Optimization of arithmetic units for embedded computing

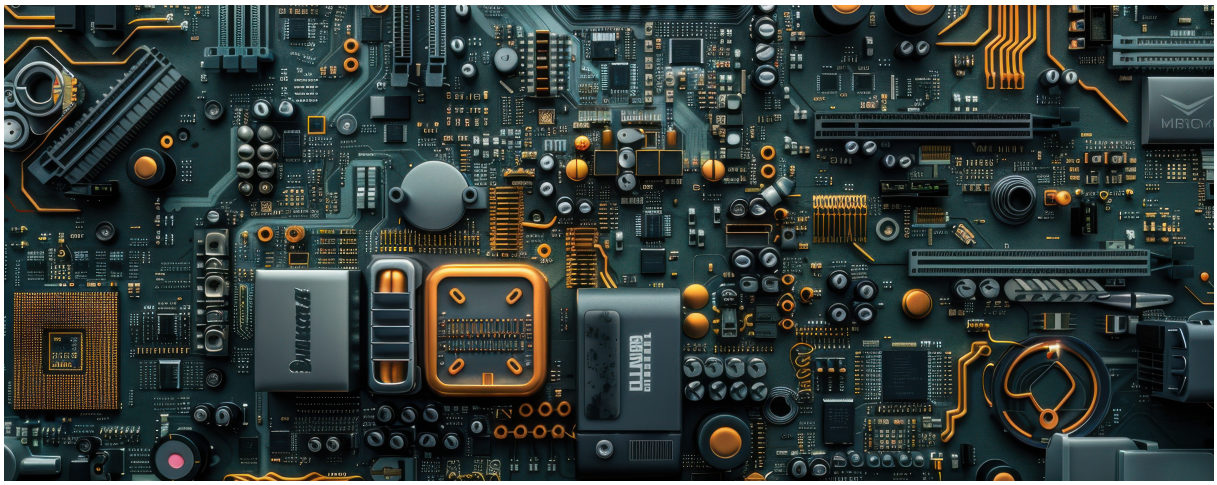
CANTALOUBE Théo

August 29, 2025

Abstract

This thesis focuses on the Multiple Constant Multiplication, a conjectured $\mathcal{NP}$-hard problem consisting in finding the optimal shit-and-add implementation of an arithmetic circuit multiplying the input by given integer target constants and whose utility lies in arithmetic for digital signal processing, among others. Whereas state-of-the-art approaches until now were relying on Linear Programming (LP) and Sat solvers (SAT), we use the Constraint Programming (CP) optimization paradigm which handles well combinatorial aspects of this problem. As soon as instances become complicated, our model far outperforms ILP and is very competitive with SAT.

A second modeling was conceived based on the resolution of Multiple Constant Multiplication knowing the topology of the adder graph. In order to implement this idea, an adder graph canonical code enumeration process through Constraint Programming was constructed.



Contents

[1] Photo credits: `https://www.inria.fr/fr/centre-inria-de-lyon`, `https://www.insa-lyon.fr/fr`, `https://www.citi-lab.fr/`, `https://fr.freepik.com`.

1 Introduction

In our modern world, electronic devices with embedded computations are everywhere: in hearing-aids, in smartwatches, in cars, in drones... The strict resource and power constraints call for highly optimized implementations. Instead of using standard data formats and generic arithmetic units, which can fit to any computations but generates extra cost, hardware designers often use code generators to implement individual or compound operators tailored to the application's requirements.

In this context, a common type of arithmetic operator is Multiplication by Multiple Constants (MCM), where an integer number is multiplied by several integer target constants known at the design time. The idea is that by exploiting the a priori knowledge of the constants, one can avoid costly generic multiplier circuits and design a custom optimized computational implementation for a given target constant set.

MCM is used in digital signal processing for linear digital filter evaluation [1, 19, 27, 31, 37], Discrete Transforms (Discrete Cosine Transform, Fast Fourier Transform) [21, 46], as well as for inference of Deep Neural Networks [16, 24]. These applications will not be studied in this thesis.

Multiplications by constants are commonly done using additions/subtractions: with 1-bit half adders, and using bit-shifts: shifting a number X by k bits on the left (resp. on the right) is equivalent to multiply X by 2^k (resp. 2^{-k}). This is called a *shift-and-add* approach. Shifts are basically wiring that is free in hardware, hence designers usually only minimize the number of additions/subtractions. This is a good high-level metric of the final hardware cost, though other low-level ones that count logic gates exist.

Let us first illustrate our problem on the Single Constant Multiplication (SCM) problem, an easier variation of MCM where there is only one integer target constant. Consider the multiplication of an integer variable X by 93, written as 1011101 in binary. Hence, $93X = 2^6X + 2^4X + 2^3X + 2^2X + X$ and a naive implementation of a shift-and-add approach for $93X$ detailed in Fig. 1 is:

$$5X = X + 2^2X; \quad 13X = 5X + 2^3X; \quad 29X = 13X + 2^4X; \quad 93X = 29X + 2^6X$$

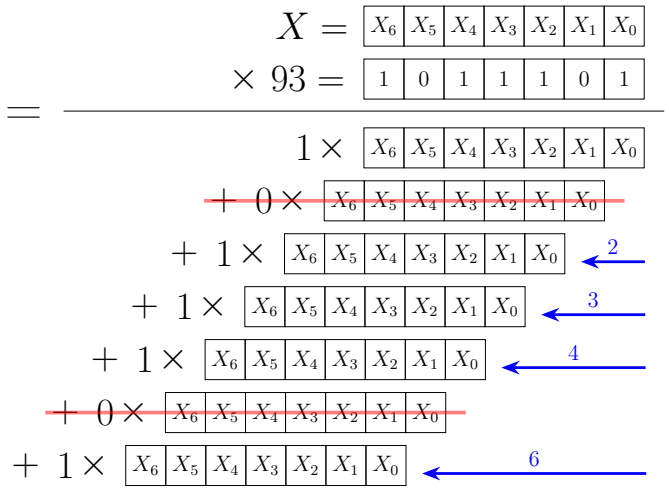


Figure 1: A shift-and-add approach to obtain $93X$

The corresponding shift-and-add graph is displayed in Fig. 2a. Blue arrows correspond to the value of the shifts. Therefore, the number of additions of such an implementation corresponds to the number of non-zero digits in the binary representation of the constant, that is to say the Hamming weight of the constant [25], minus one.

However, this idea does not take into account the possibility of making subtractions instead of additions. So we introduce Signed Digit (SD) representations of binary numbers which uses digits $\{\bar{1}, 0, 1\}$ where $\bar{1} = -1$ [26, 44]. The *Canonical Signed Digit* (CSD) representation of a binary number is an SD representation of this number in which nonzero bits are not adjacent. It can be shown that CSD exists, is unique, minimizes the number of nonzero bits among all SD representations and can be obtained in a polynomial time. The CSD representation of 93 is $93_{dec} = 10\bar{1}00\bar{1}01_{CSD}$. Hence $93X = 2^7X - 2^5X - 2^2X + X$ so a second implementation of a shift-and-add approach for $93X$ is:

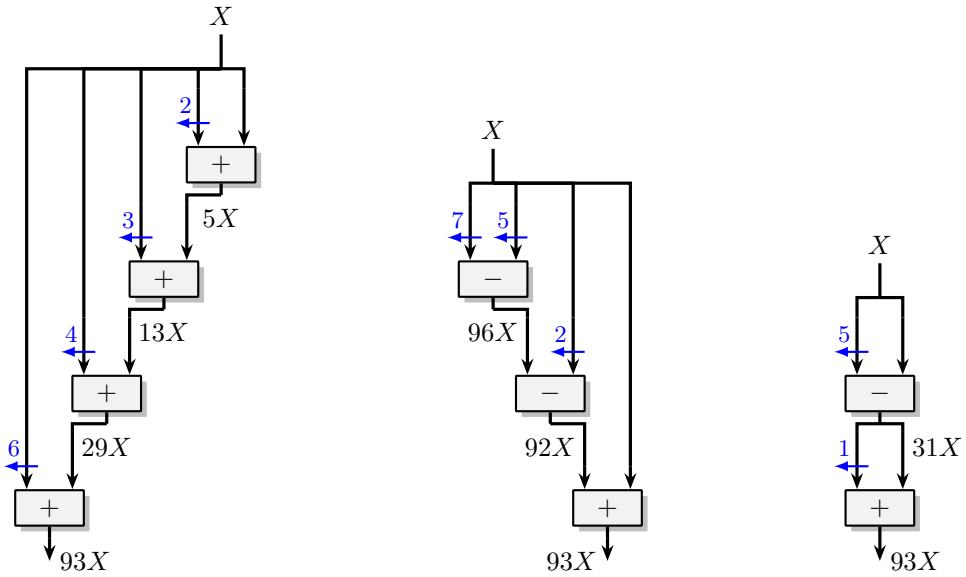
$$96X = 2^7X - 2^5X; \quad 92X = 96X - 2^2X; \quad 93X = 92X + X$$

The corresponding shift-and-add graph is displayed in Fig. 2b.

Unfortunately, if the optimality is the minimization of the number of additions, neither of these approaches is guaranteed to be optimal. Indeed, here is an optimal implementation:

$$31X = 2^5X - X; \quad 93X = 2^1(31X) + 31X \tag{1}$$

The corresponding shift-and-add graph is displayed in Fig. 2c. You can notice that previous approaches are suboptimal because as we define them, we force every addition to have X as one of the inputs, therefore forbidding additions between two previously computed multiplications.



(a) Binary approach for $T = \{93\}$ (b) CSD approach for $T = \{93\}$ (c) Optimal adder graph for $T = \{93\}$

Figure 2: Comparison of naive approaches to optimal implementation of Single Constant Multiplication

In MCM, not only one but several constants must be multiplied by the same input X. For example, consider the set of target constants $\{7, 19, 93\}$. Let say that we know the optimal SCM implementation for each constant. A naive implementation of the multiple constant multiplication could consist in producing individually each target constant. Optimal SCM implementations for $7X$, $19X$ and $93X$ are:

$$\begin{aligned} 7X &= 2^3X - X \\ 15X &= 2^4X - X; \quad 19X = 2^2X + 15X \\ 31X &= 2^5X - X; \quad 93X = 2^1(31X) + 31X \end{aligned}$$

The corresponding shift-and-add graph is displayed in Fig. 3a.

Unfortunately, if the optimality is the minimization of the number of additions, this approach is not guaranteed to be optimal. Indeed, here is an optimal implementation:

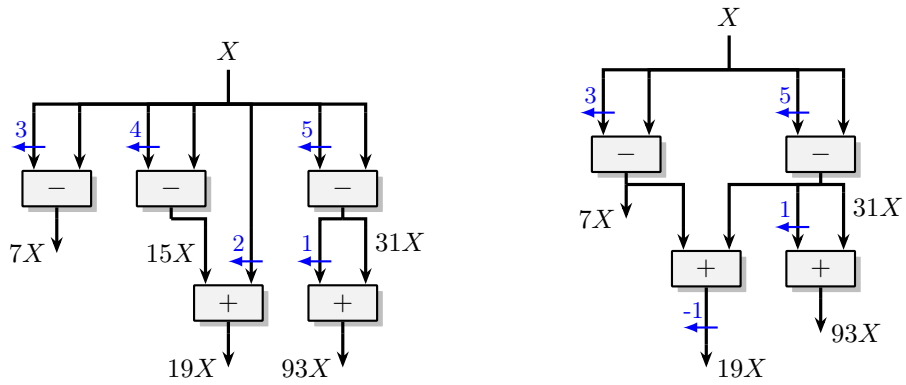
$$7X = 2^3X - X; \quad 31X = 2^5X - X; \quad 19X = 2^{-1}(7X + 31X); \quad 93X = 2^1(31X) + 31X$$

The corresponding shift-and-add graph is displayed in Fig. 3b.

You can notice that the previous approach is suboptimal since as we define it, we forbid intermediate terms to actually be shared. Note that here we used a right-shift to divide the even value $38X$ by 2 to obtain $19X$ so it is important to allow right-shifts (or "negative shifts").

From these motivation examples, we should keep in mind two aspects of the MCM problem:

- Heuristic approaches previously described do not compute optimal solutions. Actually, MCM is said to be $\mathcal{NP}$-hard (even if the existing demonstration of $\mathcal{NP}$-completeness can be questioned, see sect. 6.1), so this justifies why we need declarative and exact approaches.
- There are two complexities around the MCM problem: the first concerns the selection of intermediate results, and the second relates to the sharing of computations among the production of different target constants.



(a) Single constant approach for $T = \{7, 19, 93\}$ (b) Optimal graph for $T = \{7, 19, 93\}$

Figure 3: Comparison of naive approaches to optimal implementation of Multiple Constant Multiplication

As for the vocabulary:

- Constants to produce are called *target constants*.
- An addition or subtraction unit is called an *adder*.
- Shit-and-add graphs are called *adder graphs*.
- Intermediate values in adder graphs are called *fundamentals* (including the target constants).

Example 1.1. *Let us consider the adder graph in Fig. 3b. It has 4 adders. The target constants are* 7*,* 19 *and* 93 *and the fundamentals are* 7*,* 31*,* 19 *and* 93*.*

By convention and from now on, the X of the input is not represented in adder graphs. Then, we consider that the original fundamental is 1.

Given a set of target constants, the MCM Problem aims at finding a shift-and-add implementation that produces every target constant with fundamentals written on a given number of bits and minimizes a given metric. There are a lot of non-exclusive variations of MCM relying on different metrics used as objective functions to minimize:

- **MCM-Adders**: Number of adders [30, 34]. It is the most basic metric on adder graphs, and it is supposed to minimize the resources required to compute the target constant set.
- **MCM-Bits**: Number of bit-adders [17]. A bit-adder is an adder working on 1-bit additions. It is a low-level version of MCM-Adders.
- **MCM-Depth**: Maximum adder-depth. In simple words, the adder depth of an adder counts the maximum path length from the original fundamental to it. By minimizing the maximum adder depth, supposing that all additions are realized in parallel, we reduce the latency of an adder graph (time needed to go through the circuit, i.e. to produce every target constant).
- **PMCM**: Pipelined adder graphs [29, 32, 33]. This is a standard way to increase the throughput of the circuit. Given a fixed adder graph, we seek to pipeline it efficiently using a fixed amount of registers.
- **tMCM**: Number of truncated adders [11, 17, 18]. Sometimes, it is reasonable to truncate unnecessary bits to save computer resources. Then, obtained target constants are known with a given and acceptable precision.
- **RCM**: Reconfigurable Constant Multiplication [38,39]. RCM is applied when the user faces different exclusive target constant set scenarios. Adders have another input integer value and their behavior depends on this input (it corresponds to the scenario).
- **VLCM**: Very Large Constant Multiplication [3]. VLCM is applied when the desired target constant set contains very few but very large constants, like in cryptographic protocols.

Actually, most of these metrics do not perfectly reflect the hardware cost of adder graph implementation. For example, in thesis of [17], the correlation between MCM-Adders and MCM-Bits metrics with the number of Look-Up-Tables (hardware equivalent of adders) and the power consumption was easily greater than 0.95 but with the delay difficultly reached 0.45. In a sense, it should seem logical that it is difficult to find a "perfect metric" since we are trying to apply a high-level interpretation to very low-level behavior, when in essence we are comparing quite different things. In this thesis we will only focus on the simplest metric, the minimization of number of adders.

It is commonly accepted that MCM is an $\mathcal{NP}$-hard problem from [9]. Historically, first papers to address the MCM problem were using heuristics, most of the time based on greedy algorithms [2,5,14,33]. Then Integer Linear Programming (ILP) models were implemented, first by tabulating all possible factors that may occur in an optimal solution in preprocessing computations [2,23,28]. However, the size of these number tables grows exponentially with the word-length of target constants, then leading to models requiring highly demanding calculations.

More recent ILP models approached the MCM problem by initializing a lower bound k to the number of target constants, and by iteratively solving the decision problem "Can the MCM be solved with exactly k adders?" until reaching the smallest k for which the answer is yes [30]. This iterative approach will be called "outer loop" process. Instead of solving a sequence of decision problems, [17, 30] introduced a model that directly minimizes the number of adders (while initializing k to an upper bound computed with a good heuristic). Due to the linearization necessity implied by the use of ILP models, these methods suffer from a lack of expressiveness, which translates into really complicated models and also undergo numerical instabilities when the target constants become large.

In the recent paper [15] was proposed a SAT model solving the MCM problem using an outer loop process. Finally, in [7] were introduced SAT models minimizing the number of full/half adders, another lower-level metric, to solve the SCM Problem. Nevertheless, in this paper, we propose for the first time an approach for the MCM problem from the perspective of Constraint Programming (see Sect. 3.1).

2 Problem statement and properties

Before getting into the details, we need to formally define optimal MCM implementations.

An adder graph is a directed acyclic graph with a single root node (without precedessors) such that each node but the root is associated with an adder and has exactly 2 parents nodes. We note N the number of adders, and adders are numbered from 1 to N.

For each adder $a \in [\![1, N]\!]$, we introduce 7 other integer values:

- c_a as the fundamental of adder a
- $c_{a,l}$ and $c_{a,r}$ as the left and the right operand of the operation of adder a
- $s_{a,l}$ and $s_{a,r}$ as the shifts applied to the left and the right inputs
- $\sigma_{a,l}$ and $\sigma_{a,r}$ as the signs of the left and right inputs

The values must satisfy the fundamental adder relationship:

$$c_a = (-1)^{\sigma_{a,l}} 2^{s_{a,l}} c_{a,l} + (-1)^{\sigma_{a,r}} 2^{s_{a,r}} c_{a,r}$$

The root is noted 0 and we define $c_0 = 1$.

The optimal MCM implementation for the target constant set T and the word-length w is an adder graph and associated variables $(c_a, c_{a,l}, c_{a,r}, s_{a,l}, s_{a,r}, \sigma_{a,l}, \sigma_{a,r})$ respecting the fundamental adder relationship such that:

(i) N is minimal

(ii) $\forall\ j \in [\![1, |T|]\!],\ \exists\ a \in [\![1, N]\!] \mid c_a = T_j$ where T_j is the j-th target constant

(iii) $\forall\ a \in [\![1, N]\!],\ \log_2(c_a) \leq w$

Then we can formally define the problem MCM-Adders:

Definition 2.1. Problem: MCM-Adders(T, w)
Input: *Target constant set T and word-length w*
Output: *Optimal MCM implementation for T and w*

2.1 Reducing the search space

As explained in the previous section, the first intuitive modeling of adder constraints respects the following relationship:

$$c_a = (-1)^{\sigma_{a,l}} 2^{s_{a,l}} c_{a,l} + (-1)^{\sigma_{a,r}} 2^{s_{a,r}} c_{a,r}$$

These notations are summed up in Fig. 4a.

Example 2.1. *Let us consider in Fig. 3b the adder producing* 7. $7 = 2^3 - 1$ *so* $c_a = 7$, $c_{a,l} = c_{a,r} = 1$, $s_{a,l} = 3$, $s_{a,r} = 0$, $\sigma_{a,l} = 0$ *and* $\sigma_{a,r} = 1$. *On the other hand, let us consider the adder producing* 19. $19 = 2^{-1}(31 + 7)$ *so* $c_a = 19$, $c_{a,l} = 7$, $c_{a,r} = 31$, $s_{a,l} = s_{a,r} = -1$ *and* $\sigma_{a,l} = \sigma_{a,r} = 0$.

We can simplify this relationship by considering the following theorem [14]:

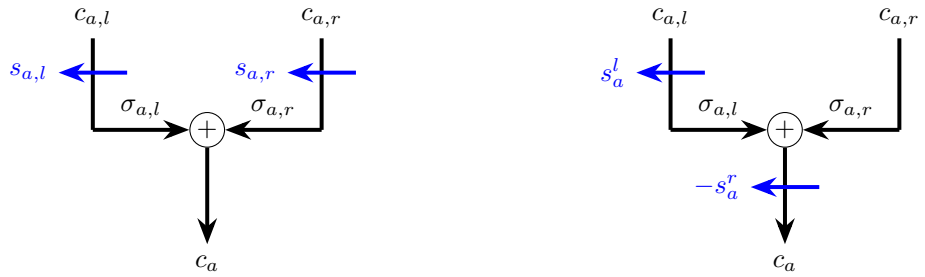
Theorem 2.1. *In odd fundamental graphs, we can consider only the following shift values:*

- *either* $s_{a,l} > 0$ *and* $s_{a,r} = 0$, *i.e., if the left input shift is positive, there is no right input shift*
- *or* $s_{a,l} = s_{a,r} < 0$, *i.e., if the left input shift is negative, the right input shift is negative and equal to the left input shift.*

The choice of applying the left shift (if any) to the left input instead of the right is completely arbitrary. Thanks to these properties, we obtain a simpler modeling of the relationship between a fundamental and the previous ones:

$$c_a = 2^{-s_a^r}((-1)^{\sigma_{a,l}} 2^{s_a^l} c_{a,l} + (-1)^{\sigma_{a,r}} c_{a,r}) \tag{2}$$

Where s_a^l and s_a^r are the left shift applied to the left input and the right shift applied to the sum of the left and right inputs. These notations are summed up in Fig. 4b.



