UDC 004.02:004.3+519.85

OPTIMIZATION PROBLEM FOR NUMBER OF LOGIC GATES NEEDED TO IMPLEMENT MULTIPLE BOOLEAN FUNCTIONS USING DECODER

ЗАДАЧА ОПТИМІЗАЦІЇ КІЛЬКОСТІ ЛОГІЧНИХ ЕЛЕМЕНТІВ, ЩО ПОТРІБНІ ДЛЯ РЕАЛІЗАЦІЇ ДЕКІЛЬКОХ БУЛЕВИХ ФУНКЦІЙ НА ДЕШИФРАТОРІ

[1]Onai M., [2]Skopyk H.
[1]Онай М., [2]Скопик Г.

[1,2]National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine
ORCID: [1]https://orcid.org/0000-0002-4938-8355, [2]https://orcid.org/0009-0008-1411-6422
E-mails: [1]onay@pzks.fpm.kpi.ua, [2]gms.4job@gmail.com

Copyright © 2024 by author and the journal "Automation of technological and business – processes".
This work is licensed under the Creative Commons Attribution International License (CC BY).
http://creativecommons.org/licanses/by/4.0

DOI:

Abstract. *Background. Computer circuits enable almost every piece of electronics to function. An opportunity for their optimization was found in one of the ways to implement a circuit, which involves a decoder. Decoders are basic circuit components whose behavior makes it easy to implement multiple Boolean functions. Functions are just rules by which informational signals are being processed in the circuit. To implement a function Logic gatEs (LEs) are used. LEs group multiple input signals and process them using binary logic operators like AND or NOR.*

Purpose. Develop a computer program that, based on provided functions, will find a solution associated with approximately minimal number of logic gates.

Relevance. Prevalence of circuits increases the value of small optimizations, since any improvement gets multiplied by millions of circuits produced. Optimization at the logic level is being explored, which will retain value regardless of the physical properties of the circuit. Additionally, the theory of optimizing LE usage for circuits built on decoders is examined, which will alleviate future research.

Methods. Development of the theory arose from an understanding that input signals can be grouped using LEs consciously. This way, LE outputs can be used in more than one function, which results in reduction of total number of LEs. It becomes evident that the problem can be restated in terms of sets. Similarity to known NP-hard problems emerges, which prompts usage of existing solution approaches. Among them, the greedy algorithm is chosen for its simplicity in program implementation. The connection between the problem and digital circuitry is weakened by the introduction of special evaluation functions that work with sets. So, the problem is reduced to several mathematical rules, which then are used as the appropriate parts of the greedy algorithm. Development of knowledge necessary to create the target program is thus concluded.

Results. The target program is successfully implemented and can be seen in full at https://github.com/Gleb-05/MakeFuncForDC. *A test is conducted on 1000 sets of four pseudorandom eight-term functions. It is estimated that the implementations proposed by the program require on average 0.75 of the initial number of LEs. The average running time is estimated at 2 milliseconds. In contrast, brute-force search for the same problem is theorized to run for years.*

Conclusions. A working program that finds an almost optimal solution is successfully created and tested. Its performance makes it potentially useful in the industrial scale of circuit manufacturing. Since the program essentially offers workload management, its possible applications go beyond the domain of digital electronics. Additionally, the theory of optimizing the use of LEs is presented for the first time. It can be used as the basis for future research on the implementation of multiple Boolean functions using decoder.

Анотація. *Контекст. Завдяки комп'ютерним схемам функціонує майже будь-яка електроніка. Можливість оптимізувати схеми знайдено в одному зі способів їх реалізації, що залучає дешифратор. Дешифратори є базовими компонентами схем, поведінка яких дозволяє легко реалізувати кілька булевих функцій. Функції – це лише правила, за якими інформаційні сигнали обробляється в схемі. Щоб реалізувати функцію використовуються Логічні Елементи (ЛЕ). ЛЕ групують кілька вхідних сигналів і обробляють їх за допомогою двійкових логічних операторів, як то AND або NOR.*

Мета. Розробити комп'ютерну програму, яка на основі наданих функцій знайде рішення, пов'язане з близькою до мінімальної кількістю Логічних Елементів.

