Converting Truth Tables to Minimized Boolean Functions

Toni Nguyen

A thesis presented for the degree of
Bachelor of Science

Algorithmic Bioinformatics
Heinrich Heine University Düsseldorf
Germany

14th October, 2022

Acknowledgments

I would like to thank Prof. Gunnar Klau for his help with finding this interesting and appealing topic and for making it possible to write this bachelor's thesis in the first place. I owe a special thanks to Eline van Mantgem for her guidance and patience throughout this work.

Abstract

This bachelor's thesis is about the improvement of the interpretability of gene regulatory networks. For this, gene regulatory networks can be displayed as Boolean networks in which case the values are normalized and discretized to Boolean values. To make them more readable, those Boolean networks can be converted to truth tables which on the other hand can be rewritten as Boolean functions. We look at different algorithms to minimize those Boolean functions, implement those algorithms in Python, and then test and compare them to each other.

The result is that the algorithms improve non-minimized Boolean functions regarding readability and memory usage, but are very similar among themselves.

Contents

1 Introduction

1.1 Background and Related Work

This bachelor's thesis is based on Systems Biology. Systems Biology can be summarized as a field that handles the analysis of complex interactions in biological systems. Two important concepts originate from it. The holistic and reductionistic approach. Holistic refers to the idea that complex systems cannot be fully understood by studying single modules. Reductionistic on the other hand turn to the suggestion that bigger systems need to be divided into multiple modules for better analysis. These two approaches seem opposing, but are combined in the field of Systems Biomedicine. There, the main goal is to apply mechanistic information to clinical applications and to improve the treatment for patients. A commonly used research concept is the repeating steps of theoretical analysis, computational modelling, and experimental validation of model hypotheses (Antony, Balling, and Vlassis 2012). For this work, we take a closer look at the computational modelling of complex systems regarding gene regulatory networks (GRN). GRN are considered complex control systems that consist of many thousand modular DNA sequences. These modules receive regulatory proteins and recognize specific sequences within them that lead to the precise transcription control of associated genes (Davidson and Levin 2005). Since these networks are very complex, we search for different modelling approaches that allow us to better access their portrayed information. One of these modellings was developed by Julio Saez-Rodriguez et al. in which they used the cell network optimizer (CNO) software to turn signalling networks into logical models and calibrate them against experimental data (Saez-Rodriguez et al. 2009). A different modelling approach was developed by Roded Sharon and Richard M. Karp in which they learn Boolean models automatically from data. To do that, they created an Integer Linear Program that minimized the sum over all experiments of the number of experimental observations that differ from the prediction of the model (Sharan and Karp 2013). The output that these approaches produce is still complex and hard to interpret. This is why the motivation for this bachelor's thesis is to work on methods to simplify these outputs and allow them to be more interpretable. For Sharon and Karp's approach, the interpretability of the output can be improved by minimizing the resulting Boolean functions derived from the logical models. Multiple algorithms deal with the topic of minimizing Boolean functions. The most famous are the Karnaugh-Veitch algorithm and the Quine-McCluskey algorithm. While the Karnaugh-Veitch algorithm approaches the problem with a grid setup, in which you look for matching cells to combine, the Quine-McCluskey algorithm relies on a more algebraic solution, that looks for complementary terms that can be deleted, to form the function. Lastly, there is one more algorithm called Petrick's Method, which works with the row numbers of minterms and uses Boolean algebra to find the minimized function. The Karnaugh-Veitch and Quine-McCluskey algorithms are both implemented on multiple websites and there is a working Java implementation that was developed with the Quine-McCluskey algorithm (*Minimizing Boolean Functions* 2021).

1.2 Own Contribution

The main goal is to increase the interpretability of given truth tables, which can be achieved by generating legible Boolean functions and minimizing them. Usually, minimized functions are half the size of their non-minimized version, which is shown in the Preliminary section. To have terms with the same purpose, but a smaller function size increases the time efficiency because the functions that need to be worked with are smaller. Furthermore, it increases memory efficiency as smaller functions are stored. To explore this further, I want to work with the different algorithms that deal with truth table minimization in more detail and compare them concerning memory usage and runtime. This extends the work to finding special cases the algorithms struggle with and types of input that take the longest to be calculated.

2 Preliminary Knowledge

To understand this thesis, some technical terms first need to be introduced. The most important concept is truth tables because they are the main source of data input used in this thesis. What this work wants to accomplish is to implement the best possible optimizer for truth tables. Truth tables are represented as a table of all the combinations of values for inputs and their matching outputs. These inputs are called variables and are binary, which means that they can only be *one* or *zero*. In the first row of the truth table, the different variables are split up into individual columns. The entries for one column represent the state of the variable of that column, which means that every cell in a truth table displays the state for one variable. The last column illustrates the output of a Boolean function that is built on the before-mentioned variables. The output of this function depends on the combination of *zeros* and *ones* in the same row, which is why rows are considered as groups. Every Row represents a unique combination of states for the variables, which is why every row is different. The phrases minterm and function output are often mentioned in this work. Minterm refers to a row in a truth table, where the function column contains a *one* and the function output represent the values in this function column. To go into further detail about Boolean functions, they are constructed by connecting variables with different logical operators. All the different variables are called literals, and their negation is symbolized by a $\neg$ in front of the literal. These literals are connected with a logical AND which can be symbolized as $\wedge$ or $*$ or logical OR, which are symbolized by $\vee$ or $+$. Figure 1 shows an example truth table with a function output of 1011 1110. This truth table is represented by Figure 2, which is its non-minimized function and Figure 3 which is the minimized version.
To prove, the before mentioned claim, that minimized functions are half the size of their non-minimized version, my idea was to calculate the average size of the non-minimized Boolean function and minimized Boolean functions for a fitting amount of samples and compare them. This idea is shown in Algorithm 1. For this pseudocode, DATA is a list of all possible combinations for function output. If we take a truth table with three variables then DATA would

A	B	C	$Function$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Figure 1: *Example truth table*

$$f = \neg A \neg B \neg C \lor \neg AB \neg C \lor \neg ABC \lor A \neg BC \lor AB \neg C \lor AB \neg C$$

Figure 2: *Non-Minimized Boolean Function*

$$f_{min} = B \neg C \lor \neg B \land C \lor \neg A$$

Figure 3: *Minimized Boolean Function*

range from 0000 0000 to 1111 1111 which is why it would have a size of 2^8 since there are 2^8 possible combination to arrange the *zeros* and *ones*. What the algorithm does is calculate the average length of non-minimized truth tables by using the property that a normal Boolean function can be calculated by combining all minterms from a truth table. To do that Algorithm 1 goes through DATA and sums up all occurrences of *one* and in the end divides them by the number of combination possibilities, which is the length of DATA. This code can be modified, so it can calculate the average length for minimized Boolean functions as well, which is possible by changing Algorithm 1 in line 5. DATA[i].count(1) needs to be replaced by DATA[i] being inputted into a minimization algorithm. If we run both codes for three variables, then the average number of terms in the non-minimized Boolean function is 4, whereas it is 2.322 for minimized functions. This bisection is also the case for four variables, where the average number of terms is 8 for non-minimized functions and 4.133 for minimized ones.

Algorithm 1 Calculate Average Function Length

```
1: function CALCAVGFUNCTIONLENGTH(DATA)
2:     $sum \leftarrow 0.$
3:     $len \leftarrow length(DATA).$
4:     for i = 0; i < len(data; i++) do
5:         $sum \leftarrow sum + DATA[i].count(1)$
6:     end for
7:     return $sum/length$
8: end function
```

In this work, we need to know about two specific types of normal-form representations. Firstly the Disjunctive Normal Form representation (DNF), where the literals are separated into different terms which are signified by brackets. The literals in the terms are connected by logical AND and those terms are then connected among themselves by logical OR. Secondly, there is the Conjunctive Normal Form (CNF) representation, which is the opposite of the DNF in regard to the fact that the literals in the terms are connected by logical OR, and the terms are tied together by logical AND.

3 Methods

This section describes all algorithms that we use in the implementation, how they work, and the limitations they have.