(a) Intuitive modeling of adders, with $s_{a,l}, s_{a,r} \in \mathbb{Z}$ (b) Optimized modeling of adders, with $s_a^l, s_a^r \in \mathbb{N}$

Figure 4: Different modeling for the adders

Example 2.2. *Let us consider in Fig. 3b the adder producing* 7. $7 = 2^3 - 1$ *so* $c_a = 7$, $c_{a,l} = c_{a,r} = 1$, $s_a^l = 3$, $s_a^r = 0$, $\sigma_{a,l} = 0$ *and* $\sigma_{a,r} = 1$. *On the other hand, let us consider the adder producing* 19. $19 = 2^{-1}(31 + 7)$ *so* $c_a = 19$, $c_{a,l} = 7$, $c_{a,r} = 31$, $s_a^l = 0$, $s_a^r = 1$, $\sigma_{a,l} = \sigma_{a,r} = 0$.

As you can guess, the search space of adder graphs to explore is very large. One of our main objectives is to reduce it, by not considering equivalent adder graphs, for example. Here are two very useful theorems to reduce the search space [14]:

Theorem 2.2. *We can remove negative fundamentals from the search space.*

Theorem 2.3. *We can remove even-valued fundamentals from the search space.*

Example 2.3. *In Fig. 2c, fundamentals* 31 *and* 93 *are positive and odd.*

Let us define a new type of adder:

Definition 2.2. *A free adder is an adder that is not re-used by any other adders as an input. Formally, free adders are adder graph sink nodes.*

Here is a basic result that optimal adders present:

Theorem 2.4. *In optimal adder graphs, free adders produce target constants.*

Proof. If a free adder was not a target constant, we could remove it from the graph because it is not re-used anywhere. But the adder graph was supposed to be optimal, so this is a contradiction. □

Example 2.4. *In Fig. 2c or Fig. 3b, target constants* 93 *or* 19 *and* 93 *are on free adders. Keep it mind that the converse is false. In Fig. 3b,* 7 *is not produced by a free adder, but it is a target constant.*

Next theorem will focus on the notion of adder depth. As a reminder:

Definition 2.3. *The adder depth of a vertex a* $ad(a)$ *is equal to the length of the longest path from the original fundamental to adder a. Formally, if we denote* $p_{a,l}$ *and* $p_{a,r}$ *as the parents of the adder a, it is recursively defined by:*

$$ad(0) = 0 \quad and \quad \forall\ a \in [\![1, N]\!],\ ad(a) \leftarrow \max(ad(p_{a,l}), ad(p_{a,r})) + 1$$

Example 2.5. *In Fig. 3b, the adder depth of adders producing* 7 *and* 31 *is* 1 *and of adders producing* 19 *and* 93 *is* 2.

We can use the CSD representation of target constants to bound the adder depth of the adders producing target constants. It has been proven in [22] that:

Theorem 2.5. $\forall\ j \in [\![1, |T|]\!]$, *let* $a \in [\![1, N]\!]$ *be the adder such that* $c_a = T_j$.
We have $ad(a) \geq D_{LB}(T_j) \coloneqq \lceil \log_2 S(T_j) \rceil$, *where* $S(T_j)$ *is equal to the number of nonzero bits in the CSD representation of* T_j.

Example 2.6. *Let a be the adder producing* 93 *in Fig. 2c.* $ad(a) = 2$ *and* $93 = 10\bar{1}00\bar{1}01_{CSD}$ *so* $S(93) = 4$ *so* $D_{LB}(93) = \lceil \log_2 4 \rceil = 2$.

Next theorem will focus on the notion of dependence number.

Definition 2.4. *The dependence number of an adder a* $n_{dep}(a)$ *is equal to the number of adders "participating" into the production of a, the number of adders that can reach a via a directed path. Formally, it is recursively defined by:*

$$dep(0) = \emptyset \quad and \quad \forall\ a \in [\![1, N]\!],\ dep(a) \leftarrow \{p_{a,l}\} \cup \{p_{a,r}\} \cup dep(p_l(a)) \cup dep(p_r(a))$$

$$\forall\ a \in [\![1, N]\!],\ n_{dep}(a) \leftarrow \#dep(a)$$

Example 2.7. *In Fig. 3b, the dependence number of adders producing* 7 *and* 31 *is* 1, *of the adder producing* 93 *is* 2 *and of the adder producing* 19 *is* 3.

We finally can use SCM values for target constants to bound the dependence number of the adders producing target constants:

Theorem 2.6. $\forall\ j \in [\![1, |T|]\!]$, *let* $a \in [\![1, N]\!]$ *be the adder such that* $c_a = T_j$.
We have $n_{dep}(a) \geq S_{LB}(T_j) \coloneqq$ *MCM-Adders*$(\{T_j\}, w)$.

Proof. If this inequality was not respected for a given T_j, then, by removing adders not participating in the creation of T_j, we would obtain an SCM implementation producing T_j contradicting the optimality of MCM-Adders. □

Example 2.8. *Let a be the adder producing* 93 *in Fig. 2c.* $n_{dep}(a) = 2$ *and MCM-Adders*$(\{93\}, 6) = 2$.

Free adders, adder depths and dependence numbers are notions that only depends on the topology of adder graph whereas D_{LB} and S_{LB} are notions that only depend on the target constants. Theorems Th. 2.4, Th. 2.5 and Th. 2.6 link adder graph topology with target constants.

These theorems also allow us to get a lower bound on the minimal number of adders:

Theorem 2.7. *MCM-Adders*$(T, w) \geq \max\left(|T|, \max_{j \in [\![0, |T|]\!]} D_{LB}(T_j), \max_{j \in [\![0, |T|]\!]} S_{LB}(T_j)\right)$

Proof. The first operand of the max operator is a consequence of needing an adder output producing each target constant. The second operand of the max operator naturally follows from Th. 2.5 and the definition by recurrence of $ad(a)$ ($ad(0) = 0$ and $ad(a) = x \implies \exists\ k \in [\![0, a-1]\!]\ \mid ad(k) = x - 1$). The third operand of the max operator naturally follows from Th. 2.6. □

To finish this section, Th. 2.7 allows us to justify that solutions of MCM are unbounded:

Theorem 2.8. *The number of required adders for a minimal MCM implementation (respecting to the number of adders) excluding those used to generate the target constants can be arbitrarily big.*

Proof. $\forall\ n \in \mathbb{N}^*$, let $t_n = \frac{4^n-1}{3}$, $T_n = \{t_n\}$ and $w_n = 2n$. $\frac{4^n-1}{3} = \frac{4^n-1}{4-1} = \sum_{i=0}^{n} 4^i = \sum_{i=0}^{n} 2^{2i}$. This is the CSD representation of t_n (each of the bits with value 1 are separated by a bit with value 0) so $S(t_n) = n$ so $D_{LB}(t_n) = \lceil \log_2 n \rceil$ (and we know that t_n fits on $w_n = 2n$ bits). From Th. 2.7, MCM-Adders$(T_n, w_n) \geq \lceil \log_2 n \rceil$. But we only want to count the adders which are not producing a target constant so: MCM-Adders$(T_n, w_n) - |T_n| \geq \lceil \log_2 n \rceil - 1$. Finally, by comparative growth, $(\text{MCM-Adders}(T_n, w_n) - |T_n|)_{n\in\mathbb{N}^*} \underset{n\to+\infty}{\longrightarrow} +\infty$. □

2.2 Target constants

Due to theorems Th. 2.2 and Th. 2.3, every target constant set T is preprocessed before launching the solving process using the operator *odd*:

$$odd(x) = \frac{x}{\max\limits_{n\geq 0}(gcd(x, 2^n))}$$

$odd(x)$ is equal to x divided by the greatest power of 2 which is a divider of x. In simpler words, we divide x by 2 until it becomes odd. Then, we preprocess T by applying it on every target constant:

$$T_{odd} = \{odd(|T_j|), j \in [\![1, |T|]\!]\}$$

Thus, all our preprocessed target constants are positive and odd, if necessary the sign is adjusted later and a shift can be applied to retrieve the original even-valued constant. This corresponds to elements of the integer sequence M2222 [45] (1, 1, 3, 1, 5, 3, 7, 1, 9, 5, 11, 3, 13, 7, 15, 1, 17...). We also remove all 1 occurrences (they can be obtained for free from initial input). In the following of the paper, we will consider every target constant as preprocessed.

On another hand, even if w is supposed to be part of the parameters of MCM-Adders, we generally consider the word length w is set to:

$$w = \max_{j\in[\![1,|T|]\!]} \lceil \log_2(T_j) \rceil$$

Example 2.9. *For* $T = \{7, 19, 93\}$, $w = 7$.

We consider in our models that every fundamental is written on w bits. We cannot decrease w because there must be some fundamentals equal to the target constants so we must have enough bits to write the target constants. However, one natural question would be: Could we save some adders by increasing w? Indeed, it is possible that big fundamentals are necessary in optimal adder graphs. But in the reality of hardware systems, the cost of increasing w is way bigger than the cost of having an additional adder.

We also consider that intermediate values of the sum of inputs before left-shifts are written on $w + 1$ bits. This is not a theoretical result but rather an empirical assumption based on experimentation. Certainly, left-shifts are quite rare in optimal solutions, and when intermediate values do not fit on w bits, it has been observed that they only need one additional bit.

Example 2.10. *In Fig. 3b, if* 31 *was part of the target constants instead of* 93*, in the adder producing* 19*,* 5 *bits would not be enough because* $7+31 = 38 > 31 = 2^5-1$ *but* 6 *bits would be because* $38 \leq 63 = 2^6-1$.

In Tab. 1 are summarized all high-level notations of Sect. 2 that will be re-used throughout this thesis.

3 Solving MCM with Constraint Programming

In Constraint Programming (CP), we rely on predefined constraint types, which serve as fundamental "building blocks". These global constraints encapsulate common combinatorial substructures, making it easier to model complex problems in a declarative way. In Tab. 2 are listed the building blocks we will be using and their semantics, i.e. the mathematical relation that must be satisfied.

For the sake of visibility, we chose to adopt in our models mathematical formulations, but keep in mind that you can reconstruct every constraint written in this thesis with these building blocks.

Name	Notation	Meaning
Mutliple constant multiplication	MCM-Adders : $\mathbb{N}^{\lvert T\rvert} \times \mathbb{N}^* \to \mathbb{N}$	MCM implementation producing T with fundamentals written on w bits minimizing the number of adders
Adder depth	$ad : [\![0, N]\!] \to [\![0, N]\!]$	Longest path from origin to a
Adder depth lower bound	$D_{LB} : T \to \mathbb{N}$	$\lceil \log_2(S(T_j)) \rceil$
Dependence number	$n_{dep} : [\![0, N]\!] \to [\![0, N]\!]$	Number of adders that can reach a
Dependence number lower bound	$S_{LB} : T \to \mathbb{N}$	MCM-Adders(T_j, w)
Odd	$odd : \mathbb{N} \to \mathbb{N}$	Operator diving x until x is odd

Table 1: Summary of high-level notations

Constraint name	**Semantics**
ALL_DIFFERENT$((x_i)_{i\in[\![1,n]\!]})$	$x_i \neq x_j, \quad \forall\, i \neq j \in [\![1, n]\!]$
ALL_DIFFERENT_EXCEPT_0$((x_i)_{i\in[\![1,n]\!]})$	$x_i \neq x_j \neq 0, \quad \forall\, i \neq j \in [\![1, n]\!]$
ELEMENT$(y, (w_i)_{i\in[\![1,n]\!]}, x)$	$y = w_x \quad (x \in [\![1, n]\!])$
ARITHM$(x, \odot, y, R, w)$	$x \ \odot \ y \ R \ w \quad (\odot \in \{+, -, \times\}, R \in \{=, \neq, <, >, \leq, \geq\})$
MOD(x, k, y)	$x \equiv y \mod k$
COUNT$(K, (x_i)_{i\in[\![1,n]\!]}, y)$	$y = \#\{i \in [\![1, n[\![\mid x_i = K\} \quad (\implies y \in [\![0, n]\!])$
IF_THEN(c_1, c_2)	$c_1 \implies c_2$
c.REIFY_WITH(x)	$x = 1 \implies c \quad (x \in \{0, 1\})$
IF_ONLY_IF(c_1, c_2)	$c_1 \iff c_2$
IF_ONLY_IF(x, c)	$x = 1 \iff c$
MEMBER$(x, (a_i)_{i\in[\![1,n]\!]})$	$\exists\, i \in [\![1, n]\!] \mid x = a_i$
TABLE$((x_i)_{i\in[\![1,n]\!]}, T)$	$(x_1, \ldots, x_n) \in T$

Table 2: Equivalence between CP constraints and semantics

3.1 Fixed Number of adders Decision Problem

In this section, we assume that the number of adders is fixed to some integer N.

3.1.1 Variables

For each adder $a \in [\![1, N]\!]$, we introduce variables to represent its operands and output:

- $c_a \in [\![1, 2^w - 1]\!]$ is the output constant;
- $c_{a,l} \in [\![1, 2^w - 1]\!]$ and $c_{a,r} \in [\![1, 2^w - 1]\!]$ are the left and right inputs.

We also introduce c_0, for the initial input, constrained to be equal to 1.

Variables $c_a, c_{a,l}$, and $c_{a,r}$ are related with sign values ($\sigma_{a,l}$ and $\sigma_{a,r}$) and shift values (s_a^l and s_a^r) as defined in Eq. 2. Instead of introducing sign and shift variables, as done in ILP models, we gather them in k_a, $k_{a,l}$, and $k_{a,r}$ variables, which are further gathered in c_a^{nsh}, $c_{a,l}^{sh,sg}$ and $c_{a,r}^{sh,sg}$ variables as displayed below:

$$\underbrace{\underbrace{2^{s_a^r}}_{k_a} c_a}_{c_a^{nsh}} = \underbrace{\underbrace{(-1)^{\sigma_{a,l}} 2^{s_a^l}}_{k_{a,l}} c_{a,l}}_{c_{a,l}^{sh,sg}} + \underbrace{\underbrace{(-1)^{\sigma_{a,r}}}_{k_{a,r}} c_{a,r}}_{c_{a,r}^{sh,sg}} \tag{3}$$

Hence, for each adder $a \in [\![1, N]\!]$, we introduce the following variables to store intermediate results:

- $k_a \in \{2^s \mid s \in [\![0, w]\!]\}$;
- $k_{a,l} \in \{-2^s, 2^s \mid s \in [\![0, w]\!]\}$;

C1: $\forall\ a \in [\![1, N]\!],\ k_a \times c_a = c_a^{nsh}$
C2: $\forall\ a \in [\![1, N]\!],\ k_{a,l} \times c_{a,l} = c_{a,l}^{sh,sg}\ \wedge\ k_{a,r} \times c_{a,r} = c_{a,r}^{sh,sg}$
C3: $\forall\ a \in [\![1, N]\!],\ c_{a,l}^{sh,sg} + c_{a,r}^{sh,sg} = c_a^{nsh}$
C4: $\forall\ a \in [\![1, N]\!],\ k_{a,l} > 0\ \vee\ k_{a,r} > 0$
C5: $\forall\ a \in [\![1, N]\!],\ k_a + k_{a,l} \neq 2\ \wedge\ k_a + k_{a,l} \neq 0$
C6: $T \subseteq \{c_a \mid a \in [\![1, N]\!]\}$
C7: $c_0 = 1$
C8: $\forall\ a \in [\![1, N]\!],\ (\nexists\ a' \in [\![k+1, N]\!] \mid ind_{a',l} = a\ \vee\ ind_{a',r} = a) \implies c_a \in T$
C9: $\forall\ a \in [\![1, N]\!],\ c_{a,l} = c_{ind_{a,l}}\ \wedge\ c_{a,r} = c_{ind_{a,r}}$
C10: allDifferent$((c_a)_{a \in [\![0,N]\!]})$
C11: $\forall\ a \in [\![1, N]\!],\ c_a \equiv 1[2]$

Figure 5: Constraints for the Decision problem, when the number of adders is fixed to N

- $k_{a,r} \in \{-1, 1\}$;
- $c_a^{nsh} \in [\![0, 2^{w+1}]\!]$;
- $c_{a,l}^{sh,sg} \in [\![-2^{w+1}, 2^{w+1}]\!]$ and $c_{a,r}^{sh,sg} \in [\![-2^w + 1, 2^w - 1]\!]$.

Finally, for each adder $a \in [\![1, N]\!]$, we introduce variables for linking the left and right inputs of a with the output of another adder. To ensure that there is no cycle in this dependency relation, the domain of these variables only contains values smaller than a:

- $ind_{a,l} \in [\![0, a-1]\!]$ and $ind_{a,r} \in [\![0, a-1]\!]$.

3.1.2 Constraints

Constraints are listed in Fig. 5.

Constraints C1 to C3 are straightforward consequences of Eq. 3.

As fundamentals cannot have negative values, we know that $k_{a,l}$ and $k_{a,r}$ cannot be both negative. This is expressed by Constraint C4.

As stated in Th. 2.1, whenever there is no right shift, *i.e.*, $k_a = 1$, then there is a left shift, *i.e.*, $k_{a,l} \notin \{-1, 1\}$. This is ensured by Constraint C5.

To be a valid MCM implementation for the given target set T, adders must produce every target constant. This is realized by Constraint C6.

Constraint C7 initializes the origin input.

According to Th. 2.4, free adders produce target constants. First member of the implication of Constraint C8 is equivalent to: "If adder a is a free adder..." and second member is equivalent to "...then it produces a target constant.".

Constraint C9 links the left and right inputs of each adder a with the output of adders $ind_{a,l}$ and $ind_{a,r}$, respectively.

Finally, constraints C10 and C11 ensure that the generated values are all different and odd, respectively.

3.1.3 Search strategy

CP solvers searchs solution by:

(i) selecting an unassigned variable x_i,

(ii) choosing a value v in the domain of x_i,

(iii) propagating the consequences of this choice in the domains of the other variables and recursively solving the subproblem obtained when assigning v to x_i.

Search strategies specify variables and value ordering heuristics for steps (i) and (ii).

Random exploration of the search space in the MCM problem rarely produces valid or meaningful solutions. Fixing the fundamental of the second adder before the first adder generally results in failure.

In contrast, constructing the adder graph incrementally, by adding one adder at a time in a well-defined order, is more likely to yield a valid solution. Thus, we impose a Lower Bound First strategy, and the variable ordering heuristic is detailed in Fig. 6.

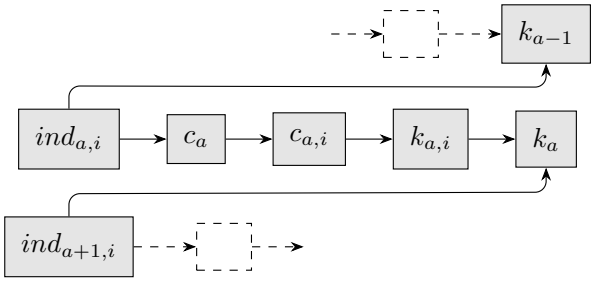


Figure 6: Branching priority for Constraint Programming modeling

3.2 Minimization Problem

To solve the MCM minimization problem, we solve a sequence of decision problems: we initialize N to $|T|$, and then we iteratively solve the model defined in Sect. 3.1 and increment N until we find a solution. Such a process is described in Fig. 7. We call this approach CP outer loop or CP_{OL}.