Актуальність. Поширеність схем підвищує цінність невеликих покращень, оскільки будь-яке вдосконалення помножується на мільйони виготовлених схем. Досліджується оптимізація на логічному рівні, яка збереже значення незалежно від фізичних властивостей схеми. Крім того, розроблюється теорія для оптимізації використання ЛЕ, яка полегшить майбутні дослідження.

Методи. Поштовхом для розвитку теорії було розуміння, що вхідні сигнали можна свідомо групувати за допомогою ЛЕ. Таким чином, виходи ЛЕ можна використовувати в більш ніж одній функції, що призводить до зменшення загальної кількості ЛЕ. Стає очевидним, що задачу можна виразити в контексті множин. Виявляється подібність до відомих NP-складних задач, що спонукає до використання існуючих підходів до рішення. Серед них обрано жадібний алгоритм за простоту його програмної реалізації. Зв'язок між задачею та схемотехнікою послаблюється після введення спеціальних оціночних функцій, які працюють із множинами. Отже, задача зводиться до кількох математичних правил, які потім використовуються як відповідні частини жадібного алгоритму. На цьому завершено формулювання знань, необхідних для створення цільової програми.

Результати. Цільова програма успішно реалізована, з нею можна ознайомитись на https://github.com/Gleb-05/MakeFuncForDC. Проведено тести на 1000 наборах з чотирьох псевдовипадкових функцій на вісім елементів. Підраховано, що реалізації, запропоновані цільовою програмою, вимагають у середньому 0,75 від початкової кількості ЛЕ. Середній час роботи становить 2 мілісекунди. В порівнянні, пошук грубою силою для тої ж задачі теоретично потребує тисячі років.

Підсумок. Працююча програма, яка знаходить близьке до оптимального рішення, успішно створена та протестована. Продуктивність програми робить її потенційно корисною у промислових масштабах виробництва схем. Оскільки програма, по суті, пропонує керування робочим навантаженням, її можливі застосування виходять за межі цифрової електроніки. До того ж, вперше представлено теорію оптимізації використання ЛЕ. Викладена теорія може слугувати основою для подальших досліджень, що стосуються реалізації кількох булевих функцій на дешифраторі.

Keywords: digital electronics, combinational circuit, circuit implementation, Boolean algebra, decoder, data protection, cost minimization, NP-hard problem

Ключові слова: цифрова електроніка, комбінаційна схема, реалізація мікросхеми, булева алгебра, дешифратор, захист інформації, мінімізація витрат, NP-складна задача

INTRODUCTION

Computer circuits form the backbone of modern technology, ensuring correctness of its functioning on the most fundamental level. With each stroke of a keyboard, each screen touch, each glance on the monitors, we rely on the accuracy of hardware in a plethora of basic circuits. Investigating ways to enhance these circuits by means different from purely physical is imperative to the technological progress.

More specifically, we will investigate decoders, which are one of many basic components that are used in combinational circuits. Some important applications of them are: code conversion, computer memory management, and data distribution [1]. They are also known for their usage in data protection systems, namely in those related to transposition ciphers. Decoders allow one output to be activated only on the occurrence of a particular combination of input signals.

In digital electronics there is a typical task of implementing multiple Boolean functions using decoder. This task is completed by grouping decoder's outputs using Logic gatEs (LEs) in accordance with each function. Let's agree to address decoder's outputs as "constituents". Provided that functions are both numerous and large, arbitrary constituents will happen to be a part of more than one function. This presents an opportunity to group repeating constituents using LEs consciously, so that the output of each such LE could be used by more than one function. In return, total number of LEs needed to make the circuit would drop. This approach is what underlies the optimization being developed. One of the many practically valuable consequences of such optimization would be the drop in the cost of electronics production on the industrial scale.

Preliminary research did not reveal any works utilizing this kind of approach to the optimization of circuit design. However, in Combinatorics conceptually similar problems already exist. Namely, "search" versions of set cover problem and set packing problem [2–3]. Both are classified as NP-hard problems, which in practice means that the number of possible solutions grows exponentially relative to the size of the problem.

Development of a new perspective on a potentially NP-hard problem was deemed valuable, since this class of problems is known for its theoretical reusability and various practical applications. Existing solutions for similar problems were found to be unapplicable, which further increases the theoretical novelty of the study.

The purpose of the work is to develop a computer program that, based on provided functions, will find exact groups of constituents associated with the least amount of LEs used. To gain knowledge necessary for creating such a program, the theory behind optimizing LE usage for circuits built on decoders is explored.