3.1 Karnaugh-Veitch-Algorithm

The Karnaugh-Veitch algorithm was first designed in 1952 by Edward W. Veitch and later in 1953 expanded by Maurice Karnaugh (Karnaugh 1953). The idea is to build a diagram that depicts a given truth table and use it to find connected groups that can be summarized and compressed. The structure of this diagram resembles the structure of a matrix, which has multiple cells that can be accessed by the right row and column number. All rows and columns represent either a literal or its negation so that all combinations of literals can be covered in the diagram. Figure 4 displays how a truth table is transitioned into a Karnaugh-Veitch-Diagram for three variables, and Figure 5 shows it for four variables. Every function output is represented by one cell entry, which can be taken over from the truth table to the diagram and is shown in the figures as the numbers in the last column. The number of entries in a Karnaugh-Veitch-Diagram is always 2^n, where n is the number of variables in the given truth table. There is a small peculiarity in Figure 4 and Figure 5. For Figure 5 the last two columns are swapped and for Figure 5 the last two columns, as well as the last 2 rows, are swapped. This was done so that the same literals or their negations are beside each other and are not alternating. In Figure 4 an example is that D in the second and third columns are next to each other and that the $\neg D$ columns are next to each other if you do not consider the edges. This pairing is important in later parts of the algorithm.

A	B	C	$Function$
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

	$\neg B, \neg C$	$\neg B, C$	B, C	$B, \neg C$
$\neg A$	0	1	3	2
A	4	5	7	6

Figure 4: Transition from Truth Table to Karnaugh-Veitch-Diagram for 3 Literals

A	B	C	D	$Function$
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

	$\neg C, \neg D$	$\neg C, D$	C, D	$C, \neg D$
$\neg A, \neg B$	0	1	3	2
$\neg A, B$	4	5	7	6
A, B	12	13	15	14
$A, \neg B$	8	9	11	10

Figure 5: Transition from Truth Table to Karnaugh-Veitch-Diagram for 4 Literals

After finishing the conversion into the Karnaugh-Veitch-Diagram, groups of *ones* develop. In this algorithm groups are defined as vertically and horizontally adjacent clusters of *ones* but they cannot be connected diagonally. However, there are some rules to the grouping of the *ones*.

(a) The size of the group can only be a multiple of 2

(b) All *ones* need to be in the biggest group they have available

(c) The grouping can exceed the edges of the diagram

(d) Every *one* needs to be in a group.

After all the *ones* are assigned to a group, those groups can be changed to function terms. To do this conversion, it is important which space the group covers in the diagram. Since all rows and columns represent one literal or their negation, it needs to be looked at if the group covers both representations. If it does, the regarded literal can be deleted from the group and If it does not, the literal needs to be incorporated into the final function term. Figure 6 shows an example of how a group in a Karnaugh-Veitch-Diagram looks like. If we want to form a function from this group, we can see that it covers the rows and columns: $A, \neg A, B, \neg B$, and C. Since there are both representations for A and B, referring to the literal and their negation, they can be deleted. As to C, there is no negation for it in the group, which means that it has to be added to the final term. So the group in Figure 6 can be translated to the term C.

	$\neg B, \neg C$	$\neg B, C$	B, C	$B, \neg C$
$\neg A$	0	1	1	0
A	0	1	1	0

Figure 6: Example for Grouping in a Karnaugh-Veitch-Diagram

This needs to be done for all groups and afterward, all incorporated terms are connected with logical OR to form the Disjunctive-Normal-Form representation. Unfortunately, this algorithm works well for variable inputs from two-four, but for the numbers above, the algorithm gets very complex, which is why these cases will not be addressed in this work. Algorithm 2 describes the sequence of the Karnaugh-Veitch algorithm.

Algorithm 2 Karnaugh-Veitch-Algorithm

```
1: function KVALGORITHM(INPUT)
2:     $kvDiagram \leftarrow setupKV(INPUT)$.
3:     $groups \leftarrow grouping(kvDiagram)$.
4:     $function \leftarrow translateToFunction(groups)$.
5:     return $function$
6: end function
```

3.2 Quine-McCluskey-Algorithm

The Quine-McCluskey-Algorithm was first designed in 1952 by Williard V. Quine and later in 1956 extended by Edward J. McCluskey (McCluskey 1956). The concept behind this Algorithm is to generate terms from all minterms of a given truth table and combine them to filter out redundant variables. The algorithm makes use of the complement law $X + \neg X = 1$ and the idempotent law $X * X = X$. In Figure 7 the only minterms are the first two terms, which means

A	B	C	$Function$
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Figure 7: Example for Combining 2 Minterms

that only they are being used for the algorithm. To use the complement law, the minterms first need to be combined into a Boolean function. Figure 7 can be converted into:

$$f = (\neg A \wedge \neg B \wedge \neg C) \wedge (\neg A \wedge \neg B \wedge C)$$

which can then be rewritten into:

$$\begin{aligned}
&(\neg A \wedge \neg B \wedge \neg C) \wedge (\neg A \wedge \neg B \wedge C) \\
\equiv\ &(\neg A \wedge \neg A) \wedge ((\neg B \wedge C) \vee (\neg B \wedge C)) \\
\equiv\ &\neg A((\neg B \wedge \neg B) \wedge (C \vee C)) \\
\equiv\ &\neg A \wedge \neg B \wedge (C \vee \neg C) \\
\equiv\ &\neg A \wedge \neg B
\end{aligned}$$

So the rule for combining two minterms is to examine them and see if they are the same except for one literal. This has to be done for all possible pairings of minterms in the truth table, and they need to be stored in order to keep working with them. Since there is a possibility that terms from the newly created list can also be paired, this process needs to be repeated until there are no combination possibilities anymore. Literals that can be deleted are then replaced by a −. Figures 8 to 11 show what a fully executed combination process looks like. What remains is a list of all the prime implicants, which are the terms that can not be further minimized. To get the final minimized function, the essential prime implicants need to be calculated. Therefore, the list of prime implicants needs to be converted into a prime implicant chart, which is a tabular representation of them. Here, it is important that every combination still has its respective row number from the truth table. Every prime implicant chart contains the modified terms on the first column and all the row numbers that are represented by the modified terms in the first row. The prime implicant chart is now filled by verifying if the row number is in the viewed modified term. Figure 12 displays the prime implicant chart for Figure 11.

	A	B	C	D	$Function$
(0)	0	0	0	0	1
(1)	0	0	0	1	1
(2)	0	0	1	0	1
(3)	0	0	1	1	1
(4)	0	1	0	0	0
(5)	0	1	0	1	0
(6)	0	1	1	0	0
(7)	0	1	1	1	0
(8)	1	0	0	0	0
(9)	1	0	0	1	1
(10)	1	0	1	0	0
(11)	1	0	1	1	1
(12)	1	1	0	0	1
(13)	1	1	0	1	1
(14)	1	1	1	0	1
(15)	1	1	1	1	1

Figure 8: Example Truth Table

	A	B	C	D	$Function$
(0)	0	0	0	0	1
(1)	0	0	0	1	1
(2)	0	0	1	0	1
(3)	0	0	1	1	1
(9)	1	0	0	1	1
(11)	1	0	1	1	1
(12)	1	1	0	0	1
(13)	1	1	0	1	1
(14)	1	1	1	0	1
(15)	1	1	1	1	1

Figure 9: Filtering out all Non-Minterms

	A	B	C	D	$Function$
$(0,1)$	0	0	0	–	1
$(0,2)$	0	0	–	0	1
$(1,3)$	0	0	–	1	1
$(1,9)$	–	0	0	1	1
$(2,3)$	0	0	1	–	1
$(3,11)$	–	0	1	1	1
$(9,11)$	1	0	–	1	1
$(9,13)$	1	–	0	1	1
$(12,13)$	1	1	0	–	1
$(12,14)$	1	1	–	0	1
$(11,15)$	1	–	1	1	1
$(13,15)$	1	1	–	1	1
$(14,15)$	1	1	1	–	1

Figure 10: Truth Table after the First Combining

	A	B	C	D	$Function$
$(0,1,2,3)$	0	0	–	–	1
$(1,3,9,11)$	–	0	–	1	1
$(9,11,13,15)$	1	–	–	1	1
$(12,13,14,15)$	1	1	–	–	1

Figure 11: Truth Table after Second Combining

	A	B	C	D	0	1	2	3	9	11	12	13	14	15
$(0,1,2,3)$	0	0	–	–	X	X	X	X						
$(1,3,9,11)$	–	0	–	1		X		X	X	X				
$(9,11,13,15)$	1	–	–	1					X	X		X		X
$(12,13,14,15)$	1	1	–	–							X	X	X	X

Figure 12: Prime Implicant Chart for Figure 12

All terms that have a unique row number incorporated are considered essential prime implicants that have to be included in the final term. These are highlighted in red in Figure 12. After finding all the essential prime implicants, the next iterations need to be prepared, in which you remove all row numbers that are already included in all the essential prime implicants and filter the remaining non-essential prime implicants. The goal is to cover all row numbers, that were not covered by the found essential prime implicants. The following steps are not documented and were discovered by testing out different approaches and analyzing which delivered the best results. The best technique to filter the remaining prime implicants, was to compare the remaining row numbers and if at least two values of those row numbers are matching with all the row numbers that have to be covered, the prime implicant can be added to the next iteration, otherwise, it needs to be deleted. This step is repeated until there are no row numbers left to cover. For Figure 12 the remaining row numbers are 9 and 11. To cover them both, either the second or third row would need to be included. At last, all extracted essential prime implicants can be converted to a function and connected with a logical OR to form the minimized Boolean function. For Figure 12 the minimized Boolean function is:

$$f_1 = \neg A \land \neg B \lor \neg B \land D \lor A \land B \text{ or } f_2 = \neg A \land \neg B \lor A \land D \lor A \land B$$

In some cases, there are no essential prime implicants left. If there are five or more indices left to cover, you would need to make use of the Petrick's method. This threshold of five was also discovered by testing it with different values. Other than the Karnaugh-Veitch-Algorithm the Quine-McCluskey-Algorithm can also be used for more than four variables. Algorithm 3 describes the structure of the Quine-McCluskey algorithm.