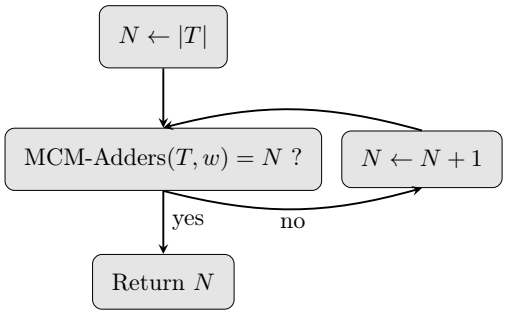


Figure 7: Outer loop process

Another possibility is to add an objective function, corresponding to the number of used adders, and directly solve a minimization problem. We call this approach CP minimization or CP_{min}. In this case, N is no longer the fixed number of adders we are testing but rather an upper bound N of the number of adders that we compute with a simple MCM greedy heuristic, the Bull-Horrocks algorithm (BHA) [8].

CP_{min} has an advantage over CP_{OL} in that it can return intermediate minimization results in the form of valid adder graphs in case it times out. This is not true with Option A (see 3.2.3), because with this search strategy, the first encountered solution is the optimal one.

3.2.1 Variables

For each $a \in [\![1, N]\!]$, we introduce a binary variable u_a such that $u_a = 1$ if adder a is used, and $u_a = 0$ otherwise. Finally, we define the objective function:

$$\min \quad \sum_{a \in A} u_a$$

3.2.2 Constraints

Constraints are listed in Fig. 8.

Constraints C1' to C7' are equivalent to Constraints C1 to C7 defined in Fig. 5, respectively.

Constraint C8 and C9 are modified as it must be ensured only when the adder is used.

Constraint C10 is modified by using the global constraint allDifferentExcept0 [10], instead of allDifferent, as c_a is set to 0 whenever a is not used, as stated in Constraint C12'.

Constraint C11 is modified to ensure that c_a is odd only when adder a is used.

Constraint C12' is meant to make variables u_a consistent with variables c_a.

$\min \sum_{a \in A} u_a$ such that

C1'-C7': See Constraints C1-C7 in Fig. 5

C8': $\forall\ a \in [\![1, N]\!],\ u_a = 1\ \wedge\ \{a' \in [\![k+1, N]\!] \mid ind_{a',l} = a\ \vee\ ind_{a',r} = a\} = \emptyset\ \implies\ c_a \in T$

C9': $\forall\ a \in [\![1, N]\!],\ u_a = 1\ \implies\ (c_{a,l} = c_{ind_{a,l}}\ \wedge\ c_{a,r} = c_{ind_{a,r}})$

C10': allDifferentExcept0$((c_a)_{a \in [\![0,N]\!]})$

C11': $\forall\ a \in [\![1, N[\![,\ c_a \equiv u_a[2]$

C12': $\forall\ a \in [\![1, N]\!],\ c_a = 0 \Leftrightarrow u_a = 0$

C13': $\forall\ a \in [\![1, N-1,]\!],\ u_a \geq u_{a+1}$

Figure 8: Constraints for the minimization MCM Problem

Finally, Constraint C13' breaks symmetries by ensuring that all unused adders have larger indices than the used adders.

3.2.3 Search strategy

We have two options:

A Adding all u_a variables before every other variables involved in the Lower Bound First strategy of Fig. 6. This is more or less equivalent to CP_{OL} (when all u_a are fixed, models from Sect. 3.1 and Sect. 3.2 are identical). It could theoretically have better results than the CP_{OL}, in the sense that between each decision problem, CP_{OL} must "start from 0", whereas with this search strategy, CP_{min} can restart from a partially constructed adder graph.

B Integrating u_a variable into the the Lower Bound First strategy of Fig. 6 before other variables linked to adder a.

We tried both strategies, Option B is surprisingly unefficient in comparison with Option A.

3.3 Table constraints

It is well-known that Constraint Programming does not handle well arithmetic constraints [4, 47]. Table constraints are usually considered in order to avoid arithmetic constraints. The idea here would be to store every possible values $(c_{a,l}, c_{a,r}, c_a)$. To formalize it, we introduce an operator $\mathcal{A}_w : [\![1, 2^w - 1]\!]^2 \to \mathcal{P}([\![1, 2^w - 1]\!])$ defined as follows:

$$\mathcal{A}_w(x, y) = \left\{ z \mid \exists\ (s, \sigma_1, \sigma_2) \in [\![0, w]\!] \times \{0, 1\}^2 \setminus (1, 1) \;\middle|\; \begin{array}{c} 2^s z = (-1)^{\sigma_1} x + (-1)^{\sigma_2} y \\ \text{or}\ \ z = 2^s (-1)^{\sigma_1} x + (-1)^{\sigma_2} y \\ \text{or}\ \ z = (-1)^{\sigma_1} x + 2^s (-1)^{\sigma_2} y \end{array} \right\}$$

$$X_w = \{x \in [\![0, 2^w]\!] \mid x \equiv 1[2]\}$$

$$A_w = \{(x, y, z) \in X_w^3 \mid x \leq y\ \wedge\ z \in \mathcal{A}_w(x, y)\}$$

$z \in \mathcal{A}_w(x, y)$ can be understood as "We can obtain z with x and y as inputs". Then we can add the following redundant table constraint:

$$((c_{a,l}, c_{a,r}, c_a) \in A_w\ \vee\ (c_{a,r}, c_{a,l}, c_a) \in A_w) \quad \forall\ a \in [\![1, N]\!]$$

Unfortunately, this table constraint is impractical for few reasons:

- The complexity of generation of A_w in $O(4(w+1)(2^{w-1})^2)$.
- The size of A_w (for example, for $w = 11$, $\#A_w > 10^6$).

In Fig. 9 is presented the evolution of the size of this table respecting to the wordlength. The lower is the better. As the scale of the y-axis is logarithmic, we can see that it is exponential in w.

One can notice that $\mathcal{A}_w$ is a symmetric operator ($\mathcal{A}_w(x, y) = \mathcal{A}_w(y, x)$). We could make the operator asymmetric by imposing that only x can be left-shifted, exactly as described in Th. 2.1, and only adding the constraint:

$$((c_{a,l}, c_{a,r}, c_a) \in A_w) \quad \forall\ a \in [\![1, N]\!]$$

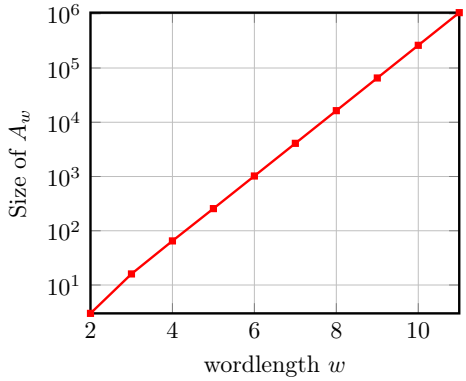


Figure 9: Evolution of the size of A_w respecting to the wordlength w.

Nevertheless, it doesn't make the use of A_w tractable.

Using A_w remains impractical even when we only consider $(x, y, z) \in X^2 \times T$ and apply this table constraint on free adders (when the topology of the adder graph is fixed, see Sect. 5.1).

3.4 Performance results

Our CP models have been implemented in Java with Choco-solver [43]. Even though the solver itself is highly optimized, we should keep in mind that using it through a Java interface introduces an additional overhead due to the encapsulation and abstraction layers inherent to the Java environment. Experiments were conducted on a machine equipped with an Intel® Xeon® Gold 6444Y processor (16 cores, 32 threads, 45 MB L3 cache) with 256 GB RAM, running on an x86_64 architecture with support for AVX-512 and related vector instruction sets. We call these approaches CP_{OL} – Choco and CP_{min} – Choco.

The benchmark we have worked on, presented in Sect. A of the appendix, is extracted from a collection of linear digital filters [40] that can be implemented with the MCM problem. It has been widely used in the literature and for the ILP-based approaches in particular [17, 30, 33]. It contains 83 instances with $|T|$ ranging from 1 to 51 constants and w from 4 to 19 bits. While the benchmark was already solved with previous approaches, those relied on resolution times beyond practical limits. Thus, our objective is not only to solve instances, but also to solve it in a reasonable amount of time.

In Fig. 10, we can see how good BHA [8], the algorithm constructing adder graphs in polynomial-time to rapidly obtain an upper bound on the optimal number of adders, is on the benchmark. Instances are sorted by increasing optimal value. Except for some specific instances, the upper bound provided by BHA is generally within a radius of 3 adders from the optimal number of adders. However, BHA performances variability does not allow us to conclude that we can start directly with the number of adders set to the upper bound before decrementing it. Conversely, we also can have an idea of how close the lower bound provided by the number of target constants is from the optimal number of adders. It is interesting to note that the cases with the greatest variability in the upper and lower limits coincide surprisingly well.

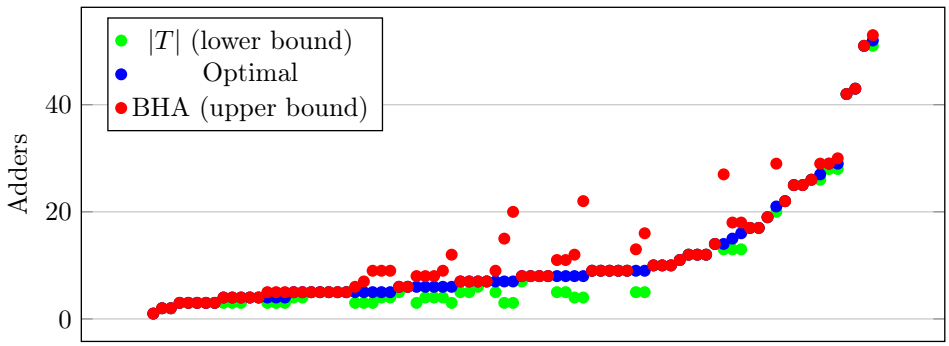


Figure 10: Comparison between optimal and heuristic number of adders of benchmark

In addition to our CP models CP_{OL} – Choco and CP_{min} – Choco, we propose 2 more models using Minizinc. MiniZinc is a high-level constraint modelling language that allows you to easily express and

solve discrete optimisation problems [41]. It provides a unified front-end for expressing combinatorial optimization problems across various paradigms (ILP, CP, SAT), and delegates the solving to a back-end solver via a standardized intermediate format (FlatZinc). As we heard how efficient SAT solvers (directly written in SAT) were for the MCM problem resolution, we used Minizinc with CP models in front-end, PicatSAT [48] in middle-end (to translate CP into SAT) and KisSAT [6] in back-end. We denote as CP_{OL} – PicatSAT and CP_{min} – PicatSAT our CP models with Minizinc solving.

The code written for these models is available as open source[2].

In these performance results, we will compare our models to state-of-the-art models, the minimization ILP model implemented in Julia with Gurobi by [17], denoted as ILP – [17], and the outer-loop SAT model implemented in C++ with Cadical by [15], denoted as SAT – [15]. As a first overall point of view, we can see how much better suited the CP_{OL} approach is to the combinatorial aspects of the MCM problem than the ILP approach by comparing the number of variables in the respective models. In Tab. 3, you can notice that unlike the ILP model, there is no dependence on the number of target constants or on the word-length and the dependence on N is linear.

	ILP – [17]	CP OL – Choco	CP min – Choco
Minimization	Objective function	Outer loop	Objective function
Intermediate results	Yes	No	No - A / Yes - B (3.2.3)
Variables number	$\sim (10 + N + w + \lvert T\rvert)N$	$\sim 12N$	$\sim 13N$
Constraints number	$\sim (10 + 2N + 4w + 2\lvert T\rvert)N + \lvert T\rvert$	$\sim 13N$	$\sim 15N$

Table 3: Comparison between models

In Fig. 11a and Fig. 12a, we compare every model on a per instance basis. Each circle represents an instance. The closer the circles are to the lower-right corner of the graph, the better the result. ILP – [17] and SAT – [15] are faster on "easy" instances, solved in less than 1s by both approaches. However, our approaches are faster on harder instances. While, in a run time limit of 3600s, CP_{OL} – Choco, CP_{min} – Choco, CP_{OL} – PicatSAT and CP_{min} – PicatSAT are respectively able to solve 73, 73, 75 and 59 out of 83 instances, whereas ILP – [17] and SAT – [15] respectively solved 61 and 74 out of 83 instances. CP_{OL} – Choco and CP_{min} – Choco have approximately the same behaviour, whereas CP_{min} – PicatSAT is largerly suboptimal in comparison to CP_{OL} – PicatSAT. As they have the best results, in the following, we will only consider the CP_{OL} (– Choco and – PicatSAT) models instead of the CP_{min} (– Choco and – PicatSAT) models. To give more details, whenever CP_{OL} – Choco or CP_{OL} – PicatSAT resulted with a Time-Out, ILP – [17] resulted with a Time-out and whenever CP_{OL} – PicatSAT resulted with a Time-Out, SAT – [15] resulted with a Time-out.

Figures Fig. 11b and Fig. 12b presents the evolution of the cumulative percentage of solved instances with respect to time. The more rapidly the proportion approaches 1 – that is, the more quickly the curve approaches the upper-left corner of the graph – the better the result. We note that ILP – [17] and SAT – [15] are more successful than CP_{OL} – Choco and CP_{OL} – PicatSAT when the time limit is smaller than 0.1s, whereas SAT – [15], CP_{OL} – Choco and CP_{OL} – PicatSAT is more successful than ILP – [17] for longer time limits.

3.5 Preprocessing step

In this section, we study the decision problem that aims at deciding whether there exists a solution with $|T|$ adders. We show that this problem may be solved in polynomial time, and that the solution process is improved when solving this decision problem during a preprocessing step.

Let us first formally define this decision problem:

Problem: T-OPT

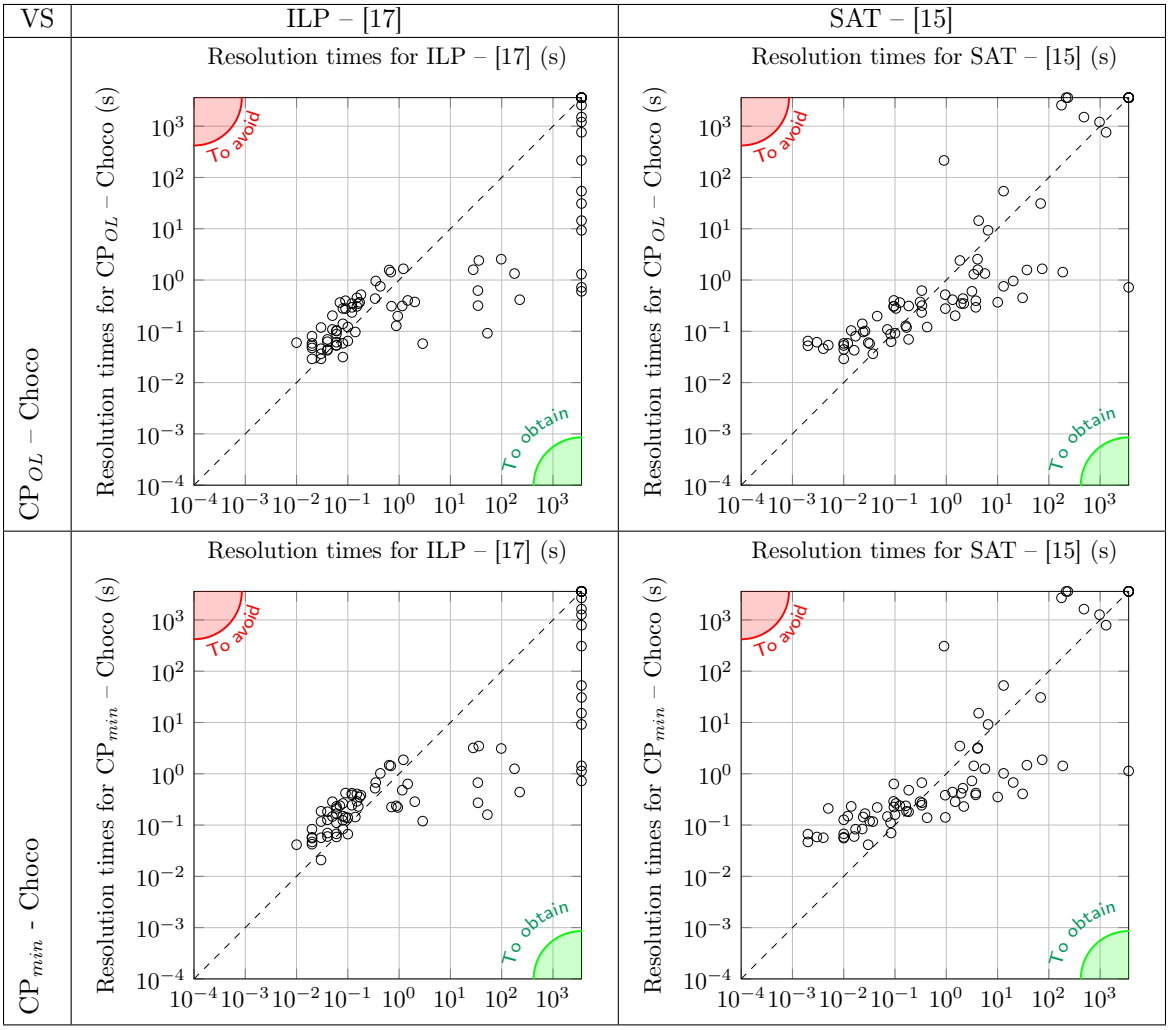
Input: Target constant set T and word-length w

Question: Do we have MCM-Adders$(T, w) = |T|$?

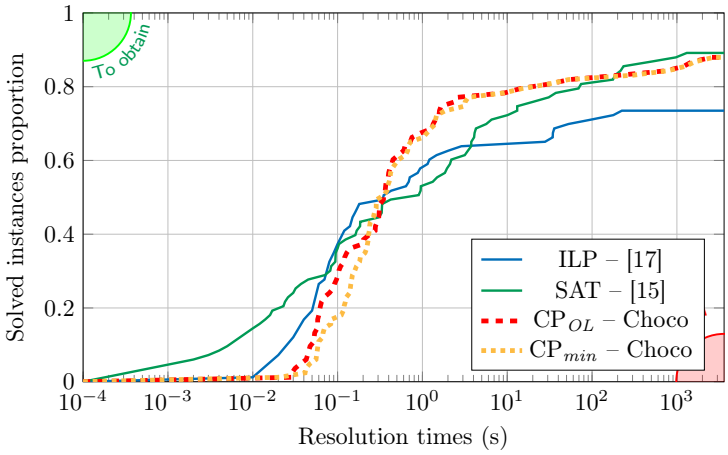
We named it T-OPT because if T-OPT is true, then T is the OPTimal set of fundamentals.