REVIEW OF THE LITERATURE

Because this study is explorative in nature, excessive amount of information is not necessary. As was already mentioned, algorithms suitable to fulfill the purpose of this work were not found. This was taken to be an indication of high specificity of the defined problem. However, while searching for similar problems, it was revealed that this one

might be of NP-hard type [4]. As was already mentioned, this type of problems is well researched. Making use of already known solution approaches, like greedy algorithm, exploration in this study was rather guided [5].

Looking forward, LE usage is conceptually tied to the loss function, which is used in both greedy and genetic algorithms [6]. At the stage of choosing between possible solutions, the greedy algorithm was picked because its software implementation is the most straightforward, so that purpose of the study is guaranteed to be fulfilled.

OBJECT, SUBJECT, AND METHODS OF THE RESEARCH

The object of this study is the process of implementing multiple Boolean functions using decoder. The subject of the study is the task of minimizing amount of LEs used in the process. It makes sense to talk about number of LEs as “cost of implementation”, which presents the subject of the research to be the task of cost minimization.

Let’s formally define the problem statement.

Take normal forms of Boolean functions that are being implemented using decoder to be $z_i, i = N \dots 1$. These normal forms are defined for one type of LE. This type of LE has k inputs such that $k \geq 2$. Consider it possible to use less than all k inputs, but no less than two. Take each function to be comprised of m_i constituents.

Define an arbitrary number of constituents to be grouped “automatically” in the following case. Grouping can be visualized as using parentheses. Starting from the first term, k consecutive terms are grouped with LE once (or twice, if LE contains negation). Grouping of consecutive terms continues on yet ungrouped terms until their number becomes less than k. After this, created groups are considered to be ungrouped terms, and the process repeats starting from the first term. When, or if, the number of terms is less than or equal to k, one final group is formed with exactly one LE, bringing total number of terms to one, thus terminating the process.

Let “bundle” denote a group of two or more constituents present in two or more normal forms. Note that the bundle is used in all relevant normal forms without any additional LEs spent beyond those required for the bundle itself. Additional constraint for bundles to not intersect in the final solution is used. Note that this constraint can be dropped to achieve generalized version of the problem.

Both bundles and constituents are considered to be “terms” in context of the normal forms in the final solution. Terms are assumed to be grouped automatically. Bundle constituents are also assumed to be grouped automatically. Note that using groups of less than k terms during grouping will lead to a much broader version of the problem.

With all the terms properly defined, the problem is as follows. For given functions define normal forms based on the given logic gates. Based on the found normal forms, define bundles and choose those that are associated with the implementation that requires the least number of LEs. The number of LEs is comprised of LEs used to form bundles, and of LEs needed in each normal form to group its not-bundled constituents with the bundles present in it.

Thus, the problem statement is concluded.

To establish a firm understanding of the problem, consider the following example. The task is to build a code converter using as few LEs as possible. Functions to implement are showed in z columns of **Table 1**, where N denotes respective constituents.

Table 1 – Decoder truth table

x4	x3	x2	x1	N	z4	z3	z2	z1
0	0	0	0	0	0	0	1	0
0	0	0	1	1	0	1	0	0
0	0	1	0	2	0	1	0	1
0	0	1	1	3	0	1	1	0
0	1	0	0	4	0	1	1	1
0	1	0	1	5	1	0	0	0
0	1	1	0	6	1	0	0	0
0	1	1	1	7	1	0	1	1
1	0	0	0	8	1	0	1	0
1	0	0	1	9	1	1	0	1
1	0	1	0	10	1	1	0	1
1	0	1	1	11	1	1	1	0
1	1	0	0	12	1	1	1	0
1	1	0	1	13	0	0	0	0
1	1	1	0	14	0	0	0	1
1	1	1	1	15	0	0	1	0

For implementation, 3-NAND logic gates and a 4-input decoder with inverse outputs are used. In accordance with the type of LE, which is NAND, respective normal forms for each function are written.

