task. Therefore they do not affect the breadth-first search much, which means that we have found a reasonable symbolic search algorithm for FMs.

This shows that Factored Mappings are a knowledge compilation language that is well suited for symbolic search algorithms for propositional and finite-domain planning tasks.

Bibliography

- [1] Randal E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. In Hillel Ofek and Lawrence A. O’Neill, editors, *Proceedings of the 22nd ACM/IEEE Design Automation Conference (DAC 1985)*, pages 688–694. Association for Computing Machinery, 1985.
- [2] Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In Toby Walsh, editor, *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI 2011)*, pages 819–826. AAAI Press, 2011.
- [3] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.
- [4] Malte Helmert and Gabriele Röger. Lecture: Planning and optimization. University of Basel, Fall Semester 2020. URL <https://dmi.unibas.ch/de/studium/computer-science-informatik/lehreangebot-hs20/lecture-planning-and-optimization/>.
- [5] Malte Helmert, Gabriele Röger, and Silvan Sievers. On the expressive power of non-linear merge-and-shrink representations. In Ronen Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein, editors, *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, pages 106–114. AAAI Press, 2015.
- [6] Silvan Sievers and Malte Helmert. Merge-and-shrink: A compositional theory of transformations of factored transition systems. Accepted for publication in *Journal of Artificial Intelligence Research*.

Declaration on Scientific Integrity

Erklärung zur wissenschaftlichen Redlichkeit

includes Declaration on Plagiarism and Fraud
beinhaltet Erklärung zu Plagiat und Betrug

Author — Autor

Leonhard Badenberg

Matriculation number — Matrikelnummer

2016-055-238

Title of work — Titel der Arbeit

Factored Mappings as Knowledge Compilation for Symbolic Search

Type of work — Typ der Arbeit

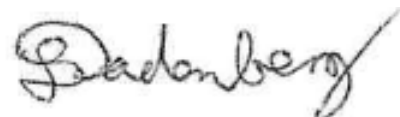
Bachelor Thesis

Declaration — Erklärung

I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe. Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Basel, June 7, 2021



Signature — Unterschrift