Before delving into the algorithm, we first need to show how to efficiently determine if z can be produced with x and y as inputs (we denote this as $\mathsf{can}(x, y, z)$). Indeed, $\mathsf{can}(x, y, z) = (z \in \mathcal{A}_w(x, y))$

[2] `https://gitlab.inria.fr/emeraude/mcm-cp`

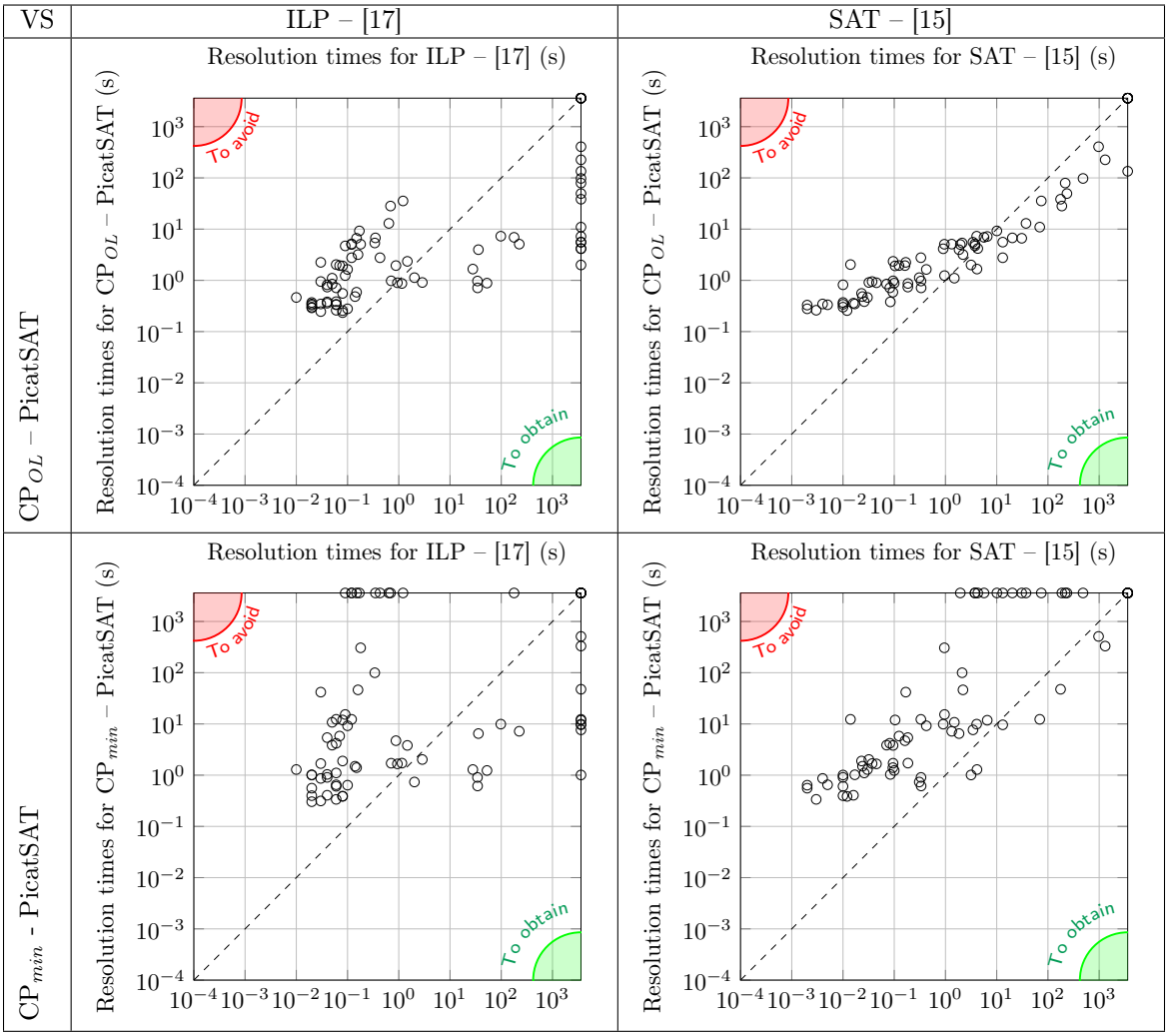


(a) Comparing CP$_{OL}$ – Choco and CP$_{min}$ – Choco to ILP – [17] and SAT – [15]

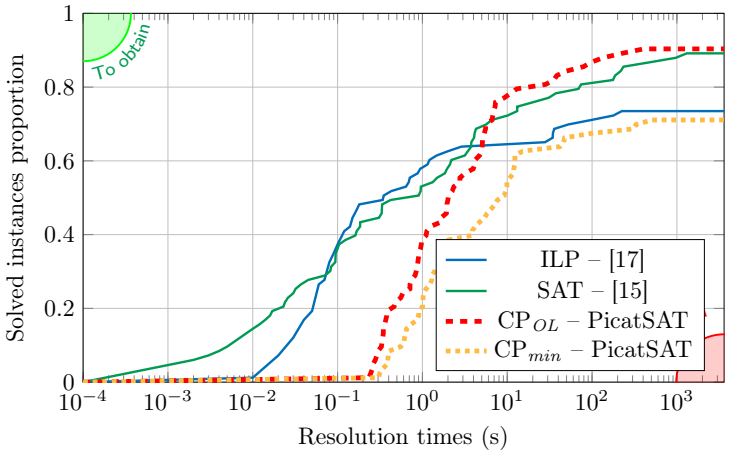


(b) Distribution of the solved instances proportion through time

Figure 11: Comparative analysis of ILP – [17] and SAT – [15] to CP$_{OL}$ – Choco and CP$_{min}$ – Choco resolution times.



(a) Comparing CP$_{OL}$ – PicatSAT and CP$_{min}$ – PicatSAT to ILP – [17] and SAT – [15]



(b) Distribution of the solved instances proportion through time

Figure 12: Comparative analysis of ILP – [17] and SAT – [15] to CP$_{OL}$ – PicatSAT and CP$_{min}$ – PicatSAT resolution times.

but computing $\mathcal{A}_w(x, y)$ is not optimal. Instead, we can use operator *odd* and do:

$$\mathsf{can}(x,y,z) = \left(\exists\ (\sigma_1,\sigma_2) \in \{0,1\}^2 \setminus (1,1) \;\middle|\; \begin{array}{c} z = odd((-1)^{\sigma_1}x + (-1)^{\sigma_2}y) \\ \text{or}\ \ (-1)^{\sigma_1}x = odd(z - (-1)^{\sigma_2}y) \\ \text{or}\ \ (-1)^{\sigma_2}y = odd(z - (-1)^{\sigma_1}x) \end{array} \right)$$

The time complexity for computing can is logarithmic in the number of bits because of the operator *odd* [12, chapter 10]. For an explicit translation of what $\mathsf{can}(x, y, z)$ represents:

$$\mathsf{can}(x,y,z) = true \iff$$

$$z = odd(x+y) \ \text{ or } \ x = odd(z-y) \ \text{ or } \ y = odd(z-x) \ \text{ or } \ z = odd(x-y) \ \text{ or } \ x = odd(z+y)$$
$$\text{or } \ -y = odd(z-x) \ \text{ or } \ z = odd(-x+y) \ \text{ or } \ -x = odd(z-y) \ \text{ or } \ y = odd(z+x) \iff$$
$$\exists\ k \in \mathbb{N}^* \ \mid \ 2^k z = x + y \ \text{ or } \ 2^k x = z - y \ \text{ or } \ 2^k y = z - x \ \text{ or } \ 2^k z = x - y \ \text{ or } \ 2^k x = z + y$$
$$\text{or } \ -2^k y = z - x \ \text{ or } \ 2^k z = -x + y \ \text{ or } \ -2^k x = z - y \ \text{ or } \ 2^k y = z + x \iff$$
$$\exists\ k \in \mathbb{N}^* \ \mid \ z = 2^{-k}(x+y) \ \text{ or } \ z = 2^k x + y \ \text{ or } \ z = x + 2^k y \ \text{ or } \ z = 2^{-k}(x-y) \ \text{ or } \ z = 2^k x - y$$
$$\text{or } \ z = x - 2^k y \ \text{ or } \ z = 2^{-k}(-x+y) \ \text{ or } \ z = -2^k x + y \ \text{ or } \ z = -x + 2^k y$$

We covered every possible combination of x and y to obtain z. One can notice that the function $((x, y) \mapsto \mathsf{can}(x, y, z))$ is a symmetric operator ($\mathsf{can}(x, y, z) = \mathsf{can}(y, x, z)$). Also, in the definition of *can*, $(\sigma_1, \sigma_2) \neq (1, 1)$ is a consequence of Th. 2.2 $(x, y, z > 0)$.

Example 3.1. *Let us use the operator* can *on* $T = \{7, 19, 93\}$ *(cf Fig. 3b).* $\mathsf{can}(1, 1, 7) = true$ *because* $1 \times 2^3 - 1 = 7$ *so* $odd(7 + 1) = 1$. $\mathsf{can}(1, 1, 31) = true$ *because* $1 \times 2^5 - 1 = 31$ *so* $odd(31 + 1) = 1$. $\mathsf{can}(7, 31, 19) = true$ *because* $2^{-1} \times (7 + 31) = 19$ *so* $odd(7 + 31) = 19$. $\mathsf{can}(31, 31, 93) = true$ *because* $31 \times 2^2 + 31 = 93$ *so* $odd(93 - 31) = 31$.

Theorem 3.1. *T-OPT can be solved in pseudo-polynomial time* $\mathcal{O}(log(w) \times |T|^3)$.

Proof. The pseudo-polynomial algorithm is displayed in Alg. 1. The idea is to process adder depth by adder depth and build R, the set of reached target constants, while W is the set of target constants waiting to be processed. We start from 1 and we compute $\mathsf{can}(1, 1, T_j)$, $\forall\ T_j \in T$. We can repeat the process until we have found all target constants, as illustrated in Fig 13.

The algorithm will necessarily terminate because $|T| = \text{MCM-Adders}(T, w)$. The while and for loops on lines 4 and 6 of Alg. 1 run at most $|T|$ times. The time complexity for each can call is logarithmic in the number of bits. Hence, the time complexity of Alg. 1 is $\mathcal{O}(log(w) \times |T|^3)$. □

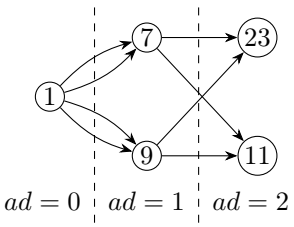


Figure 13: Illustration of Alg. 1 on the toy instance ($T = \{1, 7, 9, 11, 23\}$, $w = 5$). We have $\mathsf{can}(1, 1, 7) = \mathsf{can}(1, 1, 9) = true$ because $7 = 2^3 - 1$ and $9 = 2^3 + 1$. Hence, at the end of the first iteration of the while loop, we have $R = \{1, 7, 9\}$ and $W = \{7, 9\}$. We have $\mathsf{can}(7, 9, 11) = \mathsf{can}(7, 9, 23) = true$ because $11 = 2 \times 9 - 7$ and $23 = 2 \times 7 + 9$. Hence, at the end of the second iteration of the while loop, we have $R = \{1, 7, 9, 11, 23\}$ and $W = \{23, 11\}$. Then $R = T$ so the algorithm will return $prec = \{7 \leftarrow (1, 1), 9 \leftarrow (1, 1), 11 \leftarrow (7, 9), 23 \leftarrow (7, 9)\}$

Without any choice rule when we pop x from W, the algorithm would be non-deterministic. Choosing a proper choice heuristic, which optimizes the number of While loop, would require some further studies, however this algorithm is already very efficient so we arbitrarily adopt a FIFO heuristic.

A straightforward consequence of Th. 3.1 is that any instance (T, w) such that $\text{MCM-Adders}(T, w) = |T|$ may be solved in pseudo-polynomial time. Hence, we propose to run Alg. 1 in a preprocessing step. If the algorithm returns true, then *prec* is a solution and the instance is solved; otherwise we can set the

Algorithm 1: Pseudo-polynomial implementation of MCM on T-OPT instances

Input: Target constant set T
Output: $(A, prec)$ such that A is the answer for T-OPT(T, w) and if A=true, then $prec$ is a predecessor function corresponding to an MCM implementation

```
function MCM-Adders_poly(T)
    Let R = {1} and W = {1} be sets of constants
    while W ≠ ∅ do
        Pop x from W
        forall z ∈ T \ R do
            forall y ∈ R \ W do
                if can(x, y, z) then
                    prec(z) ← (x, y)
                    R ← R ∪ {z}
                    W ← W ∪ {z}
                    Break
        if R = T ∪ {1} then
            return (True, prec)
    return (False, ∅)
```

lower bound on the number of adders to $|T| + 1$ (which in itself is a micro-optimization), and then solve the instance with CP. We call this new approach CP^{T-OPT}.

When running this polynomial algorithm on our benchmark, we noticed that the answer was true for 44 out of the 83 MCM instances. In Fig. 14a, we compare CP_{OL} with CP_{OL}^{T-OPT}. It shows us that the preprocessing step allows us to significantly reduce the solution time for many instances, those for which MCM-Adders$(T, w) = |T|$, whereas solving times are not significantly changed for the other instances (as the running time of Algo. 1 is overwhelmingly negligible compared to the running time of Choco or PicatSAT). As shown in Fig. 14b, that displays the evolution of the cumulative percentage of solved instances with respect to time, we note that CP_{OL}^{T-OPT} – Choco and CP_{OL}^{T-OPT} – PicatSAT are always better than ILP – [17] and SAT – [15]. CP_{OL}^{T-OPT} – PicatSAT is now able to solve 79 out of 83 instances of the benchmark.

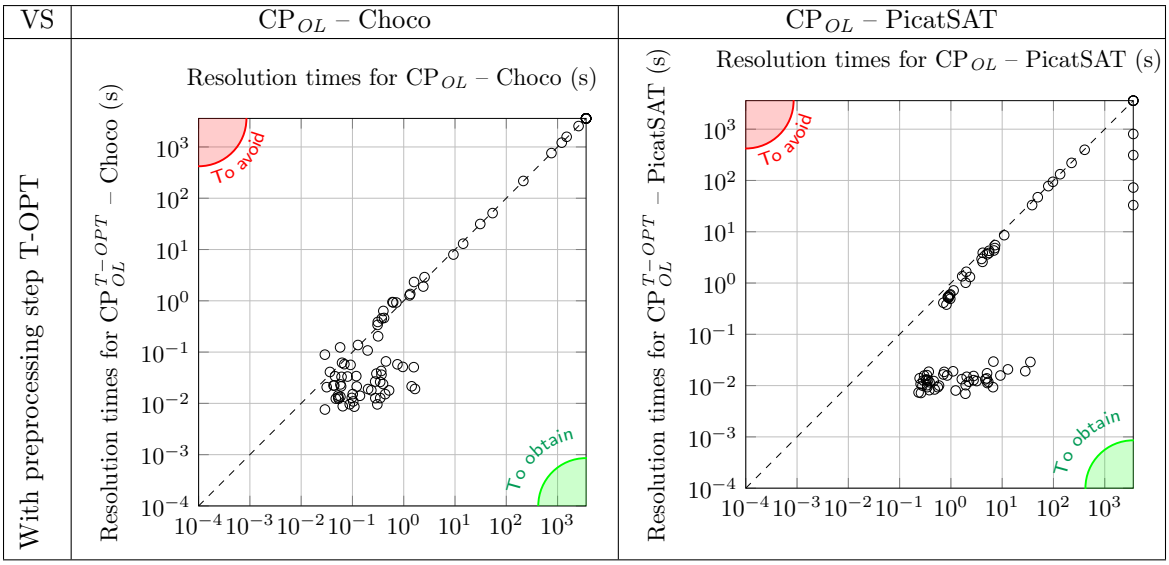
Although the Constraint Programming formulation for MCM minimization was not the main objective of this thesis (the primary objective begins in Sect. 4), the results obtained are nevertheless noteworthy.

4 Adder graph generation

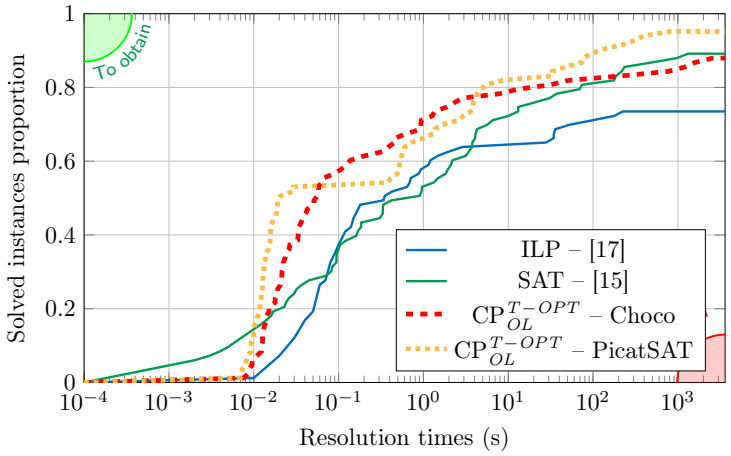
In the second part of this thesis, we want to adopt a new perspective on the solving of the MCM respecting to the number of adders. The main idea is that when we fix the topology — that is, when we impose which adder is connected to which — but let the solver determine the corresponding fundamentals, the problem remains computationally hard, but becomes significantly more tractable. We want to benefit from that by executing the Fixed Topology Decision Problem on every possible topology. The motivation arises from a specific class of instances within our benchmark. These instances involve between three and five distinct target constants and are notably difficult to solve, for example with $T = \{3787, 9943, 15567\}$. Given the small number of targets, one might reasonably expect that only a limited number of adders is required, and consequently, that the space of topologies to explore could be significantly reduced.

As is to be expected, the cost of enumerating every possible adder graph with a fixed number of adders grows very fast, but we can drastically save time by discarding symmetric adder graphs. In order to do that, a classical approach is to choose a non-deterministic algorithm yielding different representations of a graph, attribute a code to each representation and select only the smallest code among all codes possible, called the canonical code. Two graphs will be isomorphic if and only if they share the same canonical code. Hence, this process transforms the enumeration of non-isomorphic adder graphs into the enumeration of adder graph canonical codes.

The choice of the graph representation is not trivial. It can rely on adjacency matrices [36] but in this thesis, we prefer graph traversals such as Breadth First Search [42].



(a) Comparing CP$_{OL}$ – Choco and CP$_{OL}$ – PicatSAT to CP$_{OL}^{T-OPT}$ – Choco and CP$_{OL}^{T-OPT}$ – PicatSAT



(b) Distribution of the solved instances proportion through time

Figure 14: Comparative analysis of ILP – [17] and SAT – [15] to CP$_{OL}^{T-OPT}$ – Choco and CP$_{OL}^{T-OPT}$ – PicatSAT resolution times.

4.1 Adder graph Breadth First Search

As a reminder, here is the definition of an adder graph:

Definition 4.1. *An adder graph is a Directed Acyclic Graph $G = (V, A)$ such that:*

$$\exists\,!\; v_0 \in V \mid d^-(v_0) = 0 \quad \textit{and} \quad \forall\, v \in V \setminus \{v_0\},\ d^-(v) = 2$$

To give an example, left graph of Fig. 15 is an adder graph. Let us consider a few additionnal properties on adder graphs.

Theorem 4.1. *Any adder graph is weakly connected.*

Proof. There is a path from the root to every vertex. If not, because of the condition of non-zero in-degree for each vertex apart from the root, we would obtain a cycle but adder graphs are DAG. □

Theorem 4.2. *Any adder graph can be decomposed into two arborescences.*

Proof. This is trivial from the fact that every vertex has two parents, so we can store each one of the two edges in one of two digraphs. Here again, in each digraph, each vertex apart from the root has an in-degree equal to 1 and there is no cycle, so the digraphs are arborescences. □

This decomposition will be obtained by our search through the graph. We denote $G = P_L + P_R$ where $P_L = (V, A_L)$ and $P_R = (V, A_R)$ are the corresponding arborescences.

A Breadth-First Search of a graph starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. We introduce here an *Adder graph Breadth-First Search* (or BFS_{AG}) which is basically a Breadth-First Search but exploring only the neighbors whose adder depth is one step bigger than the considered vertex. As a reminder, the adder depth can be defined as the maximum distance from the root to the vertex (which is finite because adder graphs do not contain any cycle). For each vertex, there is at least one of its parents whose adder depth is one step smaller because $ad(v) = \max(ad(v_l), ad(v_r)) + 1$ where v_l and v_r are the parents of v. This implies that all vertices will be visited during a BFS_{AG}. All edges discovered through this BFS_{AG} will be stored in P_L and the other edges will be stored in P_R ($P_R = G[A \setminus A_L]$).

A BFS_{AG} code of G is a code generated by Alg. 2, which simply consists in labelling the vertices as we encounter them during a BFS_{AG} of G. Let us review the algorithm step by step:

- p_l and p_r are the parts of the code representing respectively P_L and P_R. We can easily reconstruct P_L (resp. P_R) from p_l (resp. p_r): the vertex whose label is in index k in p_l (resp. p_r) is the parent of the vertex whose label is k in P_L (resp. P_R).

  Example 4.1. *In the first decomposition of Fig. 15, P_L corresponds to the code $p_l = 00123$ because the parent of 1 is 0, the parent of 2 is 0, the parent of 3 is 1, the parent of 4 is 2 and the parent of 5 is 3.*

- $l[v]$ denotes the label associated to vertex v (we do not have problems of non-definition of l because of BFS). l represents a mapping between vertices of V and integers of $[\![1, |V|]\!]$. It is a bijective mapping so sometimes, we will be required to use l^{-1}. However, for the sake of simplicity, we will often indifferently denote v and $l(v)$ or k and $l^{-1}(k)$, even if it is technically incorrect. We will also denote $+$ as the concatenation between two strings.

- Q is the queue of vertices waiting to be popped out and labelled.

- i is the label to attribute to the currently visited vertex.

The "Adder graph" part of the BFS is in Line. 7. We iterate on every neighbor whose adder depth is equal to the adder depth of the current vertex plus one. The resulting code is the concatenation of p_l and p_r. This algorithm is not deterministic because the order of the For loop in Line. 7 is arbitrary.