$$z_4 = 5 \vee 6 \vee 7 \vee 8 \vee 9 \vee 10 \vee 11 \vee 12 = \overline{\overline{5} \cdot \overline{6} \cdot \overline{7} \cdot \overline{8} \cdot \overline{9} \cdot \overline{10} \cdot \overline{11} \cdot \overline{12}}$$

$$z_3 = 1 \vee 2 \vee 3 \vee 4 \vee 9 \vee 10 \vee 11 \vee 12 = \overline{\overline{1} \cdot \overline{2} \cdot \overline{3} \cdot \overline{4} \cdot \overline{9} \cdot \overline{10} \cdot \overline{11} \cdot \overline{12}}$$

$$z_2 = 0 \vee 3 \vee 4 \vee 7 \vee 8 \vee 11 \vee 12 \vee 15 = \overline{\overline{0} \cdot \overline{3} \cdot \overline{4} \cdot \overline{7} \cdot \overline{8} \cdot \overline{11} \cdot \overline{12} \cdot \overline{15}}$$

$$z_1 = 2 \vee 4 \vee 7 \vee 9 \vee 10 \vee 14 = \overline{\overline{2} \cdot \overline{4} \cdot \overline{7} \cdot \overline{9} \cdot \overline{10} \cdot \overline{14}}$$

Considering number of LE inputs is three, constituents of each function are grouped in some way, which leads to the "circuit form" of the function. For now, let's assume that the way of grouping constituents is automatic. See circuit form of the second function defined automatically:

$$z_2 = \left(\overline{\left(\overline{\overline{\left(\overline{\overline{\overline{0}\cdot\overline{3}\cdot\overline{4}}}\right)\cdot\left(\overline{\overline{\overline{7}\cdot\overline{8}\cdot\overline{11}}}\right)\cdot\overline{12}}}\right)\cdot\overline{15}}\right)$$

This circuit form makes use of seven LEs, which is evident from the number of parentheses and negations above them. In case that all circuit forms were defined automatically, total LE cost reaches 26.

However, the way constituents are grouped doesn't have to be automatic. Consider that fact that fourth and third function share as much as four constituents, namely $(9{,}10{,}11{,}12)$, to which a bundle given below corresponds.

$$\left(\overline{\overline{\left(\overline{\overline{\overline{9}\cdot\overline{10}\cdot\overline{11}}}\right)\cdot\overline{12}}}\right)$$

Acknowledging two more bundles comprised of $(7{,}8)$ and $(3{,}4)$, total cost drops to 24 LEs, bearing two spare LEs compared to an automatic implementation. Notice how this implementation leverages a counterintuitive grouping process in which less than all of LE's inputs are used.

The circuit forms for this implementation are given in **Table 2** along with the associated cost. There, double negation is implied when using parentheses unless shown otherwise, and negation of constituents is omitted for convenience.

Table 2 – Implementation example with associated LE cost

Bundles	Cost		Normal Forms	Cost
$b_1 = (3\cdot 4)$	2		$z_4 = \left(\overline{(5\cdot 6\cdot b_2)\cdot b_3}\right)$	3
$b_2 = (7\cdot 8)$	2		$z_3 = \left(\overline{(1\cdot 2\cdot b_1)\cdot b_3}\right)$	3
$b_3 = \left((9\cdot 10\cdot 11)\cdot 12\right)$	4		$z_2 = \left(\overline{(0\cdot b_1\cdot b_2)\cdot(11\cdot 12\cdot 15)}\right)$	5
			$z_1 = \left(\overline{(2\cdot 4\cdot 7)\cdot(9\cdot 10\cdot 14)}\right)$	5
Cost of bundles	8		Cost of normal forms	16
Total cost is 8+16 => 24				

Further manual search revealed a solution associated with a cost of only 20 LEs, with bundles defined to be $(3{,}4), (7{,}8), (9{,}10), (11{,}12)$. Compared to the automatic implementation, this one has as much as six spare LEs. Meaning, at the expense of approximately one hundred LEs it is possible to make five circuits instead of four, which is definitely beneficial on an industrial scale.

With an example that clearly demonstrates the value of making use of repeating constituents of multiple functions, let's move on to a more rigorous study of the theory behind optimizing the LE cost.

To demonstrate some of the results of the theoretical work, an updated problem statement is given, and it will be explained immediately after in parts.

Consider a universe of constituents from 0 to $2^n - 1$. On this universe, let N sets, each of length m_i, be defined in accordance with the type of LE. Those sets are denoted as $z_i, i = N \dots 1$. Bundles are defined to be subsets of length greater than one which occur in more than one set. Bundles are still prohibited from overlapping in the final solution.