Algorithm 3 Quine-McCluskey-Algorithm

```
function QMALGORITHM(INPUT)
    $minterms \leftarrow filterMinterms(INPUT)$
    $allRowNumbers \leftarrow rowNumbers(minterms)$
    $modified \leftarrow True$

    while modified do
        $tmp \leftarrow []$
        $modified \leftarrow False$
        for i=0;i<minterms.length();i++ do
            if minterms[i].canBeCombined() then
                $tmp.append(combining(minterms))$
                $modified \leftarrow True$
            end if
        end for
        $minterms \leftarrow tmp$
    end while

    while allRowNumber.length() != 0 do
        $newEpi \leftarrow epiThroughChart(minterms)$        ▷ EPI = essential prime implicants
        if newEpi.length() == 0 and allRowNumber > 4 then
            $Petrick'sMethod$
        end if
        $noEpi \leftarrow nonEpiThroughChart(minterms)$
        $allRowNumbers \leftarrow allRowNumbers - allRows(newEpi)$
        for i=0;i<noEpi.length();i++ do
            if noEpi[i] in allRowNumbers twice then
                $newEpi.append(noEpi[i])$
            end if
        end for
        $minterms \leftarrow newEpi$
    end while

    return minterms

end function
```

3.3 Petrick's Method

The Petrick's method is described by Stanley R. Petrick in 1956 (Petrick 1956). The main idea for this algorithm is to use a prime implicant chart to build a Boolean function and simplify it with Boolean algebra. To ease the work with the prime implicants, we substitute them with representative variables. The algorithm then starts by building the function in DNF representation. It goes through every column and connects the representative variables that share the same column with a logical OR. All the terms get connected by logical AND. After generating the function it needs to get simplified to a CNF representation, as all the terms in this form are representative of the truth table and the goal is to find the smallest term to have the optimal minimization. Boolean algebra rules that are needed:

(1.) (distributive law): $X * (X + Y) = X * X + X * Y$

(2.) (idempotent Law): $X * X = X$

(3.) (idempotent Law): $X + X = X$

(4.) (absorbation law): $X + X * Y = X$

Finally, when the CNF representation is built and the smallest term is chosen, it still needs to be resubstituted. Figure 13 shows an example of how the Petrick's method is used for the sample function output 01111110.

	A	B	C		1	2	3	4	5	6
$(1,3)$	1	–	0		X		X			
$(1,5)$	1	0	–		X				X	
$(2,3)$	–	1	0			X	X			
$(2,6)$	0	1	–			X				X
$(4,5)$	–	0	1					X	X	
$(4,6)$	0	–	1					X		X

Figure 13: Prime Implicant Chart with No Essential Prime Implicants

(1) $(1,3) \equiv X_0$

(2) $(1,5) \equiv X_1$

(3) $(2,3) \equiv X_2$

(4) $(2,6) \equiv X_3$

(5) $(4,5) \equiv X_4$

(6) $(4,6) \equiv X_5$

The calculation would look like this:

$$(X_0 \vee X_1)(X_2 \vee X_3)(X_0 \vee X_2)(X_4 \vee X_5)(X_1 \vee X_4)(X_3 \vee X_5)$$

$$\equiv (X_0X_2 \vee X_0X_3 \vee X_1X_2 \vee X_1X_3)(X_0X_4 \vee X_0X_5 \vee X_2X_4 \vee X_2X_5)(X_1X_3 \vee X_1X_5 \vee X_4X_3 \vee X_4X_5)$$

$$\equiv \ldots$$

$$\equiv (X_0X_2X_4X_5 \vee X_0X_3X_4 \vee X_1X_2X_3X_4 \vee X_0X_1X_3X_5 \vee X_1X_2X_5)$$

So you would either take $X_0X_3X_4orX_1X_2X_5$ and after resubstituting it from the prime implicant chart the result would be:

$$f1 = A \wedge \neg C \vee \neg A \wedge B \vee \neg B \wedge C \text{ or } f2 = A \wedge \neg B \vee B \wedge \neg C \vee \neg A \wedge C$$

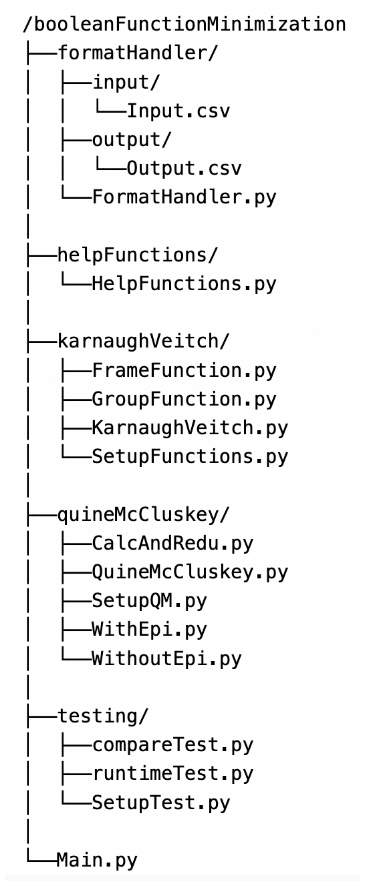
4 Implementation

The code for this implementation can be found on gitlab[1]. Firstly, the idea of the implementation is that input is given to the input folder and after the code runs, the output is placed in a folder called output. This code should be run from the main file, where every column of the CSV file is separated and checked for its number of literals, to determine which algorithm is needed. Two to four literals are solved with Karnaugh-Veitch, while more than four are solved with Quine-McCluskey. I assumed that Karnaugh-Veitch has a better runtime for up to four variables, and since it does not work for more than four variables, Quine-McCluskey is used for these situations. The project is divided into multiple packages and is bound together by the main functions. The structure of the project is shown in Figure 15. The first package that is used when running the application is the formatHandler package. It contains the folder for the input and output and is responsible for converting columns of a CSV table to a usable tuple. This tuple is composed of two elements. Firstly the topology of the nodes and secondly the function output for the matching truth table. This is further explained in the In-/Output section. Figure 14 presents what a tuple would look like.

$$('f <- A, B, C, D', '0000011100001101')$$

Figure 14: Example Tuple created from CSV

[1] *https://gitlab.cs.uni-duesseldorf.de/albi/albi-students/ba-toni-nguyen*



```
/booleanFunctionMinimization
├──formatHandler/
│  ├──input/
│  │  └──Input.csv
│  ├──output/
│  │  └──Output.csv
│  └──FormatHandler.py
│
├──helpFunctions/
│  └──HelpFunctions.py
│
├──karnaughVeitch/
│  ├──FrameFunction.py
│  ├──GroupFunction.py
│  ├──KarnaughVeitch.py
│  └──SetupFunctions.py
│
├──quineMcCluskey/
│  ├──CalcAndRedu.py
│  ├──QuineMcCluskey.py
│  ├──SetupQM.py
│  ├──WithEpi.py
│  └──WithoutEpi.py
│
├──testing/
│  ├──compareTest.py
│  ├──runtimeTest.py
│  └──SetupTest.py
│
└──Main.py
```

Figure 15: Project's package structure

Then, there is the helpFunctions package which assists both algorithms, with basic functionality. This includes sorting and removing duplicate list elements, to convert matrices to strings and functions to help with the grouping in Karnaugh-Veitch. Lastly, the Karnaugh-Veitch and Quine-McCluskey packages incorporate the implementations of the algorithms. These algorithms get tested by the test files in the testing package. Both algorithms return a list, containing strings that represent the terms for the end function. This is displayed in Figure 16. Noticeable is that in the end lists the terms are connected with a '+' which normally presents a logical OR but in this case represents a logical AND.