Algorithm 2: Adder graph Breadth-First Search

Input: Partial adder graph $G = (V, A)$
Output: BFS_{AG} code of G

```
1  function BFS_AG(G)
2      Let p_l and p_r be two empty strings
3      Let Q be an empty queue of vertices
4      l[v_0] ← 0, i ← 1, Q ← {v_0}
5      while Q ≠ ∅ do
6          Pop v_l from Q
7          forall v ∈ V \ Q | (v_l, v) ∈ A ∧ ad(v) = ad(v_l) + 1 do
8              p_l ← p_l + l[v_l]
9              Let v_r be the other parent of v
10             p_r ← p_r + l[v_r]
11             l[v] ← i
12             i ← i + 1
13             Q ← Q ∪ {v}
14     return p_l + p_r
```

Example 4.2. *Let us consider two possible BFS_{AG} codes of a given adder graph G in Fig. 15. In the first decomposition, at iteration 2, we have $Q = \{b, c\}$ as b and c are the children of a, at iteration 3, we have $Q = \{c, d\}$ as d is the child of b, at iteration 4, we have $Q = \{d, e\}$ as e is the child of c and at*

iteration 5, we have $Q = \{e, f\}$ as f is the child of d. In the second decomposition, at iteration 2, we have $Q = \{c, b\}$ as c and b are the children of a, at iteration 3, we have $Q = \{b, d, e\}$ as d and e are the children of c, at iteration 5, we have $Q = \{e, f\}$ as f is the child of d. This results into two different BFS_{AG} codes for G, 0012300220 *for the first decomposition and* 0011300210 *for the second decomposition.*

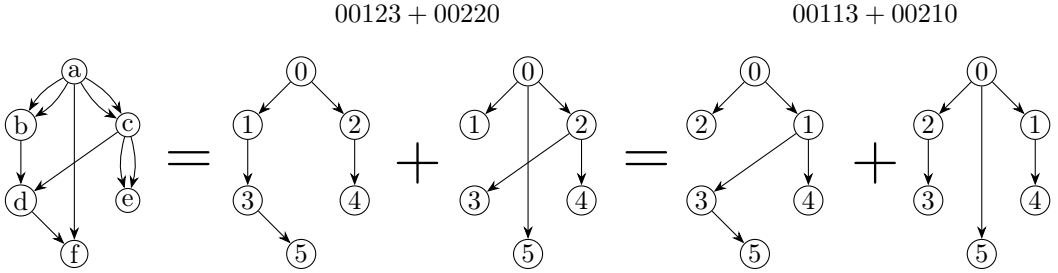


Figure 15: Two possible decompositions of an adder graph

Now, to get rid of the symmetries, we want to define a BFS_{AG} canonical code as the minimal BFS_{AG} code respecting to the lexicographic order. In Fig. 15, 0011300210 $\prec_{lex}$ 0012300220. Indeed, 0011300210 is the BFS_{AG} canonical code of G.

To explain why in the BFS_{AG} we impose to only visit vertices whose adder depth is one step bigger than the considered vertex, we need to consider this theorem:

Theorem 4.3. *In BFS_{AG} codes, $\forall\ (a, i) \in [\![1, N]\!] \times \{l, r\},\ p_i^a < a$.*

Proof. Let assume, for the sake of the argument, that there is some a such that $p_l^a \geq a$:
$\exists\ a \in [\![1, N]\!] \mid p_l^a \geq a \implies ad(a) \leq ad(p_l^a) = ad(a) - 1 \implies$ Contradiction.
On the other hand, let assume, for the sake of the argument, that there is some a such that $p_r^a \geq a$:
$\exists\ a \in [\![1, N]\!] \mid p_r^a \geq a \implies ad(a) \leq ad(p_r^a) \leq ad(p_l^a) = ad(a) - 1 \implies$ Contradiction. □

When we will construct the Constraint Program to enumerate every canonical code (see Sect. 4.2), Constraint derived from Th. 4.3 will result in a massive reduction of the search space, that we cannot ignore.

Example 4.3. *Let us consider the graph G in Fig. 16. Whereas the second decomposition corresponds to the BFS_{AG} canonical code of G as we just defined, the first decomposition corresponds to a classic BFS (without adder depth constraints). As you can see, in this decomposition, vertex c whose label is* 1 *has been visited before vertex b whose label is* 2 *even though its adder depth is greater, and the obtained code is lexicographically smaller than the BFS_{AG} canonical code (*001201 $\prec_{lex}$ 012002*). But it results that in the p_r part of the code, $p_r^1 = 2 > 1$ so $p_r^{l[c]} > l[c]$.*

That is why we impose to visit b before c.

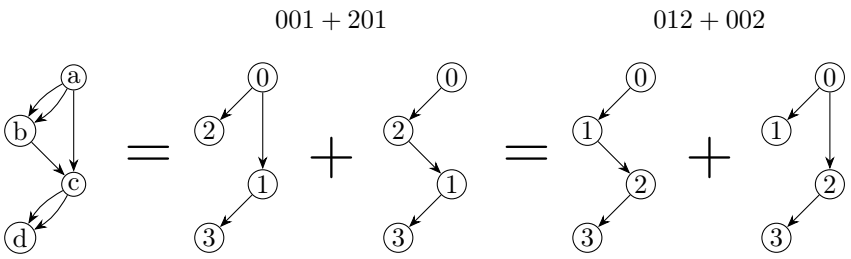


Figure 16: Comparison between two approaches of BFS

In the enumeration of adder graphs, we will need to consider prefixes of BFS_{AG} codes and a proof of canonicity of the prefixes will be essential for our canonicity check algorithm (see Sect. 4.3). A problem with this approach is that these prefixes do not describe adder graphs. Instead of that, they describe what we call *partial* adder graphs:

Definition 4.2. *A partial adder graph is a Directed Acyclic Graph $\tilde{G} = (V, \tilde{A})$ such that:*

(i) $\exists\ !\ v_0 \in V \mid d^-(v_0) = 0$

(ii) $\forall\ v \in V,\ d^-(v) \leq 2$

(iii) $\forall\ u, v \in V \setminus \{v_0\},\ ad(u) < ad(v) \implies d^-(u) \geq d^-(v)$

(iv) $\forall\ u, v \in V \setminus \{v_0\},\ |d^-(u) - d^-(v)| \leq 1$

Example 4.4. *In Fig. 18, G_1 and G_2 are partial adder graphs but G_3 and G_4 are not partial adder graphs because in G_3, $d^-(c) = 1 < 2 = d^-(e)$ and $ad(c) = 1 < 2 = ad(e)$ ((iii) not respected) and because in G_4, $|d(c) - d(f)| = |2 - 0| = 2 > 1$ ((iv) not respected).*

With this definition, it could happen that $\tilde{G}$ is not connected. In this case, the initial definition of adder depth as the maximal distance from the root could not stand, so we attribute an infinite adder depth if there is no path connecting the root to the vertex.

Example 4.5. *In Fig. 18, in G_4, f is disconnected from other vertices so $ad(f) = +\infty$.*

To obtain BFS_{AG} codes of partial adder graphs, we extend Alg. 2 which was initially designed for adder graphs by adding two modifications:

- In Line. 3, Q is sorted by decreasing in-degree. It has no impact on adder graphs because all of their vertices instead of v_0 have the same in-degree.
- Line. 9 and Line. 10 are encapsulated in an if condition testing if the right parent exists. It has no impact on adder graphs because the right parent always exists.

Theorem 4.4. *Prefixes of BFS_{AG} canonical codes are BFS_{AG} codes of partial adder graphs.*

Proof. A BFS_{AG} code c of an adder graph is the concatenation of $p_l = p_l^1 p_l^2 ... p_l^N$ and $p_r = p_r^1 p_r^2 ... p_r^N$. Let us consider a prefix of c called $\tilde{c}$ and $\tilde{G} = (V, \tilde{A})$ the graph described by $\tilde{c}$.

- If $\tilde{c}$ is in the form $p_l^1 ... p_l^k$, then $\forall\ i \in [\![1, k]\!],\ d^-(i) = 1$ and $\forall\ i \in [\![k+1, N]\!],\ d^-(i) = 0$ so $\forall\ v \in V,\ d^-(v) \leq 2$ ((ii) is respected) and $\forall\ u, v \in V \setminus \{v_0\},\ |d^-(u) - d^-(v)| \leq 1$ ((iv) is respected).

  $\forall\ u, v \in V \setminus \{v_0\}$, if $ad(u) < ad(v)$ and during the BFS_{AG} of $\tilde{G}$:
  - u and v have been visited: $d^-(u) = 1 \geq 1 = d^-(v)$.
  - u has been visited but not v: $d^-(u) = 1 \geq 0 = d^-(v)$.
  - u and v have not been visited: it is impossible because $ad(u) = ad(v) = +\infty$.
  - v has been visited but not u: it is impossible because $ad(u) = +\infty$ and $ad(v)$ is finite.

  (iii) is respected so $\tilde{G}$ is a partial adder graph.
- If $\tilde{c}$ is in the form $p_l^1 ... p_l^N p_r^1 ... p_r^k$, then $\forall\ i \in [\![1, k]\!],\ d^-(i) = 2$ and $\forall\ i \in [\![k+1, N]\!],\ d^-(i) = 1$ so $\forall\ v \in V,\ d^- \leq 2$ ((ii) is respected) and $\forall\ u, v \in V \setminus \{v_0\},\ |d^-(u) - d^-(v)| \leq 1$ ((iv) is respected).

  $\forall\ u, v \in V \setminus \{v_0\}$, if $ad(u) < ad(v)$, it means that during the BFS_{AG} of $\tilde{G}$, u has been visited before v so $l[u] < l[v]$ so $d^-(u) \geq d^-(v)$. (iii) is respected so $\tilde{G}$ is a partial adder graph.

Adder graphs are specific cases of partial adder graphs, when the prefix is the whole code. □

Example 4.6. *The BFS_{AG} canonical code* 00113002 *is a prefix of the BFS_{AG} canonical code* 0011300210 *and the corresponding graph G_1 in Fig. 18 is a partial adder graph.*

Finally, we absolutely need this fundamental property in the propagation of canonicity (see Sect. 4.3):

Theorem 4.5. *A prefix of a BFS_{AG} canonical code is a BFS_{AG} canonical code.*

Proof. Let us assume $\tilde{c}$ is non-canonical. Then the BFS_{AG} canonical code of $\tilde{G}$, denoted $\tilde{c}'$, is lexicographically smaller code than $\tilde{c}$. If the BFS_{AG} traversal of $\tilde{G}$ can be extended into a full traversal of G by discovering the remaining edges, it would result in a code c' (for which $\tilde{c}'$ is a prefix) smaller than c, which would violate the canonicity of c.

But could it not be possible to extend the traversal? As a matter of fact, it is for this precise moment that we imposed these two additional constraints in the definition of partial adder graph. The first "adder depth / degree" constraint ensures that the graph is visited "one adder depth at a time". The second "absolute difference degree" constraint ensures that every vertex is visited "once" in P_L before being visited in P_R. Finally, the order of Q in the algorithm of BFS_{AG} (decreasing in-degree) ensures that at a given adder depth, we first visit vertices whose in-degree is 2 before those whose in-degree is 1. The extension of the traversal is then always possible. □

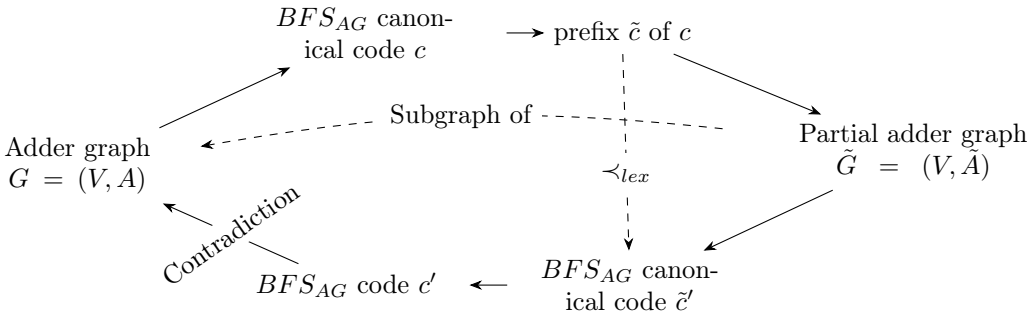


Figure 17: Illustration of the proof of Th. 4.5

Example 4.7. *In Fig. 18, G_2 is a graph that cannot be extended into the original adder graph G in Fig. 15. One could argue that its associated BFS_{AG} code* 00113001 *is not a prefix of the BFS_{AG} canonical code of G* 0011300210*, and is lexicographically smaller but in fact, this labelling of G_2 does not correspond to a BFS_{AG} traversal because the order of visit of vertices with decreasing in-degree was not respected:* 4 *was visited after* 3 *whereas $d^-(4) = 2 > 1 = d^-(3)$.*

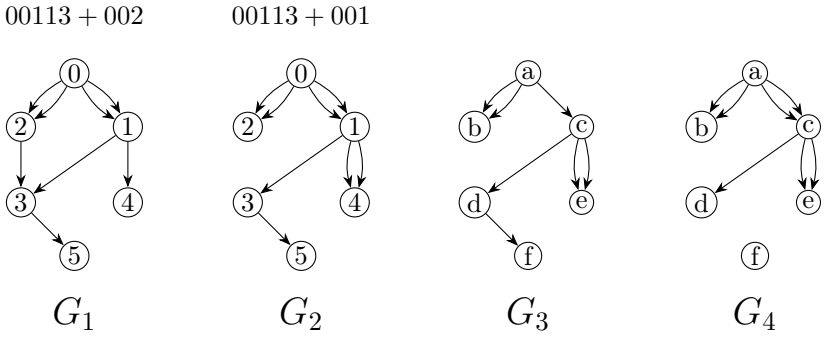


Figure 18: Good and wrong partial adder graphs

4.2 CP model

Now that we established what is a BFS_{AG} canonical code, we need to enumerate them. In order to do that, we will construct a new Constraint Program because it allows us to easily enumerate all solutions. We fix N to be the number of adders of our graph (apart from the root).

4.2.1 Variables

For each adder $a \in [\![0, N]\!]$, p_l^a and p_r^a now become variables. We know that the parents of each adder has already been visited before we visit the adder due to BFS. So the label associated to the parents are smaller than the label associated to the adder:

- $p_i^a \in [\![0, a-1]\!]$;

For each adder $a \in [\![0, N]\!]$, we introduce variables to keep track on the adder depth (necessary in order to correctly implement BFS_{AG} codes) and the out-degree in P_L of adders, which we call "left-degrees".

- $ad_a \in [\![0, a]\!]$;
- $d_l^a \in [\![0, N-a]\!]$.

4.2.2 Constraints

Constraints are listed in Fig. 4.

First, p_l^a increases with each a because after each affectation of a label to a vertex in Line. 12 of Alg. 2, i is incremented. This is expressed by Constraint C1.

C1: $\forall\ a \in [\![1, N-1]\!],\ p_l^a \leq p_l^{a+1}$
C2: $\forall\ a \in [\![1, N]\!],\ ad^a = ad^{p_l^a} + 1$
C3: $\forall\ a \in [\![1, N]\!],\ ad^{p_l^a} \geq ad^{p_r^a}$
C4: $\forall\ a \in [\![1, N-1]\!],\ ad^a \leq ad^{a+1}$
C5: $\forall\ a \in [\![1, N]\!],\ ad^{p_l^a} = ad^{p_r^a} \Leftrightarrow p_l^a \leq p_r^a$
C6: $\forall\ a \in [\![1, N-1]\!],\ d_l^a = \#\{a' \in [\![a+1, N]\!] \mid p_l^{a'} = a\}$
C7: $\forall\ a \in [\![1, N-1]\!],\ p_l^a = p_l^{a+1} \implies d_l^a \geq d_l^{a+1}$

Table 4: Constraint Programming modeling for the Enumeration of every adder graph canonical code

Second, the adder depth of adder a minus one is exactly the adder depth of its left parent, which itself is greater than or equal to the adder depth of its right parent. This is realized by Constraints C2 and C3.

Third, the redudant constraint C4 ensures that the adder depth is increasing because of BFS.

Additionally, we have this property:

Theorem 4.6. *In a BFS_{AG} canonical code: $ad^{p_l^a} = ad^{p_r^a} \iff p_l^a \leq p_r^a$*

Proof. We know that $ad^{p_l^a} \geq ad^{p_r^a}$ from Constraint C3 so $ad^{p_l^a} \neq ad^{p_r^a} \implies ad^{p_l^a} > ad^{p_r^a}$. If the adder depth of the left parent is strictly greater than the adder depth of the right parent, then $l^{-1}(p_r^a)$ has been visited before $l^{-1}(p_l^a)$ due to BFS so $p_l^a > p_r^a$.

For the other direction of the equivalence, if $ad^{p_l^a} = ad^{p_r^a}$, $p_l^a > p_r^a$ is impossible in a canonical code because otherwise, we could obtain a smaller code by popping $l^{-1}(p_r^a)$ before $l^{-1}(p_l^a)$ (they have the same depth, so it is possible during BFS), that is to say exchanging each occurrence of p_l^a and p_r^a in the canonical code, and this would contradict the canonicity of the initial canonical code. □

Example 4.8. *In Fig. 15, in the second decomposition, $ad^{p_l^3} = ad^1 = 1 = ad^2 = ad^{p_r^3}$ and $p_l^3 = 1 \leq 2 = p_r^3$.*

Constraint C5 applies Th. 4.6.

Constraint C6 ensures the consistency of d_l^a with its nature of out-degree in p_l part.

On the other hand, we have the property on left-degrees:

Theorem 4.7. *In a BFS_{AG} canonical code: $p_l^a = p_l^{a+1} \implies d_l^a \geq d_l^{a+1}$*

Proof. Similarly to Th. 4.6, if $p_l^a = p_l^{a+1}$, $d_l^a < d_l^{a+1}$ is impossible in a canonical code because otherwise, we could obtain a smaller code by visiting $l^{-1}(p_r^a)$ before $l^{-1}(p_l^a)$. Indeed, if we exchange these vertices, p_l^a and p_l^{a+1} will not change (they were equal in the first place) but there would be d_l^{a+1} occurrences of a and then d_l^a occurrences of $a+1$ which leads to a code smaller than d_l^a occurrences of a and then d_l^{a+1} occurrences of $a+1$. □

Example 4.9. *In Fig. 15, in the second decomposition, $p_l^1 = p_l^2 = 0$ and $d_l^1 = 2 \geq 0 = d_l^2$.*

Constraint C7 implements Th. 4.6.

4.3 Canonicity check

The constraints introduced above ensure that every solution satisfies the properties of BFS_{AG} codes and that no BFS_{AG} canonical code is ommited. However, constraints like C7 are not sufficient to break all symmetries, thus cannot guarantee the canonicity of each generated code. Thus, we propose a post-processing which will discard non-canonical codes. The algorithm Alg. 3 is not efficient – which is expected, since we conjecture the problem of determining whether a BFS_{AG} adder graph code is canonical not to be a polynomial-time problem (unlike tree canonicity). Otherwise, there would be no need for a CP-based formulation, as a direct polynomial-time algorithm could be used. This algorithm is instead a refined version of brute-force BFS_{AG}. To be precise, it is a Deep First Search to find every BFS_{AG} codes. While it costs a lot, we are counting on the fact that the number of BFS_{AG} codes produced by our CP model is not very big.