Additionally, take function $LEn_k(m)$ to return number of times a set of length m should be arranged in groups of k to get a set of length one. This function is based on the automatic grouping process, which is defined in accordance with the number of LE inputs. Finally, take a function $TLEn$ to return total number of times grouping occurred in a solution, which accounts for grouping in bundles, as well as for grouping in z sets after bundles were defined. This function is an auxiliary function that specifies how exactly LEn_k should be used for the evaluation of a solution.

Then, the problem is as follows. Define a family of bundles based on the intersections of z sets. Choose a configuration of bundles which is associated with the smallest value of $TLEn$.

Thus, the new problem statement is concluded.

Let's start the explanation with the reason for updating the problem statement. The new statement is constructed to abstract from details of digital electronics. This way, the fact that this problem is similar to the set cover problem becomes more apparent. Assuming all the bundles are defined, this problem is almost identical to the optimization version of a weighted set cover problem, where weight would be the number of LEs used. The only difference is that there is N sets instead of just one, which makes searching for the solution more difficult. Instead of searching for the least costly configuration of bundles for a single set, each bundle should be chosen with attention to constituents it affects across all the sets.

The similarity to the NP-hard variation of a set cover problem is taken to be a sufficient proof of NP-hardness of the problem. It is adequate to make use of common solution methods for this class of problems, including approximation (greedy algorithm), randomization (genetic algorithm), parametrization, and heuristics. Brute force is another known method which will be discussed briefly later on. Greedy algorithm is chosen over the genetic algorithm due to its deterministic nature and behavior that can be more easily studied.

The following structure of the greedy algorithm is devised, based on the key components: sampling, feature, choice, iteration and termination.

- Sampling. Raw data should be processed to get information suitable for the algorithm.
- Feature. The available information must be organized in some way, most often ordered by some quantitative characteristic, so that it is possible to make a choice.
- Choice. Based on the feature, there must be a selection rule, according to which only one choice is made. Usually, it is a choice of extremes – the largest or the smallest.
- Iteration and termination. Because a greedy algorithm is iterative in nature, there must be a rule by which information changes between iterations and a rule by which the algorithm terminates.

For now, let's assume a complete family of bundles has been obtained and is now used as information for the greedy algorithm. The next step is to define a feature by which those bundles will be compared, and the most obvious choice is to define a function that tells how many LEs is needed to cover all the terms in the bundle. This is exactly the $LEn_k(m)$ function. On a circuit level it assumes automatic grouping and is associated with a deterministic number of LEs needed for any given expression.

For the time being, let's get back to circuits and assume LE does not have negation (LE might be AND or OR). Let's analyze the effect of grouping k terms with one LE in an expression with an arbitrary number of terms. Obviously, total number of terms is reduced by k due to those grouped using LE, while simultaneously it's increased by one due to the fact that the group becomes a term itself. Thus, each use of one LE brings total number of terms in the expression down by $k-1$. Total number of times LE should be used is reasonably expected to resemble a quotient of expression's length divided by $k-1$. By definition, each normal form is effectively represented with a single output on a circuit. Similarly, each bundle should be represented with one output. This means that any expression that is being grouped using LEs should end up with a total number of terms equal to one. To better understand special edge cases, **Table 3** is constructed, where 'o' denotes a term, content of which does not matter.

Table 3 – LE usage for different number of terms with corresponding LE number

Terms	Usage for $k = 2$	LEn		Usage for $k = 3$	LEn
1	(o)	1		(o)	1
2	(o o)	1		(o o)	1
3	((o o) o)	2		(o o o)	1
4	((o o) (o o))	3		((o o o) o)	2
5	(((o o) (o o)) o)	4		((o o o) o o)	2
6	(((o o) (o o)) (o o))	5		((o o o) (o o o))	3
7	(((o o) (o o)) ((o o) o))	6		((o o o) (o o o) o)	3
8	(((o o) (o o)) ((o o) (o o)))	7		((((o o o) (o o o) o) o)	4

Based on **Table 3**, a formula (1) is derived:

$$LEn_k(m) = \left\lfloor \frac{m-2}{k-1} \right\rfloor + 1 \qquad (1)$$

where $LEn_k(m)$ is the number of Logic gatEs with k inputs needed to cover an expression with m terms; k, m are positive integers. Division is meant to be integer division, but on positive integer arguments floor division can be used.