['C + A' , '-D + C + B' , 'D + C + -B']

Figure 16: Output of Both Algorithms for Figure 15

4.1 In-/output

For input and output, the chosen format is CSV due to its universal use. These CSV tables contain two rows, where the first row describes the topology of the nodes and the second row is the output of a truth table, which is its last column. The format for the topology in the first row is split up into two halves. The first part is the left side of <-, which is the child node, and the right side describes the parent nodes. These topologies are comma separated. The second row on the other hand is just a sequence of *ones* and *zeros* without any separation. For the output, the first row is the same as the input but the second row then displays the minimized boolean function. In this format, negations are represented by ¬, logical OR with ∨, and logical AND with ∧. Essentially every column describes one truth table that needs to be optimized. The main function reads out every column of the CSV file and selects the fitting algorithm for it. Both input and output formats can be seen in Figure 17 and Figure 18.

f_1 <– A,B	f_2 <– A,B,C	f_3 <– A,B,C,D
1100	11101000	1011110111000100

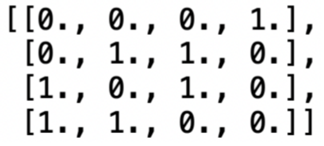
Figure 17: Example CSV Input File

f_1 <– A,B	f_2 <– A,B,C	f_3 <– A,B,C,D
$\neg B$	$\neg B \wedge \neg A \vee \neg C \wedge \neg B \vee \neg C \wedge \neg A$	$\neg B \wedge \neg A \vee \neg C \wedge \neg B \vee \neg C \wedge \neg A \vee \neg C \wedge \neg A$

Figure 18: Example CSV Output File

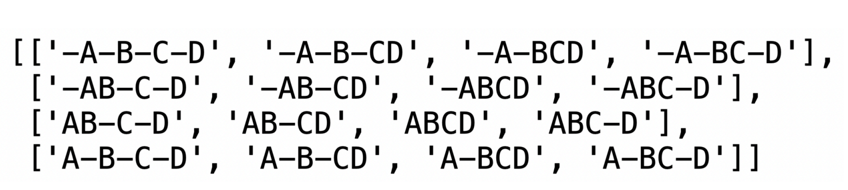
4.2 Implementation of Karnaugh-Veitch

The implementation of the Karnaugh-Veitch-Algorithm is divided into four Python scripts. The most important data structures are two different matrices. One functions as the Karnaugh-Veitch-Diagram, which is implemented as a binary NumPy matrix and is used for all tasks regarding the grouping. The second matrix is used to store which combination of literals is dedicated to which cell of the other matrix. The format that is used to save the combinations of literals is strings. Figures 19 and 20 display examples of the two different types of matrices.



```
[[0., 0., 0., 1.],
 [0., 1., 1., 0.],
 [1., 0., 1., 0.],
 [1., 1., 0., 0.]]
```

Figure 19: Example for KV-diagram



```
[['-A-B-C-D', '-A-B-CD', '-A-BCD', '-A-BC-D'],
 ['-AB-C-D', '-AB-CD', '-ABCD', '-ABC-D'],
 ['AB-C-D', 'AB-CD', 'ABCD', 'ABC-D'],
 ['A-B-C-D', 'A-B-CD', 'A-BCD', 'A-BC-D']]
```

Figure 20: Example for KV-frame

This assembling of matrices was implemented in setupFunctions.py. To construct the strings of literals, FrameFunctions.py is used. There, the fitting frame is created for the given input data.

The idea is to go through the NumPy array in different ways to get all the possible groups that can be built and save them in lists of tuples, where each tuple contains the row and column number for the position of a *one* in the matrix. Since this algorithm is only meant to work for up to 4 variables the maximum size of the matrices is 4x4 which means the only possible size of groups are groups of one, two, four, eight, and 16. The functions vCheck() and hCheck() both run between all rows and columns to find groups of two. For situations with groups bigger than two, there is a function for every group size, that stores all possible groups for that group size. So after collecting all possible groups and saving them in a list, this list needs to be filtered of all duplicate and redundant groups, which happens in the function filterGroups(). It goes through the list of all groups and compares if all the tuples of the considered group are already in different groups. If so, the viewed group can be deleted. All those functions are located in groupFunctions.py. In Figure 21 are three different Karnaugh-Veitch-Diagrams that show all the possible groups collected by the algorithm for a function output of 0000 1010 0000 1010. The end list would contain all groups depicted in (a), (b) and (c) but only (c) would be kept since it represents all other groups shown in (a) and (b).

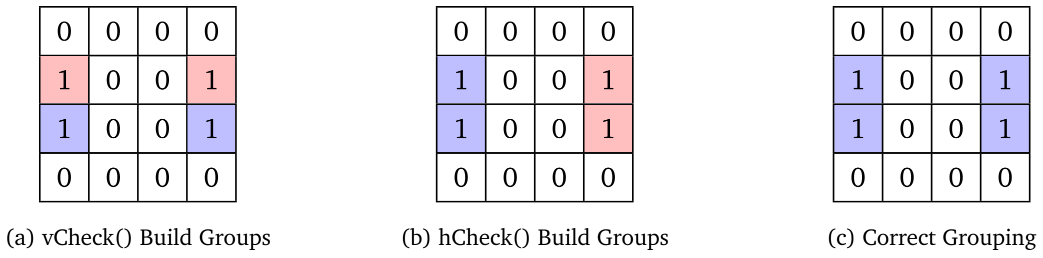

(a) vCheck() Build Groups (b) hCheck() Build Groups (c) Correct Grouping

Figure 21: Example for Filtering Groups

Furthermore, it is important to note that there can be multiple optimal solutions to minimization. Figure 22 shows such a case, for which Karnaugh-Veitch-Algorithm can find two different solutions for the same Karnaugh-Veitch-Diagram.

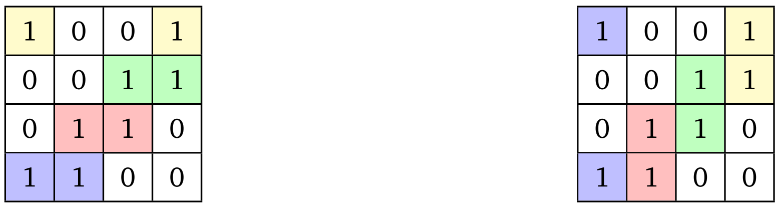


Figure 22: Example for Filtering Groups

As there are some cases where it matters in which the sequence groups are deleted, there is also a function that returns multiple versions of the same list in a different order. This collection of the different ordered lists is put into the filtering function, and the result with the smallest amount of terms is chosen as the output of the algorithm. This is further explained in the Obstacle section. All those steps are done in the KarnaughVeitch package, which also converts the group format to the actual list format that the formatHandler demands. In the case that all output values in the truth tables were *one*, the entire KV-diagram is a group, so all literals can be reduced, which gives us a minimized Boolean function of 1.

4.3 Implementation of Quine-McCluskey

The implementation of the Quine-McCluskey-Algorithm is divided into multiple Python scripts. To start the algorithm, a fitting truth table for the input is generated by using a NumPy array, to find all minterms. These minterms then get stored as strings in a directory with their matching row number, where the minterm is used as a key for the row number. Figure 24 is an example of such a dictionary.

{ ’000’ : ’0’ , ’010’ : ’2’ , ’011’ : ’3’ , ’100’ : ’4’ , ’101’ : ’5’ , ’110’ : ’6’ }

Figure 23: Example of dictionary

After that, every minterm is compared to all other minterms and if a possibility for combination is found, the combination will be saved in a new list. Instead of deleting the literal that is combined, it gets replaced by a -1 for later distinctions between minterms. The initial minterm list gets replaced by the new list, and the process gets repeated until there are no more possible combinations. The last list that is constructed contains all prime implicants. Even though the prime implicants are combinations of minterms, they have to adopt their literals row numbers, which are also kept in a dictionary. The row numbers are stored as strings and are separated with + to make sure that they are more distinguishable and not error prone with two-figure numbers. Since there is a way to reduce the number of terms even more, the prime implicant chart needs to be replicated. This is done by going through the prime implicants and comparing them to the row numbers that need to be covered. To compare them the combined row numbers of the prime implicants are used and if a row number should be unique it is considered an essential prime implicant. We have to repeatedly search for these essential prime implicants. If there are no essential prime implicants and the number of rows that have to be covered is above four, the Petrick’s method is used to solve the remaining part. Figure 24 displays a situation where the minimized function can be found without the Petrick’s method and Figure 25 where Petrick’s method is necessary.