Alg. 3 is separated in CC, the entry point, and CC_{REC}, the auxiliary recursive function. CC_{REC} uses:

Algorithm 3: Canonicity check

Input: $\tilde{p_r}$ associated with the current BFS_{AG}, Q queue of adders waiting to be popped out and labelled, i current label, $l : V \to \mathbb{N}$ vertex labels map
Output: *true* if c is smaller or equal to all extensions of $\tilde{c} = \tilde{p_l}$, $false$ otherwise
Global Variables: Adder graph $G = (V, A)$, BFS_{AG} code $c = p_l \cdot p_r = p_l^1 ... p_l^N p_r^1 ... p_r^N$ of G

1 **function** $CC_{REC}(\tilde{p_r}, Q, i, l)$
2 **if** $Q = \emptyset$ **then**
3 **return** $(p_r \prec_{lex} \tilde{p_r})$ `// Basis case, comparison of` p_r `parts`
4 Get v_l first vertex of Q
5 $N \leftarrow \{v \in V \setminus Q \mid (v_l, v) \in A \wedge ad(v) = ad(v_l) + 1\}$ `// Set of next vertices`
6 **if** $N = \emptyset$ **then**
7 **return** $CC_{REC}(\tilde{p_r}, Q \setminus \{v_l\}, i, l)$
8 **if** $l[v_l] \neq p_l^i$ **then**
9 **return** $(l[v_l] > p_l^i)$ `// The current code is different from the reference code`
10 **forall** $v \in \arg\max_{w \in N} \#\{u \in V \mid (w, u) \in A \wedge ad(u) = ad(w) + 1\}$ **do**
11 Let v_r be the other parent of v
12 **if** $\neg(CC_{REC}(\tilde{p_r} + l[v_r], Q \cup \{v\}, i + 1, l \cup \{v \mapsto i\}))$ **then**
13 **return** *false* `// Some smaller code was found`
14 **return** *true*

Input: BFS_{AG} code $c = p_l^1 ... p_l^N p_r^1 ... p_r^N$
Output: *true* if c is a BFS_{AG} canonical code, $false$ otherwise

15 **function** $CC(c)$
16 $G = (V = (v_i)_{i \in [\![1,N]\!]}, A = \{(v_{p_l^i}, v_i), (v_{p_r^i}, v_i), i \in [\![1, N]\!]\})$ `// Initialization`
17 **return** $CC_{REC}(\text{""}, \{v_0\}, \{\}, 1, \{v_0 \mapsto 0\})$

- as global variables; elements linked to the code whose canonicity is checked: G, c
- as arguments; elements linked to current state of the BFS_{AG}: $\tilde{p_r}$, Q, i, l

Some parts of Alg. 3 are pretty much identical to Alg. 4. Indeed, this recursive function should be seen as the BFS_{AG} algorithm where we replace the while loop by a recursive call. We colored in red every difference between the two algorithms.

In addition to these familiar features, we added specific return calls. In Line. 8, we compare the label of the next vertex to add to the code we are constructing with the corresponding label in the reference code c. We use the canonicity of the prefix (cf. Th. 4.5) to make a conclusion:

- If the former is smaller than the latter, it means that c is not a canonical code.
- If the former is greater than the latter, it means that our current BFS_{AG} cannot contradict the canonicity of c.

In Line. 12, we choose vertex v as the next vertex in our current BFS_{AG} and try in the recursive call to extend it in order to find a code contradicting the canonicity of c. If the attempt was successful, it means c was not canonical.

Finally, in Line. 2, we deal with the base case, when the queue is empty. This means that our BFS_{AG} is finished. Although no canonicity contradiction was raised throughout the execution, we still need to check whether the p_r part of our constructed code is greater than the p_r part of the reference code or not.

Example 4.10. *In Fig. 19 are two BFS_{AG} codes of the same adder graph with same part p_l but different p_r.*

The entry function simply consists in a call of the recursive function with initial arguments (the empty $\tilde{p_r}$, a queue only containing v_0, the label map initialized with 0 value for v_0, the current index 1). Of course, adder depths are actually computed in this entry function because we do not want to recompute adder depths during each recursive call.

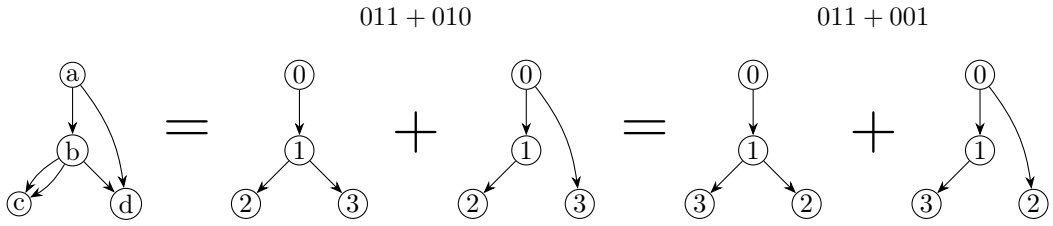


Figure 19: Two possible decompositions of an adder graph with same part p_l

We still need to explain the order in Line. 10. One important consideration in canonicity check is that whereas, with a canonical code as c, we need to explore the whole "recursive tree" of CC_{REC} calls, having a non-canonical code allows us to stop as soon as we encounter a canonicity contradiction. So we want to find the canonical code as soon as possible. With that in mind, we can understand that the order of the for loop is an implication of Th. 4.7 (for 2 vertices u and v having the same parent, it is better to explore u before v if the out-degree of u is greater than the out-degree of v).

4.4 Instance compatibility check

We are going to consider additional constraints depending on the instance we are working on thanks to theorems from Sect. 2.1. These constraints will not be handled within the Constraint Programming model, as they would induce an undesirable overhead. Instead, they will be enforced by simple post-processing polynomial algorithms, which discard canonical codes that are not valid for the given target constant set.

First, let us add additional binary variables $free_a$ evaluating if adder a is a free adder.

$$\forall\ a \in [\![1, N]\!],\ free_a \leftarrow \begin{cases} 1 & \text{if } d^+(a) = \#\{(a', i) \in [\![1, N]\!] \times \{l, r\} \mid ind_{a',i} = a\} = 0 \\ 0 & \text{otherwise} \end{cases}$$

Theorem 4.8. $\#\{a \in [\![1, N]\!] \mid free_a = 1\} \leq |T|$

Proof. From Th. 2.4, we know that in optimal adder graphs, all free adders produce target constants, so there cannot be more free adders than target constants. □

Second, if we denote $\mathcal{A} = \{D_{LB}(T_j) \mid j \in [\![1, |T|]\!]\}$, we have the following:

Theorem 4.9. $\forall\ k \in \mathcal{A},\ \#\{j \in [\![1, |T|]\!] \mid D_{LB}(T_j) \geq k\} \leq \#\{a \in [\![1, N]\!] \mid ad(a) \geq k\}$

Proof. From Th. 2.5, we know that if an adder produces target T_j, then its adder depth is at least $D_{LB}(T_j)$.

Target constants are different and adders produce only one value at once, so for every possible adder depth k, there must be at least as many adders whose adder depth is greater than k as there are target constants whose adder depth lower bound is greater than k. This is exactly what describes the inequality of Th. 4.9. □

Example 4.11. *In Fig. 20, if we consider $T = \{7, 19, 31\}$ ($D_{LB}(T) = \{1, 2, 1\}$), G_2 and G_3 are compatible with Th. 4.9 but G_1 is not because $ad(b) = ad(c) = ad(d) = 1 < 2$ so $\#\{j \in [\![1, |T|]\!], \mid D_{LB}(T_j) \geq 2\} = \#\{19\} = 1 > 0 = \#\{a \in [\![1, N]\!] \mid ad(a) \geq 2\}$. Simply said, we are missing one adder at the correct depth (at least 2) to produce 19. If we consider $T = \{3787, 9943, 15567\}$ ($D_{LB}(T) = \{3, 3, 3\}$), G_2 and G_3 are compatible with Th. 4.9.*

Third, if we denote $\mathcal{D} = \{S_{LB}(T_j) \mid j \in [\![1, |T|]\!]\}$ then we have the following:

Theorem 4.10. $\forall\ k \in \mathcal{D},\ \#\{j \in [\![1, |T|]\!] \mid S_{LB}(T_j) \geq k\} \leq \#\{a \in [\![1, N]\!] \mid n_{dep}(a) \geq k\}$

Proof. From Th. 2.6, we know that if an adder produces target T_j, then its dependance number is at least $S_{LB}(T_j)$.

Target constants are different and adders produce only one value at once, so for every possible dependence number k, there must be at least as many adders whose dependence number is greater than k as there are target constants whose dependence lower bound is greater than k. This is exactly what describes the inequality of Th. 4.10. □

Example 4.12. *In Fig. 20, if we consider* $T = \{3787, 9943, 15567\}$ *(*$S_{LB}(T) = \{4, 4, 4\}$*), G_3 is compatible with Th. 4.10 but G_2 is not because* $n_{dep}(h) = n_{dep}(i) = 4 \neq 3 = n_{dep}(j)$ *so* $\#\{j \in [\![1, |T|]\!], \mid S_{LB}(T_j) \geq 4\} = \#\{3787, 9943, 15567\} = 3 > 2 = \#\{h, i\} = \#\{a \in [\![1, N]\!] \mid n_{dep}(a) \geq 4\}$*. Simply said, we are missing one adder with the correct dependence number (at least* 4*) to produce* 3787*,* 9943 *or* 15567*.*

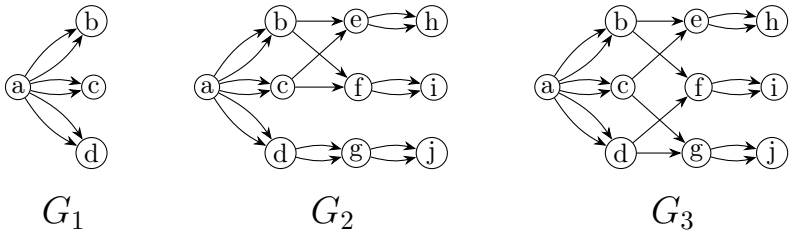


Figure 20: Compatibility with adder graphs

In the reality of our process, we enumerate every canonical code, store them once for all in preprocessed data files and when we consider a specific instance, get rid of any canonical code non-proper to this specific instance by checking if every instance-specific constraints listed above are respected.

4.5 Performance results

In Fig. 21, we displayed our results on adder graph generation. In Fig. 21a are displayed our different codes quantities:

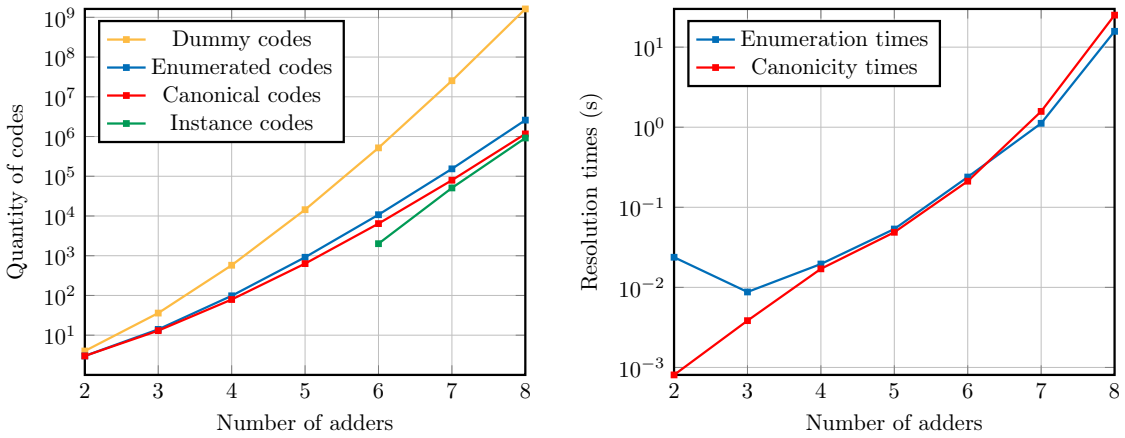
1 The "dummy codes" in yellow are the codes we would have obtained without thinking the process through (they contain every redundant adder graph).

2 The "enumerated codes" are the codes obtained after the CP model in Sect. 4.2. They still contain redundant graphs but most of them were removed.

3 The "canonical codes" are the codes obtained after canonicity checks Sect. 4.3. At this point, we are sure that they do not contain any redundancy.

4 The "instance codes" are the codes obtained after instance check of Sect. 4.4 by removing every canonical code incompatible with the specific target set instance $T = \{3787, 9943, 15567\}$.

In Fig. 21b are displayed the time resolutions for the CP enumeration and Canonicity check executions. These execution times are reasonable when the number of adders is small enough. However, as the number of adder graphs grows exponentially with respect to the number of adders, this approach cannot be used when the number of adders is too large.



(a) Comparison between number of codes

(b) Comparison of resolution times between first enumeration and canonicity check

Figure 21: Analysis of enumeration and canonicity check results.

First and foremost, you can notice that the number of enumerated codes and the number of canonical codes seem to have the same exponential rate (the quotient of their logarithm is close to 1). This can be reassuring. Indeed, the number of adder graphs exponentially increases with the number of adders, as for the number of codes corresponding to the same adder graph. But this double exponentiation is only slightly reflected in the results, so we do not produce "too much" non-canonical codes. On another hand, even with all our refinements, the number of codes to test stays very high (over a million for 8 adders) and probably unsuitable for use.

5 Solving MCM with canonical codes

So far, we established Constraint Programming models for the MCM problem, one Decision problem with fixed number of adders and one Minimization of the number of adders. We also found an effective way to enumerate every possible adder graph thanks to canonical codes. We finally can realize the final step of this thesis: linking these models together to solve the MCM problem.

5.1 Fixed Topology Decision Problem

By the "topology" of an adder graph, we mean the edges of an adder graph without any associated fundamental. The Fixed Topology Decision Problem consists in finding a proper fundamental attribution of such a graph in order to obtain a realizable solution, that is to say, finding proper values of shifts, signs and fundamentals, knowing for each adder who are their parents. We thought our Satisfiability model such that it can be "grafted" onto the constraints of Fig. 5.

5.1.1 Constraints

C1-C11: Constraints from CP Model with fixed number of adders +
C12: $\forall\ a \in [\![1, N]\!],\ \{ind_{a,l}, ind_{a,r}\} = \{p_{a,l}, p_{a,r}\}$
C13: $\forall\ j \in [\![1, |T|]\!],\ T_j \in \{c_a \mid a \in [\![1, N]\!] \wedge ad(a) \geq D_{LB}(T_j) \wedge n_{dep}(a) \geq S_{LB}(T_j)\}$
C14: $\{c_a \mid a \in [\![1, N]\!] \wedge \text{adder } a \text{ is a free adder}\} \subset T$
C15: $\forall\ O \in \mathcal{O}(G) \mid |O| > 1,\ \forall\ m \in [\![1, |O| - 1]\!],\ c_{O_m} < c_{O_{m+1}}$

Figure 22: Constraint Programming modeling for Fixed Topology Decision Problem

Constraints are listed in Fig. 22.

Let us go into further details about the additional constraints of this new model. Fixing the topology does not mean that we are going to fix every $ind_{i,a}$ values because for each fundamental value, we do not know who is the left input and who is the right input between the two parents. This distinction is very important since the way we handle input is asymmetrical (we can only left-shift the left input, cf. Th. 2.1). Therefore, Constraint C12 ensures that either $ind_{a,l} = p_{a,l}$ and $ind_{a,r} = p_{a,r}$, either $ind_{a,l} = p_{a,r}$ and $ind_{a,r} = p_{a,l}$ (these cases are identical if $p_{a,l} = p_{a,r}$).

We can also easily precompute the adder depth and the dependence of a given adder in polynomial time. This allows us to model Th. 2.5 and Th. 2.6 in Constraint C13.

On the other hand, we can know for free if a given adder is a free adder. This allows us to model Th. 2.4 in Constraint C14.

Constraint C15 will be covered in Sect. 5.3.

5.2 Adder propagator

When the topology is fixed, to speed up the computations, we created our own propagator. A propagator in a CP solver is an algorithm attached to a constraint that prunes the variables' domains without removing any valid solutions. It uses the constraint to eliminate impossible values, thereby speeding up the search by reducing the solution space.

We introduce a new type of adder graphs:

Definition 5.1. *A "flattened" adder graph is an adder graph such that:*

$$\forall\ a \in [\![1, N]\!],\ ad(a) = \min\{\max(ad(u), ad(v)) + 1 \mid (u, v) \in [\![1, a-1]\!]^2 \text{ and } \mathsf{can}(c_u, c_v, c_a)\}$$

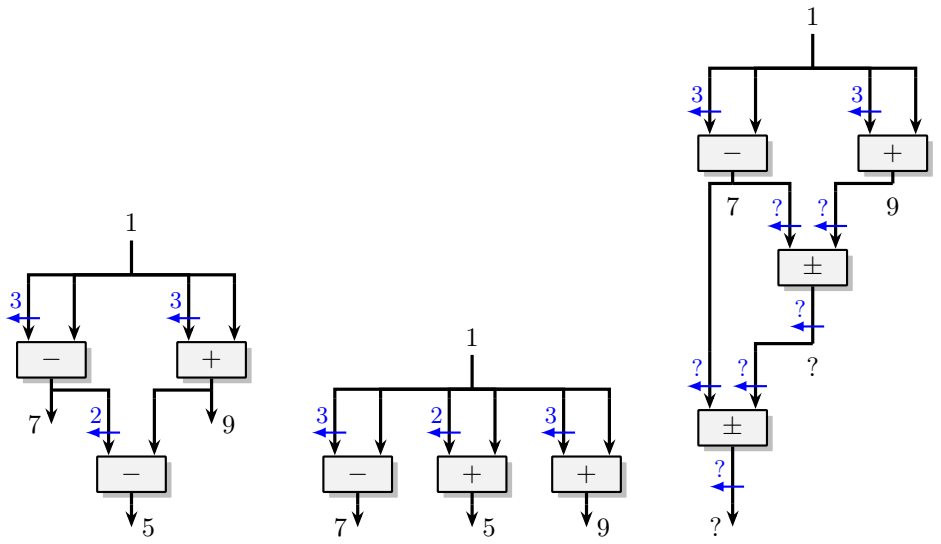
The principle of the propagator is based on the following theorem:

Theorem 5.1. *For every target constant set, there always exists an optimal flattened adder graph.*

Proof. Let us consider an optimal adder graph. If $\exists\ a \in [\![1, N]\!]$ such that $\exists\ (u, v) \in [\![1, a-1]\!]^2$ | $\mathsf{can}(c_u, c_v, c_a)$ and $ad(a) > \max(ad(u), ad(v))+1$, then we just have to change the parents of a: $ind_{a,l} \leftarrow u$ and $ind_{a,r} \leftarrow v$. We can repeat the process until we reach the adder depth bound for every fundamental. □

This theorem tells us we can "flatten" adder graphs as much as possible.

Example 5.1. *In Fig. 23a, we can notice that* $c_3 = 5 = 2^2 + 1 = 2^2 c_0 + c_0$ *so* $\mathsf{can}(c_0, c_0, c_3)$ *but* $ad(3) = 2 > 1 = \max(ad(0), ad(0)) + 1$. *This means that the adder graph is not flattened. By branching* $p_l(3)$ *and* $p_r(3)$ *on* 0 *and* 0, *we obtain a flattened adder graph displayed in Fig. 23b.*



(a) Optimal adder graph for $\{5, 7, 9\}$ (b) Flattened optimal adder graph for $\{5, 7, 9\}$ (c) Partially instantiated adder graph

Figure 23: Flattening of adder graph used in the propagation

In the context of our propagation, we typically encounter partially instantiated adder graphs such as the one presented in Fig. 23c. As long as we ensure that fundamentals are instantiated by increasing order, we can impose the following propagation rule; $\forall\ a \in [\![1, N]\!]$, if c_b is instantiated $\forall\ b \in [\![1, a]\!]$:

- $\forall\ a' \in [\![a+1, N]\!],\ ad(a') > ad(a) + 1 \implies c_{a'} \notin \bigcup_{k \in [\![0,a]\!]} \mathcal{A}_w(c_a, c_k)$
- $\forall\ a' \in [\![a+1, N]\!],\ \exists\ k \in [\![0, a]\!] \mid \{p_{a',l}, p_{a',r}\} = \{a, k\} \implies c_{a'} \in \mathcal{A}_w(c_a, c_k)$

Example 5.2. *At this given step in Fig. 23c, if* $c_2 = 9$ *was the last fundamental to be instantiated, then we will first make sure that* $c_4 \notin \mathcal{A}_w(1, 9) \cup \mathcal{A}_w(7, 9) \cup \mathcal{A}_w(9, 9)$ *and then that* $c_3 \in \mathcal{A}_w(7, 9)$.

It is important to understand that the propagation will result with a fail with some optimal adder graphs but never with the optimal flattened graph.

5.3 Adder graph orbits

A graph automorphism is a symmetry function mapping the graph onto itself while preserving the edge-connectivity. Mathematically, an automorphism of $G = (V, E)$ is a permutation $\pi : V \mapsto V$ such that $(u, v) \in E \Leftrightarrow (\pi(u), \pi(v)) \in E$, that is a graph isomorphism from G to itself. The set of automorphism of G is denoted $Aut(G)$. Adder graph automorphisms exist: by definition, the identity permutation is an automorphism for every graph. But adder graphs can have non-identity automorphisms.