This formula leads to many useful properties and intuitions, not presented in this work for the sake of brevity. However, note one of them that explains why updated problem statement disregards the type of LE in some sense. Reason for that is, number of negated Logic gatEs (abbreviated as $nLEn$), like NOR or NAND, can be easily derived from the formula (1) in the following way:

$$nLEn_k(m) = 2 \cdot LEn_k(m) - 1$$

On a circuit level, to preserve truth table of any expression, LEs are used twice, which is a common way to cancel out negations. Based on function specifications similar to those in table 1, any expression should use twice the LEs required for its not negated LE counterpart, except for the outer LE which is always used once. Exactly that is reflected in the formula above. Very important consequence for the new problem statement is that the number of used LEs can be evaluated without paying attention to its type, since the number always can be converted afterward.

Another application of formula (1) is an auxiliary function that allows to evaluate any solution. See **Table 2 – Implementation example with associated LE cost** to refresh the general understanding of how a particular solution can be evaluated. Formalization of this understanding results in the following formula:

$$TLEn = \sum_{j=1} LEn_k(b_j) + \sum_{i=N}^{1} LEn_k(u_i + m_i')$$

where "Total LE number", abbreviated as $TLEn$, makes use of: N – number z sets, b_j – number of terms in bundle j, u_i – number of constituents unique to z_i, uncommon across sets, occurring exactly once, m_i' – number of terms that are not unique across sets, counted for z_i after bundles are formed.

Thus, the justification for restating the problem is concluded, with valuable theoretical insights as a byproduct. The problem was compared to an existing NP-hard problem, greedy algorithm was chosen as a solution approach, and operating with digital circuits was changed to operating with sets.

Before moving forward in the development of the greedy algorithm, let's define the "sampling" part and specify how family of bundles is created. Sets (i.e., functions) and their constituents are not suitable for the algorithm by themselves.

Consider an alternative form of writing down problem's conditions as seen in **Table 4**, where information is taken from **Table 1**, c marks constituents, and z marks sets.

Table 4 – Constituents and the associated lists of sets where they are present

c	2	3	4	7	8	9	10	11	12
z	3 1	3 2	3 2 1	4 2 1	4 2	4 3 1	4 3 1	4 3 2	4 3 2

Notice that not all constituents are present in **Table 4**. This is due to the definition of the bundle. If a constituent is single among sets, it cannot be a part of the bundle, and thus omitted. Conversely, if set is of length one, it will also be omitted. It is now natural to derive the following table by iterating over values of z. This way intersections of sets and the constituents associated with them are obtained.

$\cap z$	4 3 2	4 3 1	4 3	4 2	4 1	3 2	3 1	2 1
c	11 12	9 10	9 10 11 12	7 8 11 12	7 9 10	3 4 11 12	2 4 9 10	4 7

Notice that some of the possible bundles are intentionally omitted. There is a possibility that a singled-out bundle, for example $(9\ 10\ 11)$, could be a part of the best solution, as opposed to $(9\ 10\ 11\ 12)$ or $(9\ 10)$ that are most probably approximate to the best solution. However, family of bundles is intentionally being limited to what's possible to derive from table similar to the **Table 4**. This way, one-to-one relationship between set intersections and bundles is maintained, making it easier to implement the algorithm.

Proceeding with the greedy algorithm, consider the fact that choosing one bundle over another cannot come down to a comparison of their LE costs alone. It is obvious that algorithm's performance would suffer from predisposition to choose shorter bundles. Along with an intuition that more repetitions of a bundle correlate with its value to the solution, this fact motivates an update for the "feature" part of the algorithm. Formalization of this understanding leads to the formula (2):

$$\rho = \frac{\text{count}(bundle) \cdot \text{count}(z\ with\ bundle)}{LEn_k\big(\text{count}(bundle)\big)} \tag{2}$$

where $z\ with\ bundle$ denotes all sets where the specified bundle occurs, $\text{count}(.)$ determines the number of elements in some enumeration, LEn_k is formula (1). Density function reflects how beneficial a bundle would be to the solution

Having formula (2), it makes sense to move on to the "choice" part of the algorithm, as can be seen in **Table 5**. There, information is based on **Table 4**, and k is assumed to be 3.