	A	B	C		0	2	3	4	5	6
$(0,2,4,6)$	0	$-$	$-$		X	X		X		X
$(2,3)$	$-$	1	0			X	X			
$(4,5)$	$-$	0	1					X	X	

Figure 24: Prime Implicant Chart with Essential Prime Implicants

There are two different packages to deal with the possibility that at least one essential prime

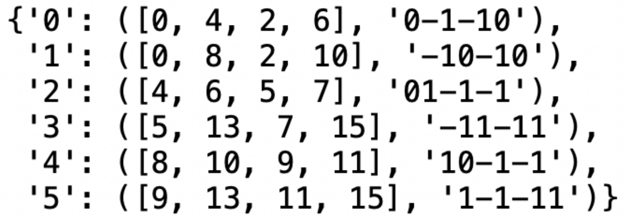
	A	B	C		1	2	3	4	5	6
(1,3)	1	–	0		X		X			
(1,5)	1	0	–		X				X	
(2,3)	–	1	0			X	X			
(2,6)	0	1	–			X				X
(4,5)	–	0	1					X	X	
(4,6)	0	–	1					X		X

Figure 25: Prime Implicant Chart with No Essential Prime Implicants

implicant was found or no essential prime implicants were found. Both are split up into WithEpi.py and WithoutEpi.py. WithEpi.py also includes different functions to determine if another iteration is necessary or not. For that, it examines the list of remaining row numbers that need to be covered and uses the situation fitting function. In withoutEpi.py only Petrick's method and its rules are located. Afterward, all the functions get summed up in the QuineMcCluskey.py. Lastly, the Quine-McCluskey function uses the resulting NumPy array as input for the translateToFunction() method, to generate the end list and pass it to the main function. Looking further at the structure of the implementation, the generating of truth tables and finding all minterms are all done by setupQM.py. CalcAndRedu.py is responsible for all filtering and reduction functions. This includes calculating the essential prime implicants and filtering the row numbers for withEpi() and withoutEpi().

4.4 Implementation of Petrick's

In this application, the Petrick's method is only implemented as an engine for the Quine-McCluskey-Algorithm, to calculate the cases in which the Quine-McCluskey-Algorithm does not find any essential prime implicants. The implementation is located in the withoutEpi package, and it gets all the information it needs from the Quine-McCluskey part of the code. To get the needed terms, withoutEpi calculates the terms needed for Patrick's method by building a matrix that represents the prime implicant chart. They are saved with a key in a dictionary and this key is used as a substitution for the groups, which can be seen in Figure 26.



```
{'0': ([0, 4, 2, 6], '0-1-10'),
 '1': ([0, 8, 2, 10], '-10-10'),
 '2': ([4, 6, 5, 7], '01-1-1'),
 '3': ([5, 13, 7, 15], '-11-11'),
 '4': ([8, 10, 9, 11], '10-1-1'),
 '5': ([9, 13, 11, 15], '1-1-11')}
```

Figure 26: Example of dictionary for substitution

This pairing of keys is the input for the Petrick's method, where the format uses terms in the form of strings. In these strings, literals are separated with $+$ as logical OR and with $*$ as logical AND. Following the build of the function, the algorithm takes a pair of two terms and combines them into one. After merging them together, a set of rules need to be applied to those terms and these rules are located in rules(). It makes sure that if the same variables are connected with an $*$ or a term is already existing in another one, the spare literal will be deleted. This will be done until there are no brackets anymore and you get a final function. To find the minimized version with the Petrick's method, you look for the term with the smallest size and resubstitute it back with the mentioned dictionary. Since this function is only a plug-in into the QuineMcCluskey algorithm, it is possible that the Petrick's method has to be used after the second or higher iteration, which causes it to have essential prime implicants in earlier iterations. If that is the case, these essential prime implicants also have to be added to the final function.

4.5 Obstacles

There were many problems regarding both implementations. For the Karnaugh-Veitch algorithm, it started with choosing the right format because there was no universal method to depict multiple groups. As the work progressed, more and more data structures got added to the code which made it increasingly difficult to monitor all the relations. The most difficult task was not to build the groups, but to filter them. Since all possible groups were stored in one list, some criteria were needed to differentiate between important and non-important groups. These criteria seemed very intuitive, but were hard to establish. Figure 29 is an example of such a filtering problem. It represents the Karnaugh-Veitch-Diagram for the function output 1111 1111 1110 0111. Groups are represented by a list of tuples, where each tuple is composed of a row number and a column number. Every column and row starts with zero and ends with three. In this example, the import groups are:

(a) $[(0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,2),(1,3)]$

(b) $[(0,0),(0,1),(3,0),(3,1)]$

(c) $[(1,1),(1,2),(2,1),(2,2)]$

(d) $[(1,2),(1,3),(2,2),(2,3)]$

(e) $[(0,0),(0,1),(0,2),(0,3)]$

(f) $[(0,3),(1,3),(2,3),(3,3)]$

Figure 27 and Figure 28 show the difference between deleting group (d) and group (c) first. In Figure 27 group (d) gets deleted first which causes cell (2,2) to only be viable through group (c). This causes group (e) to be redundant and deletable. This gives us a total number of four groups. In Figure 28 on the other hand, group (c) gets deleted first, which causes cell (2,1) to only be viable by group (e) so that it cannot be deleted like in Figure 27. This gives us a total number of five groups. So even though we have the same groups, the order in which they get deleted matters.

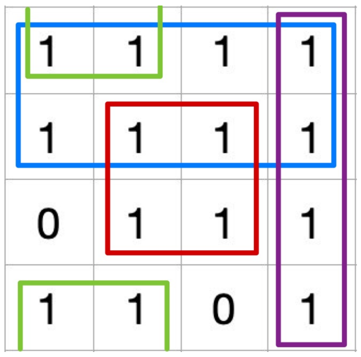


Figure 27: (d) gets deleted first

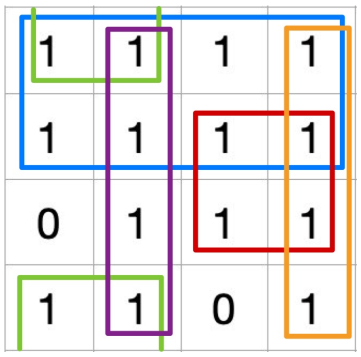


Figure 28: (c) gets deleted first

Figure 29: Example, where the order of deletion matters

Since this problem was mostly connected to one group, the problem could be solved by shuffling the groups. This increased the runtime drastically but made sure it always found the optimal solution, so it was essentially trading runtime for accuracy. There was another problem that needed to be fixed. The ruling for grouping stated that the groups have to be as big as possible, but in one instance this was not the case. This problem can be seen in Figure 32. The groups here are:

(a) $[(1,1),(1,2),(2,1),(2,2)]$

(b) $[(0,2),(1,2)]$

(c) $[(2,2),(2,3)]$

(d) $[(1,0),(1,1)]$

(e) $[(2,1),(3,1)]$

In Figure 30, there is a total number of five groups while in Figure 31 there are only four. This is caused because all the tuples in group (a) are also in other groups that need them. Therefore, group (a) could technically be deleted which is shown in Figure 31. This is a contradiction to the before-mentioned rule. To fix this, all the cases that had this problem needed to have their biggest group deleted.