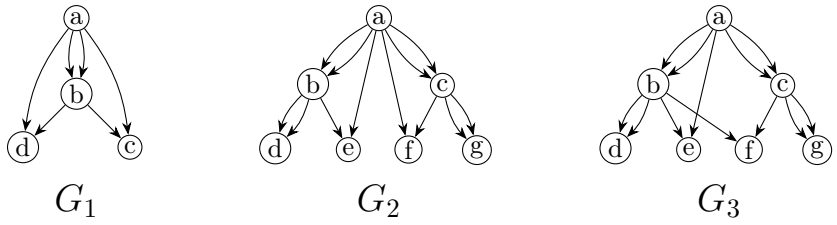


Figure 24: Adder graphs with and without automorphisms

Example 5.3. *In Fig. 24, in* G_1, $\pi(a,b,c,d) = (a,b,d,c)$ *and in* G_2, $\pi(a,b,c,d,e,f,g) = (a,c,b,g,f,e,d)$ *are non-identity adder graph automorphisms.* G_3 *is an adder graph which admits no other automorphism than the identity permutation.*

It is reflected in the search for the canonical code. While the canonical code of an adder graph is the minimum code (respecting to the lexicographic order) among all BFS_{AG} traversals, it is generally not produced by a unique BFS_{AG} traversal. Indeed, BFS_{AG} traversals resulting in the same code correspond to adder graph automorphisms.

The orbit set of a graph $G = (V, A)$ is denoted $\mathcal{O}(G)$ and is formally defined using, $\forall\ (u, v) \in V^2$:

$$u \text{ and } v \text{ are in the same orbit} \Leftrightarrow \exists\ \pi \in Auto(G)\ \mid\ \pi(u) = v,\ \pi(v) = u \text{ and } Pred(u) = Pred(v)$$

Where $Pred(v)$ is the set of precedessors of v, that is to say, in adder graph, its two parents:

$$Pred(v) = \{u \in V \mid (u, v) \in A\}$$

Being in the same orbit defines an equivalence relation on V. Orbits are defined as equivalence classes induced by this relation. If a graph only admits the identity permutation as an automorphism, then its orbit set only contains one-vertex orbits.

Example 5.4. *In Fig. 24, in* G_1, $\{c, d\}$ *and in* G_2, $\{b, c\}$ *are non one-vertex adder graph orbits.* G_3 *is an adder graph which admits no other orbit than one-vertex orbits.*

From the Canonicity Check Algorithm Alg. 3, it is very simple to construct an algorithm generating all orbits of an adder graph. We have described such an algorithm in Alg. 4. Labels have the role of the permutations. The principle is simple: when chosing the next vertex to consider in the For loop of Line. 11, if two different next vertices lead to the reference canonical code, then they belong to the same orbit (they are "exchangeable"). The rest of Alg. 4 is pretty much the same as Alg. 3, we colored in red every difference between the two algorithms.

Now that we have at our disposal every orbit, we can impose an order relationship constraint on the values of orbits. The idea is that if a solution is valid on a graph, we do not need to visit symmetric solutions with permutations of values into orbits. The corresponding constraint is:

$$\forall\ O \in \mathcal{O}(G) \mid |O| > 1,\ \forall\ m \in [\![1, |O| - 1]\!],\ c_{O_m} < c_{O_{m+1}}$$

The strict order is implied by the constraint "allDifferent" over the fundamentals. We call each of these constraints an "orbit ordering constraint". One way to measure the efficiency of our orbit approach is to count the number of orbit ordering constraints. It is equal to:

$$\sum_{O \in \mathcal{O}(G)} (|O| - 1) = \sum_{O \in \mathcal{O}(G)} |O| - |\mathcal{O}(G)|$$

In order to determine whether this orbit approach is really effective, we analyzed in Fig. 25 the distribution of orbits. In Fig. 25a, we displayed the proportion of adder graphs which have orbits with more than one element, that is to say for which it admits at least an orbit ordering constraint. It is interesting to notice that bigger adder graph lead to fewer orbit ordering constraints. In Fig. 25b, we displayed a more detailed graphic showing the distribution of the number of adder graphs corresponding to different numbers of orbit ordering constraint. It is unsurprisingly decreasing: adder graphs with lot of symmetries are very constrained so there are not many of them.

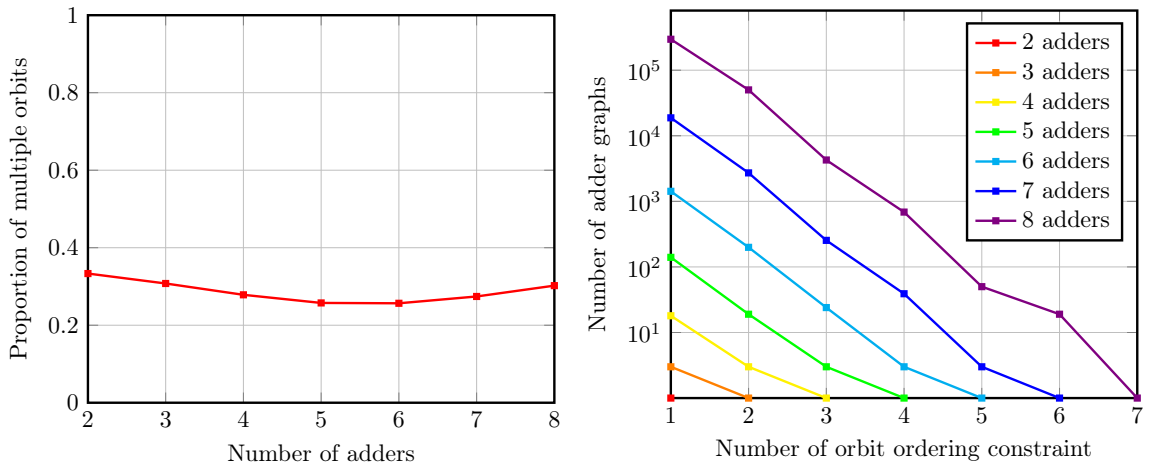
Algorithm 4: Adder graph orbits

Input: $\tilde{p_r}$ associated with the current BFS_{AG}, Q queue of adders waiting to be popped out and labelled, i current label, $l : V \to \mathbb{N}$ vertex labels map
Output: True iff the BFS_{AG} can lead to the canonical code
Global Variables: Adder graph $G = (V, A)$, BFS_{AG} code $c = p_l \cdot p_r = p_l^1 ... p_l^N p_r^1 ... p_r^N$ of G, Orbit set $\mathcal{O}(G)$

function $Orbit_{REC}(\tilde{p_r}, Q, i, l)$
  if $Q = \emptyset$ **then**
    return $(p_r = \tilde{p_r})$
  Get v_l first vertex of Q
  $N \leftarrow \{v \in V \setminus Q \mid (v_l, v) \in A \wedge ad(v) = ad(v_l) + 1\}$
  if $N = \emptyset$ **then**
    return $Orbit_{REC}(\tilde{p_r}, Q \setminus \{v_l\}, i, l)$
  if $l[v_l] > p_l^i$ **then**
    return *False*
  Let O be an empty set of vertices
  forall $v \in \arg\max_{w \in N} \#\{u \in V \mid (w, u) \in A \wedge ad(u) = ad(w) + 1\}$ **do**
    Let v_r be the other parent of v
    if $Orbit_{REC}(\tilde{p_r} + l[v_r], Q \cup \{v\}, i + 1, l \cup \{v \mapsto i\})$ **then**
      $O \leftarrow O \cup \{v\}$
  $\mathcal{O}(G) \leftarrow \mathcal{O}(G) \cup \{O\}$
  return $O \neq \emptyset$

Input: BFS_{AG} canonical code $c = p_l^1 ... p_l^N p_r^1 ... p_r^N$
Output: Set of all orbits of G

function $Orbit(c)$
  $G = (V = (v_i)_{i \in [\![1,N]\!]}, A = \{(v_{p_l^i}, v_i), (v_{p_r^i}, v_i), i \in [\![1, N]\!]\})$
  Let $\mathcal{O}(G)$ be an empty set of subsets of V
  $Orbit_{REC}(\text{""}, \{v_0\}, \{\}, 1, \{v_0 \mapsto 0\})$
  return $\mathcal{O}(G)$



(a) Proportions of adder graphs with effective orbit (orbit with more than 1 vertex)

(b) Distribution of number of adder graphs respecting to the number of orbit ordering constraints

Figure 25: orbit ordering constraint distribution analysis

5.4 Implementation

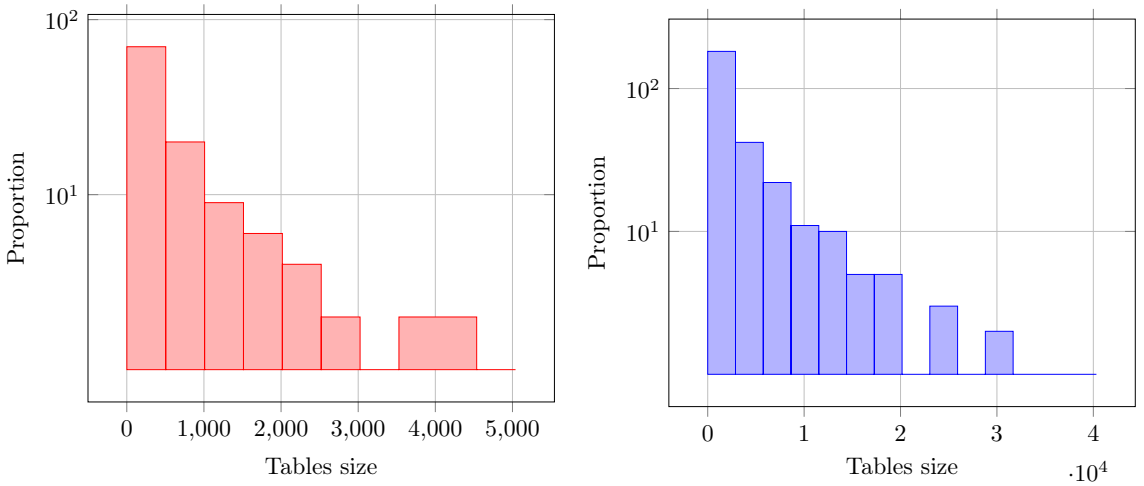
The linking between our models works with an outer loop on the number of adders (we start with the number of target constants, and we increment it while no topology with the given amount of adders is

satisfiable). We have few options in order to implement the content of this outer loop, see Tab. 5:

	Option 1	**Option 2**	**Option 3**	**Option 4**
Outer loop	Launching the model on every topology	Creating a Table constraint linking $ind_{a,i}$ variables and $p_{a,i}$ values of canonical codes + Launching the model with this constraint	Fixing the p_l part of some canonical code + Creating a Table constraint linking $ind_{a,r}$ variables and $p_{a,r}$ values with fixed p_l part + Launching the model with this constraint	Combining fixed topology model with $p_{a,i}$ subject to canonical code enumeration constraints
Type of model	Fixed Topology Decision Problem	Fixed Number of adders Decision Problem	Fixed Number of adders Decision Problem	Fixed Topology Decision Problem + Canonical Code Enumeration
Number of models	Number of canonical codes (parallelizable)	One	Number of canonical codes p_l parts (parallelizable)	One

Table 5: Characteristics of options

The main default of **Option 1** lies in the number of CP models to launch (equal to the number of canonical codes). As a reminder, with 8 adders, the number of adder graphs exceeds 10^6. Similarly, we expect **Option 2** to be intractable because of the size of the table of canonical codes. However, **Option 3** could provide an in-between: there are not too many models to launch (for example, with 8 adders, there are 286 different p-parts) and not too large tables (for example, with 8 adders, there is minimum 1 and maximum 40320 topologies corresponding to the same p-part). In Fig. 26 are displayed the histograms of the sizes of p-part tables corresponding to 7 and 8 adders. We can observe that most of p-part tables are small. Indeed, fixing the p-part is quite constraining so there is generally not a lot of adder graphs with the same p-part.

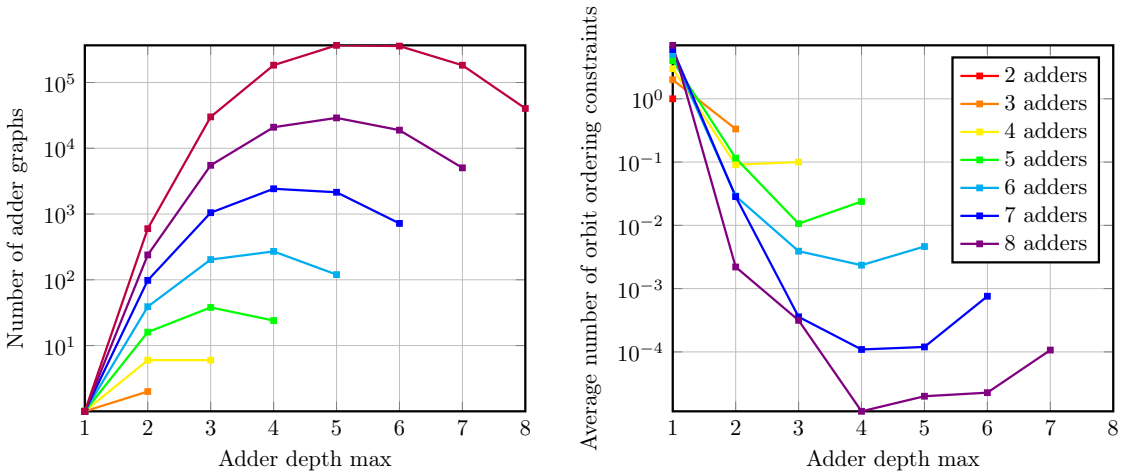


(a) Histogram of fixed p part table sizes with 7 adders (b) Histogram of fixed p part table sizes with 8 adders

Figure 26: Fixed p part analysis

We impose in **Option 1** that topologies are visited by increasing maximal adder depth. There are two reasons for such an order. First, since optimal flattened adder graphs minimize the maximal adder depth, by starting with small adder depth, we have more chances to find it faster. In Fig. 27a, we displayed the distribution of number of adder graphs with given maximal adder depth. The number of adder graphs grows very fast so without imposing this order, we could waste a lot of time exploring unflattened topologies. Second, as confirmed by Fig. 27b, it is intuitive that a small maximal adder depth leads to a large number of possible orbit order constraints. Indeed, small maximal adder depths lead to a lot of

free adders so a lot of potential automorphisms. If the number of orbit ordering constraints is very large, then our previous orbit approach is very constraining, so MCM Decision Problem resolution time are smaller. Be aware that even though the average number of orbit ordering constraints is very large with small maximal adder depth, it is compensated by the small corresponding amount of adder graphs.



(a) Distribution of number of adder graphs respecting to adder depth max

(b) Distribution of the number of orbit ordering constraints respecting to adder depth max

Figure 27: Adder depth and orbits analysis

To give more explanations, there is only one adder graph with maximal adder depth equal to one for every number of adders because it corresponds to the adder graph where both inputs of every adder are the origin.

Finally, **Option 4** is a completely different model trying an hybrid approach, consisting in keeping the MCM model from Sect. 3.1 but adding fixed topology constraints from Sect. 5.1 with variables $p_{a,i}$ subject to constraints from Sect. 4.2. We will not directly handle canonical codes but rather the codes subset generated by the Constraint Program for adder graph enumeration. As we instantiate fundamentals during the propagation, CP will discard branching instanciations contradicting the canonicity of the code. During the outer loop process, it will not impact satisfiability proofs (statistically, the number of cut unsatifiable solutions is compensated by the number of cut satisfiable solutions) but it will greatly impact unsatifiability proofs (the more solutions we cut, the faster we explore all the search space).

5.5 Performance results

These models have been implemented in Java with Choco-solver [43]. As explained in options **1** and **3**, we took advantage of parallelism by using the Java concurrency library (java.util.concurrent) to distribute independent tasks across multiple threads. Unfortunately, neither of these 3 first approaches was viable. This was mainly due to the bad performances of the CP models from Sect. 5.1 and Sect. 3.1 with table constraints.

Option 4 still remains to be properly tested and benchmarked.

6 Conclusion

In this Master thesis, we studied optimization problems related to an arithmetic operator called Multiple Constant Multiplication. Engineers in charge of making hardware implementation decisions generally do not have the necessary background to ensure that their choices are the best ones. Therefore, we created a black box framework using Constraint Programming adapted to their needs.

We demonstrated how strong Constraint Programming was to model some combinatorial problems which are difficultly expressible using Integer Linear Programming. This superiority is reflected both in the ease of modeling (cf Sect. 3.1 and Sect. 3.2) and the performances of our models, which outerformed ILP and was very competitive with SAT (cf Sect. 3.4).

It is also very interesting to notice that fixing the topology of the MCM implementation does not make the resolution of the problem free, far from it on the contrary (cf Sect. 5.5). We initally expected

the complexity of the problem to be lying in the decision of how to allocate adders in relation to each other, but in reality, a great part of complexity comes from the decision of the fundamentals.

However, we can extract from fixed topology a surprisingly large number of knowledge by benefiting from graph theory properties (cf Sect. 4.4).

6.1 Further work and perspectives

The paper presented in Sect. D of the appendix, co-written by Cantaloube Théo, Peng Xiao, Solnon Christine and Volkova Anastasia, containing the model from Sect. 3.1 was submitted, accepted and presented on the 11^{th} of August during the ModRef workshop of The 31^{st} International Conference on Principles and Practice of Constraint Programming.

I will be starting a PhD on the 1^{st} of October with Solnon Christine and Volkova Anastasia, funded by a grant from the InfoMathsdoctoral school. Studies will still focus on MCM on different aspects:

- Can we leverage fix topologies by implementing adder graph generation and solving of MCM with canonical codes using SAT models? Re-writing models in C++ instead of Java also would speed up computations. Another option could be to try an "hybrid" approach consisting in fixing the topology and choosing fundamentals "as we go along".
- Is MCM really a $\mathcal{NP}$-hard problem? It seems that the demonstration of $\mathcal{NP}$-hardness of the MCM problem from [9] is not correct because it assumes that the MCM problem is the same as the Ensemble Computation problem proven $\mathcal{NP}$-hard in [20], which is false. Thus, a proper demonstration of the $\mathcal{NP}$-hardness of the MCM problem would be useful. In fact, even MCM with fixed topology is an $\mathcal{NP}$-hard problem. A preliminary proof has been developed, but it still requires formalization.

On another hand, we will consider optimization at a higher level of abstraction, working on FloPoCo [13] and MLIR [35], other tools used or maintened by the Emeraude team. This would also involve finding subgraphs in graphs, a problem where graph symmetries play a key role.

A Benchmark description

Here is the totality of the 83 instances of the benchmark we have worked on [40].