Table 5 – Intersections of sets, constituents and density associated with them

$\cap z$	4 3 2	4 3 1	4 3	4 2	4 1	3 2	3 1	2 1
c	11 12	9 10	9 10 11 12	7 8 11 12	7 9 10	3 4 11 12	2 4 9 10	4 7
ρ	$2 \cdot 3/1$	$2 \cdot 3/1$	$4 \cdot 2/2$	$4 \cdot 2/2$	$3 \cdot 2/1$	$4 \cdot 2/2$	$4 \cdot 2/2$	$2 \cdot 2/1$

Finally, there is a reliable way to choose a bundle – one of the bundles with the highest density should be chosen.

As for the "iteration and exit". Guided by the definition of the bundle and by the fact that intersections are prohibited, for the bundle that became a part of the solution all its constituents are excluded from the sets. Then, table similar to table 5 is constructed and the next bundle is chosen. Process continues until it is impossible to find at least two sets with at least two shared constituents.

Thus, all the necessary theory is laid out, and definition of the greedy algorithm is concluded.

RESULTS

Based on the theoretical work carried out, a computer program is created in the Python programming language that allows to quickly find bundles that should be used to minimize LE usage. Full implementation of the program is available at the https://github.com/Gleb-05/MakeFuncForDC. Example of using this program with data from **Table 1** can be seen in **Fig. 1** below. There, notice how bundles of the programmatically found solution differ from those in the manually found ones. Yet the final LE cost, if converted, is still 12.

To check the effectiveness of the developed program, a test was conducted. The algorithm was used 1000 times with pseudo-randomly created sets of four eight-term functions. Final distribution for the number of LEs used can be seen on **Fig. 2**. It should be compared to the default number of LEs needed for the set of four eight-term functions, which is 16. Using the diagram in **Fig. 2**, an "expected gain" can be calculated as follows:

$$\frac{1}{1000}(3 \cdot 7 + 19 \cdot 6 + 175 \cdot 5 + 579 \cdot 4 + 224 \cdot 3)$$

For this particular test, expected gain turned out to be 3.998. Which means that the coefficient of expected improvement is $(16 - 3.998)/16 \approx 0.75$. Practical significance lies in the fact that, using this algorithm, arbitrary sets of four functions are expected to be implemented with just 0.75 of the initial LE cost.

Along with the greedy algorithm, a program implementation of brute force has been drafted for comparison. Consider that for a function with m terms, there is $m!$ permutations if the placement of LEs is ignored. And with N functions it would be necessary to evaluate approximately $(mean(m_i)!)^N$ different cases. Even for the example outlined at the beginning, number of combinations approaches $(7!)^4 \approx 10^{14}$. Assuming that one combination takes 10^{-6} seconds to be

processed, which is unlikely fast, entire brute force can be expected to run for over three years. For this reason, current version of the brute force is considered insufficient.

An average time of work for greedy algorithm turned out to be 0.0024 second. Brute force, on the other hand, expectedly took around four hours to process just one permutation of functions.

Absence of relevant algorithms prevents more rigorous testing. In particular, it remains unproven whether the solutions proposed by this implementation of the greedy algorithm are optimal. A speculation can be held about how close those results come to the truly optimal solution and, if sufficiently so, then how often.

```
functions z and terms c that define them
 4 : ['5', '6', '7', '8', '9', '10', '11', '12']
 3 : ['1', '2', '3', '4', '9', '10', '11', '12']
 2 : ['0', '3', '4', '7', '8', '11', '12', '15']
 1 : ['2', '4', '7', '9', '10', '14']

terms c and functions z where they are used
 2 : ['3', '1']
 3 : ['3', '2']
 4 : ['3', '2', '1']
 7 : ['4', '2', '1']
 8 : ['4', '2']
 9 : ['4', '3', '1']
 10: ['4', '3', '1']
 11: ['4', '3', '2']
 12: ['4', '3', '2']

---one or no z remains, nothing to pair with, exit

> function 4 used bundles: [['7', '9', '10'], ['8', '11', '12']]
> function 3 used bundles: [['2', '4']]
> function 2 used bundles: [['8', '11', '12']]
> function 1 used bundles: [['7', '9', '10'], ['2', '4']]
With 3 bundles and 3 logic gates used


Total LE solution cost: 12
Original cost: 15
```