Due to the Quine-McCluskey being the second algorithm that was implemented, there were fewer problems. After the first step of combining all the minterms, I realized that the row

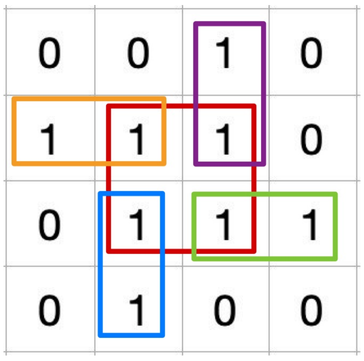


Figure 30: Non-Optimized Grouping

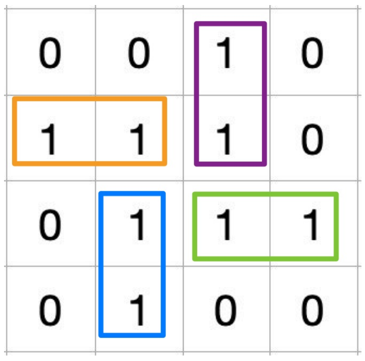


Figure 31: Opimized Grouping

Figure 32: Example where a basic rule needs to be broken

numbers were needed to do the following steps, and so I needed to fit this information into the model, which was through the use of the dictionaries. This caused more problems, as the previously missing row numbers need a lot more functionality to be added to withEpi.py and withouEpi.py. This is the reason dictionaries were included in the first place, to make sure that the row numbers can always be easily accessed for the prime implicant chart. This resulted in it being more complex than it was supposed to be. Based on some misconceptions about the algorithm, there were a lot of edge cases uncovered that needed to be addressed. The biggest problem was the deleteWithoutOccurence() function because it had no description of which prime implicants needed to be removed so it had to be determined without any algorithm and through experimenting. The last big problem was the transition from withEpi() to withoutEpi(). Since there is a possibility that in the while-loop of withEpi() a provisional result is calculated with no essential prime implicant it needs to be changed to the withoutEpi() function. Therefore, the best solution was to just give the provisional result to the other function when the remaining row numbers that had to be covered were below five because up to four variables could easily be solved without using the Petrick's method. These cases are distinct and solved by the allIndCases(). Figures 34 and 35 show prime implicant charts for the function output 0000 0100 1101 1011 and their transition from a chart that is used for withEpi() to a chart that is used for withoutEpi(). In Figure 33 there is a single essential prime implicant which after getting removed forms the prime implicant chart in Figure 34. This essential prime implicant chart contains no essential prime implicant, which is why withoutEpi() needs to be used for it.

4.6 Testing

There were a couple of things that the code had to be tested on. Firstly if it works correctly and secondly how fast the algorithms are. To test these problems, sample files were needed, which setupTest.py is responsible for. There, all the possible outputs a truth table can produce are generated for any number of variables. To test if the algorithms work correctly, the sample data is used as input for the two different algorithms and the resulting lengths of the outputs

	A	B	C	D		5	8	9	11	12	14	15
$(8,9)$	–	0	0	1			X	X				
$(8,12)$	0	0	–	1			X			X		
$(9,11)$	1	–	0	1				X	X			
$(11,15)$	1	1	–	1					X			X
$(12,14)$	0	–	1	1						X	X	
$(14,15)$	–	1	1	1							X	X
(5)	1	0	1	0		X						

Figure 33: Example for withEpi() prime implicant chart

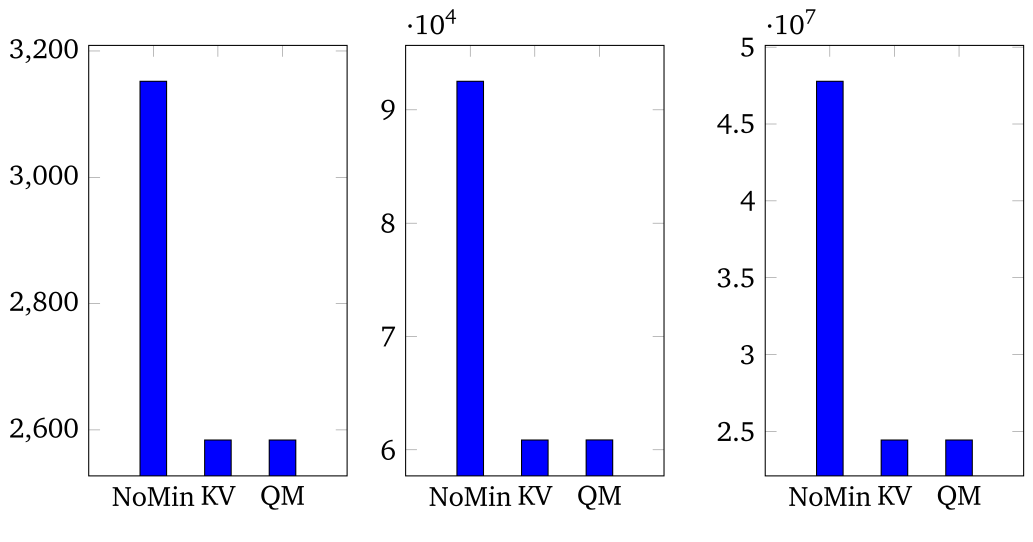
	A	B	C	D		8	9	11	12	14	15
$(8,9)$	–	0	0	1		X	X				
$(8,12)$	0	0	–	1		X			X		
$(9,11)$	1	–	0	1			X	X			
$(11,15)$	1	1	–	1				X			X
$(12,14)$	0	–	1	1					X	X	
$(14,15)$	–	1	1	1						X	X

Figure 34: Example for withoutEpi() prime implicant

are compared. Since both algorithms were not implemented simultaneously, and they did not work for all situations during their development. The comparison between them always showed, which function output did not work for which algorithm. Therefore, If the length of all the outputs were the same in the end, it would guarantee that the codes work correctly. Unfortunately, you can not test the algorithms if the solution contains the same terms, because the minimization is not unique, and it is possible for one truth table to be represented by multiple minimized Boolean functions. To test the runtime of both algorithms, I used the defaultTimer() function from the timeit module to calculate the start and end times of every calculation in the sample file. End and start time can then be subtracted to calculate the runtime for one sample. To see how much memory space the result for both algorithms take, I used the asizeof() function from the pympler module, which returns the size for Python objects in Bytes. Karnaugh-Veitch only works for two to four variables, therefore the cases for five or above are not tested for Quine-McCluskey. Since there are 2^{32} possible function outputs for truth tables with five literals and even 2^{64} for six literals, it would take too long to run Quine-McCluskey for all possible function outputs. So I used reverse engineering to test for some samples. In this case, reverse engineering meant using a minimized Boolean function to create a new truth table by hand and inputting it into the algorithm to see if the result remains unchanged. What was also important when testing the memory use was to compare non-minimized Boolean functions and the result the algorithms returned. Thus, a function needed to be implemented that calculated all Boolean functions without minimization. This is done in TestMemory().