Fig. 1 – Example of full output after executing the program

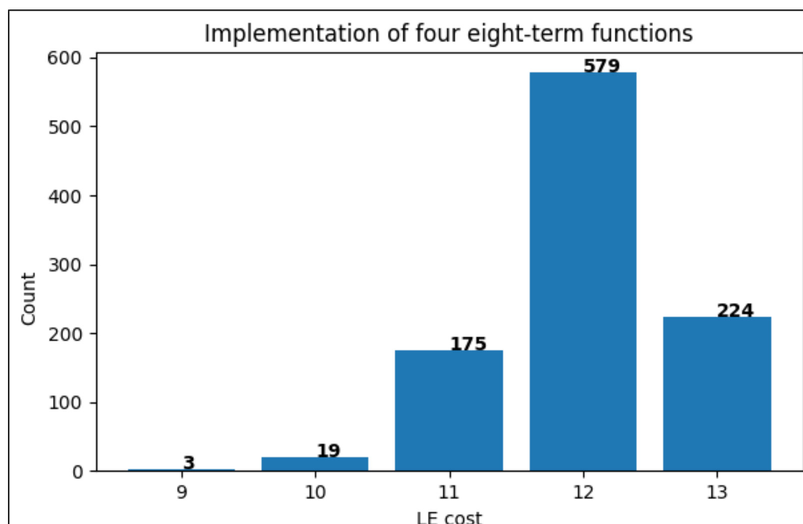

Fig. 2 – Results of testing the greedy algorithm 1000 times

There are more applications for the program beyond the obvious implementation of Boolean functions. It is possible to represent arbitrary real-life problems as z_i sets, the tasks included in them as m_i constituents, and maximum number of tasks possible to accomplish at a time as k. Then, it is possible to use the developed program for task management. Restriction for groups in the final solution to not intersect proves to be of value. It reflects the reality that some tasks are too resourceful and should not be repeated. A good example would be matrix multiplication, used to specify some

transformation of space. Suppose there is one transformation represented with a matrix product 1 2 3 4 5, and another transformation represented with 1 2 3 6 7. Suppose then that the computer processor cannot handle more than three matrices at the same time. Application of the developed program would lead to the conclusion that the product of 1 2 3 can be computed once, thus saving some computer resources.

CONCLUSIONS

Scientific novelty of the research is as follows. Theory behind LE use optimization in context of circuits is laid out for the first time. A number of approaches and formulas essential to the solution is proposed. Among them is a function for calculating quantifiable “density” for any given group of constituents. Based on its value, parts of the solution are selected in iterations of the greedy algorithm, until the whole solution is obtained.

Practical significance of the study is as follows. Based on the research conducted, a computer program is successfully developed in the Python programming language. A common configuration of four eight-term functions is analyzed. The expected coefficient of improvement is 0.75, which means that with the help of the developed program, it is expected to save a quarter of the initial LE cost. Such savings are very important on an industrial scale, particularly in household appliances. Expected time of program execution is 0.0024 seconds. In contrast, the brute force search is expected to run for at least three years. Also, an ability emerges to test the functionality of a circuit more cheaply, which has a positive effect on experimental circuit development. Moreover, since the program essentially offers workload management, its possible applications go beyond the domain of digital electronics. Arbitrary problems can be represented as sets, with tasks comprising them as constituents, where the maximum size for the group of tasks is predefined.

The prospect for further research is developing a program for generalized and broadened versions of the problem, where restrictions specified in the problem statement do not apply. It would also be valuable to evaluate the complexity of the solution in terms of big-O notation. Possible route of development is to compare results of using the program with other similar programs, or develop a sufficient enough brute force algorithm and compare costs of truly optimal and almost optimal solutions.

REFERENCES

1. R. J. Tocci, “Decoders,” in *Digital Systems – Principles and Applications*. New Jersey: Prentice-Hall Inc., 2006.
2. S. Liang, K. Alanazi and K. Al Hamoud, “Set covering problem,” in *Cornell University Computational Optimization Open Textbook*. Cornell University, [online document], 2020. Available: https://optimization.cbe.cornell.edu/index.php?title=Set_covering_problem. [Accessed: February 10, 2024].
3. S. S. Skiena, *The Algorithm Design Manual*. Springer: Nature New York Inc, 2020.
4. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W. H. Freeman, 1979.
5. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Eds., “16 Greedy Algorithms,” in *Introduction to Algorithms*. Cambridge MA: MIT, 2001.
6. "Loss Function," *Wikipedia: The Free Encyclopedia*, March 7, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Loss_function. [Accessed: January 15, 2024].