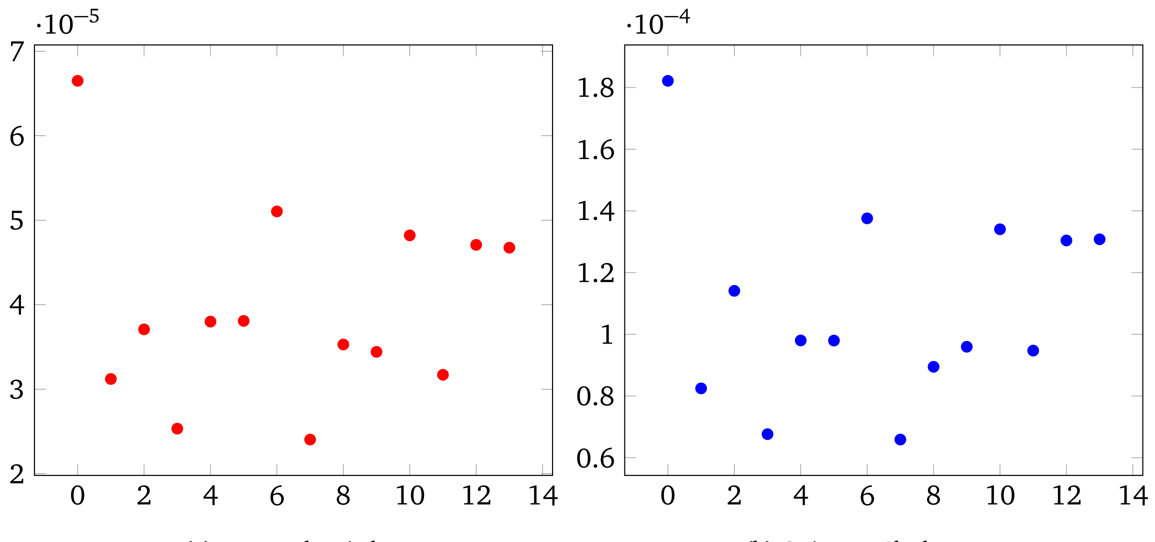
5 Results

In this section, we discuss all results gained from the implementation of the minimizing algorithms. These results were all calculated by using the sample files mentioned in the testing section but can vary depending on the used device. If we take a look at an algorithm regarding x literals, that means we take a look at the algorithm with the sample file as input, which would contain 2^x input tuples. The first thing to talk about is the storage consumption. Figure 35 shows how much memory is used for the different combinations of literals and used algorithms. The Y-axis represents the storage used in Byte and the X-axis which method was used. For all three graphs (a), (b) and (c) the most memory is used when not running a minimization algorithm. This increase in storage seizure is up to 50% in comparison to both algorithms for two and three literals. For four literals, it is even close to 100%. The minimizing Algorithms on the other hand only vary a small amount. For two literals the memory usage is the same with 2584 Byte, for three literals Karnaugh-Veitch's memory usage is 60864 Byte and Quine-McCluskey's is 60872 Bytes and for four variables the Karnaugh-Veitch takes 24444472 Bytes of storage, while Quine-McCluskey takes 24445232 Bytes. Karnaugh-Veitch has the smallest memory consumption regarding any number of literal, beating Quine-McCluskey only by around 1%. The average size for a single minimized Boolean function with two literals is 160 Byte, for three literals it is 240 Byte and for four literals it is 370 Byte. All in all the statistics show that there is not a big difference in memory consumption between both algorithms and for a bigger sample size the memory use is still reasonable, since the 2^{16} minimized Boolean functions only use around 25 MegaByte of storage.
The next topic to look at is the different runtimes of both algorithms for each number of literals. Figures 36 to 39 represent all the combinations possible. The Y-axis describes the runtime in seconds and the X-axis is the function outputs of the sample file. Karnaugh-Veitch is plotted in red, while Quine-McCluskey is presented in blue. Figure 36 starts with the runtime for 2 Literals. Karnaugh-Veitch has an average runtime of $3,9*10^{-5}$ seconds, a minimum of $2,4*10^{-5}$ seconds, and a maximum of $5,6*10^{-5}$ seconds, while Quine-McCluskey has an average runtime of $1,06*10^{-4}$ seconds, a minimum of $6,6*10^{-5}$ seconds, and a maximum of $1,6*10^{-4}$ seconds. This means that Karnaugh-Veitch's average runtime is faster than Quine-McCluskey's and even the output with the biggest runtime in Karnaugh-Veitch is around the speed of the fastest output in Quine-McCluskey. The gap between both average times is $6,7*10^{-5}$.
Figure 37 describes the runtime for three literals. Karnaugh-Veitch has an average runtime of $9,8*10^{-5}$ seconds, a minimum of $3,2*10^{-5}$ seconds, and a maximum of $8,03*10^{-4}$ seconds, while Quine-McCluskey has an average runtime of $*2,24^{-4}$ seconds, a minimum of $8,58*10^{-5}$



(a) 2 Literals (b) 3 Literals (c) 4 Literals

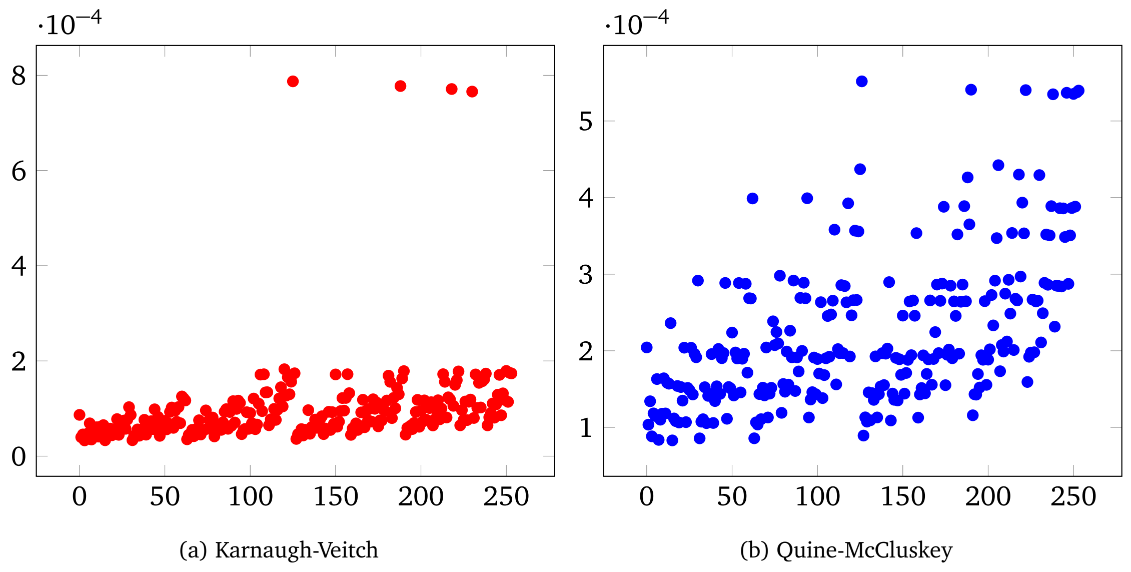
Figure 35: Storage Use for Different Number of Literals



(a) Karnaugh-Veitch (b) Quine-McCluskey

Figure 36: Runtime for 2 Literals

seconds, and a maximum of $5,6*10^{-4}$ seconds. Karnaugh-Veitch's runtime is mostly between 0 seconds and $2*10^{-4}$ seconds with some exceptions that reach up to $8*10^{-4}$ seconds, while Quine-McCluskey is equally distributed between $1*10^{-4}$ seconds and $5*10^{-4}$ and doesn't have a single time above $6*10^{-4}$ seconds. Here the average runtime for Karnaugh-Veitch still beats Quine-McCluskey by $1,26*10^{-4}$ seconds but the gap is significantly bigger than with two literals.
Figure 38 describes Karnaugh-Veitch with four literals. It has an average runtime of $5,4*10^{-4}$ seconds, a minimum of $4,6*10^{-5}$ seconds, and a maximum of 0,015 seconds. Figure 39 on the other hand describes Quine-McCluskey with four literals. Quine-McCluskey has an average runtime of $7,04*10^{-4}$ seconds, a minimum of $1,25*10^{-5}$ seconds, and a maximum of $8,82*10^{-3}$ seconds. Here it is clearly visible that both algorithms fluctuate a lot. Karnaugh-



(a) Karnaugh-Veitch (b) Quine-McCluskey

Figure 37: Runtime for 3 Literals

Veitch and Quine-McCluskey both stagnate at 0.01 seconds. The graphs look very similar, in consideration of their distribution between 0 seconds and 0,005 seconds but Karnaugh-Veitch's average time is still faster with $2 * 10^{-4}$ seconds. To conclude this section both algorithms are rather fast and only differentiate between 100 microseconds which is why they can both be used equally. However, the Karnaugh-Veitch algorithm is just slightly faster.

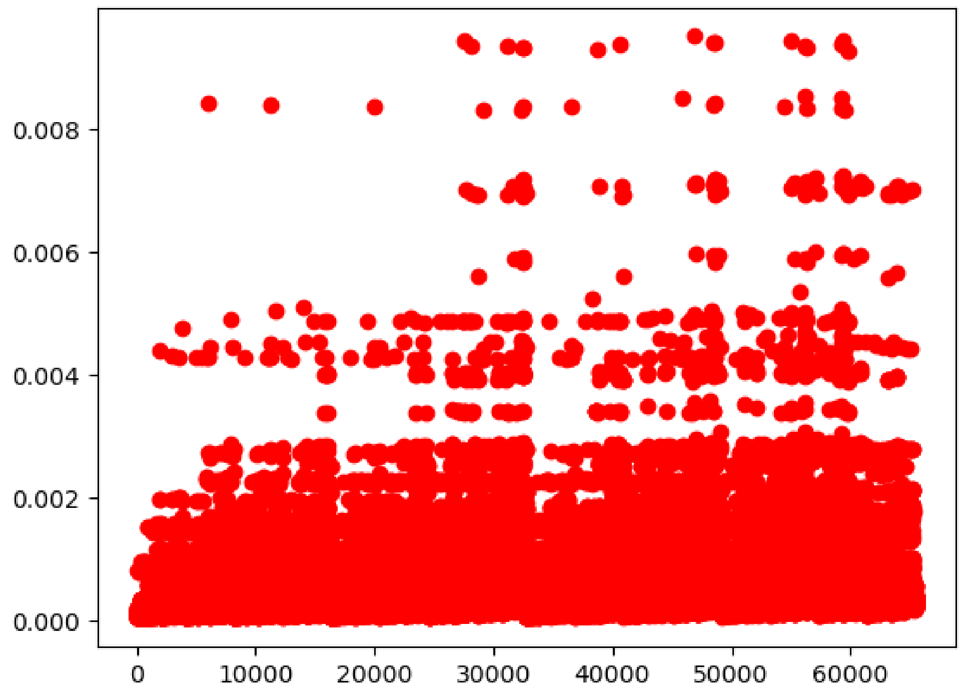


Figure 38: Runtime for Karnaugh-Veitch with 4 Literals

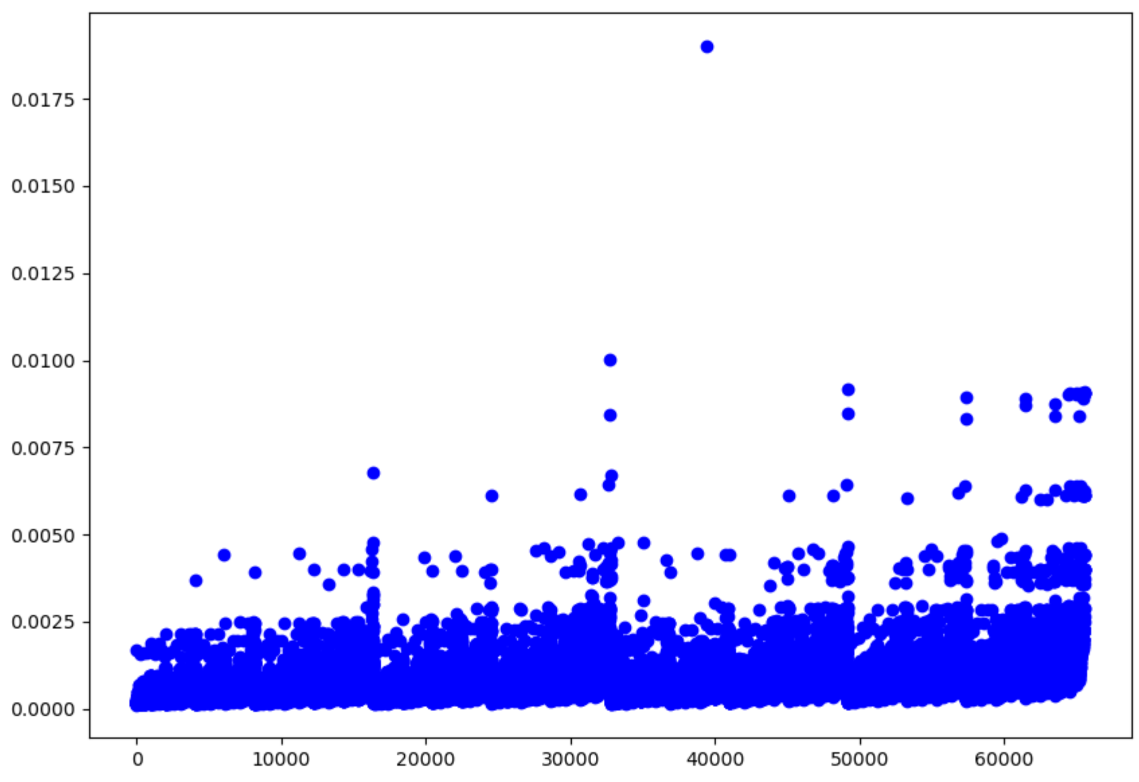


Figure 39: Runtime for Quine-McCluskey with 4 Literals

6 Discussion

6.1 Improvements

This implementation is not optimized and can be improved in different areas. Starting with the Karnaugh-Veitch-Algorithm it would be more efficient to only store the important groups and not save all unimportant ones when building them. By filtering the groups there is a lot of runtime and storage lost, because the list of important groups has to be shuffled, and then they have to all be used as input for the last part of the code which increases the runtime by the number of extra lists that were added. A way to improve that would be to avoid reshuffling at all and find a way to detect the best list of the groups without filtering them. Furthermore, there is a lot of typecasting involved, so it would make the code more understandable by optimizing it and leaving out unnecessary data structure conversion. In Quine-McCluskey some things can be improved as well. The use of dictionaries, where minterms are used as keys to store their matching row numbers, is redundant because the minterms can be seen as binary numbers that can be converted to decimal values, which would then represent the row numbers. Lastly, If Quine-McCluskey runs with six Variables and a lot of function outputs of *one*, the algorithm takes a lot of time to calculate the result. This can be refined by finding ways to exit the while-loop in withEpi() faster.

6.2 Future Work

There are some parts of this work that can be extended. To make the application more user-friendly, a GUI can be developed for it. Then you can try to implement the Karnaugh-Veitch algorithm for five or more variables. As a foundation, this code can be expanded by replacing minor parts of the code in KarnaughVeitch(), adding more cases to the groupFunction as well as expanding the frameFunctions. The Input and Output could be worked on and instead of using CSV files as the format of choice, a more fitting form can be selected to represent the truth tables more intuitively.

7 Conclusion

The goal of this work was to implement the presented algorithms and compare them regarding specific efficiency criteria. To decide which algorithm should be used for which case of literal numbers, both algorithms were tested for runtime and memory usage. My hypothesis was that the Karnaugh-Veitch-Algorithm would work the best for situations with two to four variables regarding runtime, which was the case. The problem is that the samples for five to six literals are not implemented for Karnaugh-Veitch, therefore these cases can only be solved with Quine-McCluskey. When talking about faster, the results are only 100 microseconds apart, which is very small even for higher amounts of samples. Nevertheless, the average runtime for two to four variables is calculated faster with Karnaugh-Veitch. The difference between the runtime is still surprisingly close which is most likely caused by the shuffling of the list in Karnaugh-Veitch. Some parts of the algorithm had to unnecessarily run multiple times, which increased the original runtime by the number of different lists of groups the shuffling produced. To take memory usage into consideration when talking about the use cases, we can say that using any of the two mentioned algorithms to minimize Boolean functions lowers the storage consumption by 50% and up to 100%. Among themselves, the difference between memory usage is very little. For four variables, Karnaugh-Veitch has around 800 Bytes less memory usage than Quine-McCluskey. Finally, it can be said that using Karnaugh-Veitch for two to four variables is more efficient since it is faster and takes less memory to store. In case they are above four literals, it needs to be solved with Quine-McCluskey because these cases are not implemented with Karnaugh-Veitch. In conclusion, debugging was more time-consuming than expected, which is why some features, that were mentioned in the Future Work section, are missing. Nevertheless, the results are as expected and everything works as it was planned.

References

[1] Paul MA Antony, Rudi Balling, and Nikos Vlassis. "From Systems Biology to Systems Biomedicine". In: *Current Opinion in Biotechnology* 23.4 (2012). Nanobiotechnology Systems biology, pp. 604–608. ISSN: 0958-1669. DOI: https://doi.org/10.1016/j.copbio.2011.11.009. URL: https://www.sciencedirect.com/science/article/pii/S0958166911007208.

[2] Eric Davidson and Michael Levin. "Gene regulatory networks". In: *Proceedings of the National Academy of Sciences* 102.14 (2005), pp. 4935–4935. DOI: 10.1073/pnas.0502024102. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.0502024102. URL: https://www.pnas.org/doi/abs/10.1073/pnas.0502024102.

[3] Julio Saez-Rodriguez et al. "Discrete logic modelling as a means to link protein signalling networks with functional analysis of mammalian signal transduction". In: *Mol Syst Biol* 5.331 (2009). DOI: 10.1038/msb.2009.87. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2824489/.

[4] Roded Sharan and Richard M. Karp. "Reconstructing Boolean Models of Signaling". In: *Journal of Computational Biology* 20.3 (2013). PMID: 23286509, pp. 249–257. DOI: 10.1089/cmb.2012.0241. eprint: https://doi.org/10.1089/cmb.2012.0241. URL: https://doi.org/10.1089/cmb.2012.0241.

[5] *Minimizing Boolean Functions*. Dr. Christoper Vickery. 2021. URL: https://babbage.cs.qc.cuny.edu/courses/Minimize.

[6] M. Karnaugh. "The map method for synthesis of combinational logic circuits". In: *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics* 72.5 (1953), pp. 593–599. DOI: 10.1109/TCE.1953.6371932.

[7] E. J. McCluskey. "Minimization of Boolean functions". In: *The Bell System Technical Journal* 35.6 (1956), pp. 1417–1444. DOI: 10.1002/j.1538-7305.1956.tb03835.x.

[8] Stanley R Petrick. "A direct determination of the irredundant forms of a Boolean function from the set of prime implicants". In: *Air Force Cambridge Res. Center Tech. Report* (1956), pp. 56–110.