Name	w	$\|T\|$	T
Bull91 32	7	10	3 5 7 9 15 35 45 49 107 127
Chang06 3	10	3	139 283 815
Chen99 15	11	4	13 223 897 2017
Chen99 28a	9	9	3 7 9 21 27 43 69 97 261
Chen99 28b	11	9	5 9 13 19 23 27 97 119 1151
Chenyao01 28a	8	8	3 5 11 15 25 33 121 177
Chenyao01 28b	10	9	3 5 11 15 25 39 121 133 1023
Dempster02 25	12	13	35 133 199 291 327 331 355 499 505 699 1943 2987 3395
Dempster04 a3	8	3	5 117 195
Dempster04 b3	8	3	13 43 215
Dempster04 c3	11	3	947 973 1243
Dempster04 d3	10	3	105 479 939
Dempster04 e3	14	3	3787 9943 15567
Dempster04 f3	14	3	3981 6243 14603
Dempster04 g4	13	4	303 3643 6729 7479
Dempster04 h4	14	4	10083 12095 12975 15103
Dempster04 i5	12	5	63 407 1823 3059 3817
Dempster04 j5	11	5	19 841 911 935 1561
Dempster04 k5	12	5	407 569 831 1115 2473
Dempster04 l5	12	5	947 973 1243 1991 3651
Gaussian 3	8	3	3 21 159
Gaussian 5	11	3	23 343 1267
Goodman77 c7	4	1	9
Goodman77 d7	5	2	3 19
Goodman77 e11	7	3	3 25 75
Goodman77 f11	8	4	9 11 13 173
Goodman77 g11	8	3	7 53 151
Goodman77 h15	9	5	3 29 33 245 401
Goodman77 i19	12	5	9 29 429 1277 2521
Highpass 15	9	12	3 5 7 9 11 13 15 17 19 21 23 507
Highpass 5	7	4	3 5 7 121
Highpass 9	7	5	3 5 7 11 125
Jain91 11	5	3	3 5 17
Jang02 11	3	3	3 5 7
Jheng04 29a	9	9	5 7 11 17 25 31 71 73 389
Jheng04 29b	6	9	5 9 13 15 17 19 33 39 47
Jheng04 30	7	8	3 5 7 11 29 39 43 123
Johansson08 28	10	12	3 9 33 45 57 73 139 143 167 273 363 571
Johansson08 30	10	28	37 47 53 57 67 81 93 133 165 179 185 211 223 253 261 283 367 375 409 441 447 495 497 647 915 961 963 975
Johansson08 a5	10	4	9 387 745 805
Johansson08 b5	10	5	57 155 189 431 611
Kwentus97 11	7	3	3 25 75
Kwentus97 15	10	3	5 277 621
Kwentus97 47	14	13	21 27 29 55 187 207 381 623 913 977 1147 1295 16359
Laplacian 3	7	3	5 21 107
Lim83 121	14	51	3 7 9 17 19 21 23 35 37 39 41 45 51 55 59 61 65 69 71 75 77 89 109 125 131 141 143 151 165 167 193 195 267 273 277 323 375 399 541 545 585 761 1071 1515 1653 1739 2823 5927 6747 7343 10267
Lim83 36	4	5	3 5 7 9 15
Lim83 37	7	5	3 5 7 9 127
Lim83 63	9	22	3 5 7 9 15 19 21 23 29 31 33 37 41 59 61 71 99 119 195 321 351 431

Name	w	$\|T\|$	T
Lim83A 36	5	5	3 5 7 9 31
Limakt08 121	14	42	3 5 7 9 11 17 19 21 23 25 31 33 35 37 41 49 61 65 69 73 79 89 119 131 151 161 167 189 197 271 281 379 413 533 537 541 641 869 1685 2993 5921 14671
LimpaskO99 121	14	51	3 7 9 17 19 21 23 35 37 39 41 45 51 55 59 61 65 69 71 75 77 89 109 125 131 141 143 151 165 167 193 195 267 273 277 323 375 399 435 541 545 585 741 761 827 1071 1515 1687 2567 2823 14687
Limyu07 121	14	43	3 5 7 9 11 13 15 19 29 31 53 57 59 89 113 125 129 131 133 139 145 183 241 263 271 339 355 375 475 491 493 553 595 713 789 1215 1275 1343 1369 1371 6115 6655 9305
Lowpass 15	11	25	5 7 13 17 19 21 27 41 43 45 53 61 79 93 101 103 113 133 137 199 331 333 613 1097 1197
Lowpass 5	7	5	11 33 35 53 103
Lowpass 9	9	12	5 7 25 31 63 65 67 73 97 117 165 303
Martinez02 4	10	4	105 621 815 831
Nielsen89 67a	15	26	3 9 11 13 27 39 45 51 95 113 153 217 263 323 379 411 431 471 987 993 1523 2347 3413 3753 10693 32765
Nielsen89 67b	15	28	3 5 11 13 19 25 27 39 51 95 113 145 161 215 247 307 379 411 431 469 527 761 943 993 1173 6827 16383 21387
Rosa04 49	9	11	3 5 7 11 17 33 35 37 87 167 207
Samueli89 60	13	26	7 11 17 21 23 27 29 31 47 49 57 59 67 79 125 129 133 161 241 251 263 529 1217 2055 2239 4737
Shahein11 b	9	10	3 5 7 9 13 15 23 25 45 95
Shahein11 s2	11	19	3 5 11 17 19 23 27 31 33 35 53 63 91 109 145 165 281 305 321
Shi11 a	10	14	3 7 21 23 25 45 67 75 79 171 201 509 833 969
Shi11 g1	7	2	3 19
Shi11 l2	10	17	3 5 9 13 15 19 31 35 49 51 79 85 203 339 371 499 911
Shi11 s1 a	7	4	3 5 13 17
Shi11 s1 b	7	5	3 5 7 15 65
Shi11 s2	10	17	3 5 13 21 23 39 49 59 65 87 93 267 413 437 449 575 757
Shi11 x1	10	4	7 105 113 509
Shi11 y1	10	6	17 21 23 205 497 527
Shi11 y2	11	10	3 9 11 15 43 91 101 137 171 255
Unsharp 3-1	7	3	3 11 69
Unsharp 3-2	11	3	43 171 1109
Vinod03 15	11	4	97 101 391 1289
Vinod03 26a	6	7	3 7 9 11 19 31 47
Vinod03 26b	14	13	89 223 611 621 1027 2201 2503 3067 3645 4999 6053 8139 14433
Xu07 15	12	4	5 69 553 2483
Xu07 28	11	8	3 11 23 25 35 53 79 407
Yeung04 40	19	20	5 13 15 63 115 157 177 283 453 579 961 2113 3215 3401 3549 6981 31381 58565 116589 251821
Yli01 30	10	7	3 7 9 23 25 37 543
Yoshino90 64	13	25	5 9 11 15 17 19 23 39 41 51 53 61 63 65 67 85 93 107 133 141 167 215 243 391 623
Zahosam89 25	8	6	3 5 7 17 23 123

B Full results

Here are the results of the most efficient solving processes of Sect. 3, that is to say CP_{OL}^{T-OPT} – Choco and CP_{OL}^{T-OPT} – PicatSAT. Results with * correspond to Time-outs. Optimal values are then only lower bounds on the optimal value (due to outer loop process).

	CP_{OL}^{T-OPT} – Choco		CP_{OL}^{T-OPT} – PicatSAT	
Name	Time	Optimal value	Time	Optimal value
Bull91 32	0.009609594	10	0.016504137	10
Chang06 3	2.8938096	6	5.489361	6
Chen99 15	0.2031537	5	0.4127643	5
Chen99 28a	0.010915883	9	0.014854384	9
Chen99 28b	0.009550468	9	0.010016248	9
Chenyao01 28a	0.008628239	8	0.01576085	8
Chenyao01 28b	0.017995132	9	0.0153848	9
Dempster02 25	3600*	16	3600*	16
Dempster04 a3	0.33831158	4	0.5994603	4
Dempster04 b3	0.10752247	4	0.55014396	4
Dempster04 c3	12.990078	6	3.891557	6
Dempster04 d3	1.8966944	5	2.9508114	5
Dempster04 e3	3600*	7	132.47794	7
Dempster04 f3	3600*	8	47.145218	7
Dempster04 g4	1576.7051	8	94.156654	8
Dempster04 h4	3600*	9	77.52995	8
Dempster04 i5	2554.422	8	33.09207	8
Dempster04 j5	31.479206	8	8.579239	8
Dempster04 k5	1213.3083	9	396.40903	9
Dempster04 l5	759.65424	9	220.25233	9
Gaussian 3	0.0409617	4	0.583858	4
Gaussian 5	2.3252704	5	1.3561186	5
Goodman77 c7	0.007590095	1	0.013904119	1
Goodman77 d7	0.012374914	2	0.012586374	2
Goodman77 e11	0.014187181	3	0.013168614	3
Goodman77 f11	0.12346478	5	0.5190394	5
Goodman77 g11	0.3869104	4	0.5341624	4
Goodman77 h15	7.9795375	7	4.803683	7
Goodman77 i19	51.32192	7	4.260484	7
Highpass 15	0.034020726	12	0.011920487	12
Highpass 5	0.033752292	4	0.009327971	4
Highpass 9	0.013508394	5	0.018640144	5
Jain91 11	0.020691222	3	0.007354204	3
Jang02 11	0.03289043	3	0.007187529	3
Jheng04 29a	0.012659903	9	0.006951145	9
Jheng04 29b	0.012776701	9	0.008414842	9
Jheng04 30	0.014066492	8	0.009304543	8
Johansson08 28	0.015146864	12	0.011819075	12
Johansson08 30	0.9252687	29	33.023888	29
Johansson08 a5	1.3574243	6	4.307877	6
Johansson08 b5	1.2759924	7	3.8657022	7
Kwentus97 11	0.008800533	3	0.009988316	3
Kwentus97 15	0.6337919	4	1.3124253	4
Kwentus97 47	3600*	15	3600*	16
Laplacian 3	0.012318313	3	0.012874664	3
Lim83a 36	0.013083825	5	0.016002545	5
Lim83 121	3600*	52	312.22693	52
Lim83 36	0.02342677	5	0.010740752	5
Lim83 37	0.015166009	5	0.01164789	5
Lim83 63	0.012709469	22	0.013825828	22
Limakt08 121	0.050794814	42	0.02072016	42
Limpasko99 121	0.019122452	51	0.028886931	51
Limyu07 121	0.021527754	43	0.019105386	43
Lowpass 15	0.05772519	25	0.012771023	25
Lowpass 5	0.056310773	6	0.52233666	6
Lowpass 9	0.026176175	12	0.012262725	12

	CP_{OL}^{T-OPT} – Choco		CP_{OL}^{T-OPT} – PicatSAT	
Name	Time	Optimal value	Time	Optimal value
Martinez02 4	0.4625469	6	3.6224847	6
Nielsen89 67a	3600*	1,860	806.08154	27
Nielsen89 67b	3600*	1,850	72.73883	29
Rosa04 49	0.026622001	11	0.008026673	11
Samueli89 60	0.051398817	26	0.029395018	26
Shahein11 B	0.058135763	10	0.018511575	10
Shahein11 S2	0.023877047	19	0.013248091	19
Shi11 a	0.017881913	14	0.017379316	14
Shi11 g1	0.08914335	2	0.015650125	2
Shi11 l2	0.03802163	17	0.011217601	17
Shi11 s1 a	0.022314426	4	0.01272829	4
Shi11 s1 b	0.03334947	5	0.010201113	5
Shi11 s2	0.0658538	17	0.009312298	17
Shi11 x1	0.13736784	5	1.0136085	5
Shi11 y1	0.042791057	6	0.010200253	6
Shi11 y2	0.01902727	10	0.019007655	10
Unsharp 3-1	0.022333339	4	0.37976244	4
Unsharp 3-2	0.4529274	5	0.71737456	5
Vinod03 15	0.94390047	6	1.6572123	6
Vinod03 26a	0.06190169	7	0.008155939	7
Vinod03 26b	3600*	17	3600*	17
Xu07 15	0.9315285	5	0.49189606	5
Xu07 28	0.02134489	8	0.013357681	8
Yeung04 40	3600*	572	3600*	21
Yli01 30	217.13768	8	2.5888438	8
Yoshino90 64	0.036168657	25	0.015755763	25
Zahosam89 25	0.021439714	6	0.012248605	6

References

[1] Levent Aksoy, Eduardo Costa, Paulo Flores, and José Monteiro. Optimization of gate-level area in high throughput multiple constant multiplications. In *2011 20th European Conference on Circuit Theory and Design (ECCTD)*, pages 588–591, 2011.

[2] Levent Aksoy, Eduardo da Costa, Paulo Flores, and JosÉ Monteiro. Exact and approximate algorithms for the optimization of area and delay in multiple constant multiplications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(6):1013–1026, 2008.

[3] Levent Aksoy, Debapriya Basu Roy, Malik Imran, Patrick Karl, and Samuel Pagliarini. Multiplierless design of very large constant multiplications in cryptography. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 69(7):1716–1720, 2022.

[4] Krzysztof R. Apt and Peter Zoeteweij. A comparative study of arithmetic constraints on integer intervals, 2004.

[5] Robert Bernstein. Multiplication by integer constants. *Software: Practice and Experience*, 16(7):641–652, July 1986.

[6] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020. In *Proceedings of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[7] Hendrik Bierlee, Jip J. Dekker, Vitaly Lagoon, Peter J. Stuckey, and Guido Tack. Single constant multiplication for sat. In Bistra Dilkina, editor, *Integration of Constraint Programming, Artificial*

Intelligence, and Operations Research – 21st International Conference, CPAIOR 2024, Proceedings, volume 14742 of *Lecture Notes in Computer Science*, pages 84–98, Cham, Switzerland, 2024. Springer.

[8] David R. Bull and David H. Horrocks. Primitive operator digital filters. *IEE Proceedings G (Circuits, Devices and Systems)*, 138(3):401–412, 1991.

[9] Peter Cappello and Kenneth Steiglitz. Some complexity issues in digital signal processing. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 32(5):1037–1041, October 1984.

[10] Radoslaw Cymer. Dulmage-mendelsohn canonical decomposition as a generic pruning technique. *Constraints An Int. J.*, 17(3):234–272, 2012.

[11] Florent de Dinechin, Silviu-Ioan Filip, Martin Kumm, and Luc Forget. Table-based versus shift-and-add constant multipliers for fpgas. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 151–158. IEEE, June 2019.

[12] Florent de Dinechin and Martin Kumm. *Application-Specific Arithmetic*. Springer, 2024.

[13] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with flopoco. *IEEE Design & Test of Computers*, 28(4):18–27, 2011.

[14] Andrew G. Dempster and Malcolm D. Macleod. Constant integer multiplication using minimum adders. *IEE Proceedings - Circuits, Devices and Systems*, 141(5):407–413, 10 1994.

[15] Nicolai Fiege, Martin Kumm, and Peter Zipf. Bit-level optimized constant multiplication using boolean satisfiability. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 71(1):249–261, 2024.

[16] Rémi Garcia, Léo Pradels, Silviu-Ioan Filip, and Olivier Sentieys. Hardware-Aware Training for Multiplierless Convolutional Neural Networks. In *ARITH 2025 - 32nd IEEE International Symposium on Computer Arithmetic*, pages 1–8, El Paso, United States, May 2025. IEEE.

[17] Rémi Garcia and Anastasia Volkova. Toward the multiple constant multiplication at minimal hardware cost. *IEEE Transactions on Circuits and Systems I: Regular Papers*, pages 1–13, 2023.

[18] Rémi Garcia, Anastasia Volkova, and Martin Kumm. Truncated multiple constant multiplication with minimal number of full adders. In *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, Austin, Texas, United States, May 2022. IEEE.

[19] Rémi Garcia, Anastasia Volkova, Martin Kumm, Alexandre Goldsztejn, and Jonas Kühle. Hardware-aware design of multiplierless second-order iir filters with minimum adders. *IEEE Transactions on Signal Processing*, 70:1673–1686, 2022.

[20] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition edition, 1979.

[21] Sidinei Ghissoni, Eduardo Costa, Cristiano Lazzari, José Monteiro, Levent Aksoy, and Ricardo Reis. Radix-2 decimation in time (dit) fft implementation based on a matrix-multiple constant multiplication approach. In *2010 17th IEEE International Conference on Electronics, Circuits and Systems*, pages 859–862, 2010.

[22] Oscar Gustafsson. Lower bounds for constant multiplication problems. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 54(11):974–978, November 2007.

[23] Oskar Gustafsson. Towards optimal multiple constant multiplication: A hypergraph approach. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems and Computers (ACSSC)*, pages 1805–1809, Pacific Grove, CA, USA, October 2008. IEEE.

[24] Tobias Habermann, Jonas Kühle, Martin Kumm, and Anastasia Volkova. Hardware-aware quantization for multiplierless neural network controllers. In *2022 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pages 541–545, 2022.

[25] Richard W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, April 1950.

[26] Reid M. Hewlitt and Earl S. Swartzlander. Canonical signed digit representation for fir digital filters. In *2000 IEEE Workshop on Signal Processing Systems. SiPS 2000. Design and Implementation.* IEEE, 2000. Cat. No.00TH8528.

[27] Charles D. Howard, Linda S. DeBrunner, and Victor DeBrunner. Hybrid mcm implementation for fir filters in fpga devices. In *2014 IEEE International Conference on Electro/Information Technology (EIT)*, pages 1–6. IEEE, June 2014.

[28] Martin Kumm. *Multiple Constant Multiplication Optimizations for Field Programmable Gate Arrays.* Springer, Wiesbaden, Germany, October 2015.

[29] Martin Kumm. *Multiple Constant Multiplication Optimizations for Field Programmable Gate Arrays.* Springer Fachmedien Wiesbaden, Wiesbaden, 2016.

[30] Martin Kumm. Optimal constant multiplication using integer linear programming. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 65(5):567–571, May 2018.

[31] Martin Kumm, Anastasia Volkova, and Silviu-Ioan Filip. Design of optimal multiplierless fir filters with minimal number of adders. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(2):658–671, 2023.

[32] Martin Kumm and Peter Zipf. High speed low complexity fpga-based fir filters using pipelined adder graphs. In *2011 International Conference on Field-Programmable Technology (FPT)*. IEEE, December 2011.

[33] Martin Kumm, Peter Zipf, Mathias Faust, and Chip-Hong Chang. Pipelined adder graph optimization for high speed multiple constant multiplication. In *2012 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 49–52. IEEE, May 2012.

[34] Vitaly Lagoon and Amit Metodi. Deriving optimal multiplication-by-constant circuits with a sat-based constraint engine. In *ModRef 2020 – The 19th Workshop on Constraint Modelling and Reformulation*, September 2020. September 2020.

[35] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

[36] Hongli Li, Georges Grinstein, and Loura Costello. A visual canonical adjacency matrix for graphs. In *2009 IEEE Pacific Visualization Symposium (PacificVis)*, pages 89–96. IEEE, 2009.

[37] Pramod Kumar Meher and Yu Pan. Mcm-based implementation of block fir filters for high-speed and low-power applications. In *Proceedings of the IEEE/IFIP 19th International Conference on Very Large Scale Integration and System-on-Chip (VLSI-SoC)*, pages 118–121. IEEE, October 2011.

[38] Konrad Möller, Martin Kumm, Marco Kleinlein, and Peter Zipf. Pipelined reconfigurable multiplication with constants on fpgas. In *Proceedings of the 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6. IEEE, September 2014.

[39] Konrad Möller, Martin Kumm, Marco Kleinlein, and Peter Zipf. Reconfigurable constant multiplication for fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(6):927–937, June 2017.

[40] Singapore Nanyang Technological University. FIRsuite: Suite of Constant Coefficient FIR Filters. `https://www.firsuite.net/FIR/FromPublication`, November 2023.

[41] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard cp modelling language. In Christian Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007.

[42] Xiao Peng and Christine Solnon. Using canonical codes to efficiently solve the benzenoid generation problem with constraint programming. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:17, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[43] Charles Prud'homme and Jean-Guillaume Fages. Choco-solver: A java library for constraint programming. *Journal of Open Source Software*, 7(78):4708, 2022.

[44] George W. Reitwiesner. Binary arithmetic. In Franz L. Alt, editor, *Advances in Computers*, volume 1, pages 231–308. Academic Press, 1960.

[45] Neil James Alexander Sloane and Simon Plouffe. *The Encyclopedia of Integer Sequences*. Academic Press, 1995.

[46] James B. Wendt, Nathaniel A. Conos, and Miodrag Potkonjak. Multiple constant multiplication implementations in near-threshold computing systems. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1071–1075. IEEE, April 2015.

[47] Neng-Fa Zhou and Håkan Kjellerstrand. Optimizing sat encodings for arithmetic constraints. In *Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming (CP 2017)*, volume 10613 of *Lecture Notes in Computer Science*, pages 824–840. Springer, 2017.

[48] Neng-Fa Zhou and Håkan Kjellerstrand. The picat-sat compiler. In Marco Gavanelli and John H. Reppy, editors, *Practical Aspects of Declarative Languages (PADL 2016)*, volume 9585 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2016.

A new Constraint Programming model for the Multiple Constant Multiplication

Théo Cantaloube
Ensimag, Grenoble, France
Inria, INSA Lyon, CITI, UR3720, 69621 Villeurbanne, France

Xiao Peng
LAAS-CNRS, Université de Toulouse, Toulouse, France

Christine Solnon
INSA Lyon, Inria, CITI, UR3720, 69621 Villeurbanne, France

Anastasia Volkova
Inria, INSA Lyon, CITI, UR3720, 69621 Villeurbanne, France

Abstract

The Multiple Constant Multiplication (MCM) problem arises in many applications such as, for example, digital signal processing. Given a set T of target constants, the goal of MCM is to find the most efficient way for multiplying an input number with each constant in T, where multiplications are realized through bit-shifts and additions, and where intermediate results may be shared to produce different target constants. Different metrics may be considered for evaluating the cost of a solution, and a classical objective function is to minimize the number of adders. State-of-the-art methods, based on Integer Linear Programming (ILP), suffer from numerous performance and scalability bottlenecks. In this work, we propose for the first time a Constraint Programming (CP) model for minimizing the number of adders for the MCM.

Compared to the state-of-the-art ILP approach, CP does not suffer from the curse of linearization, hence permits significantly simpler formulations of the mathematical model. In order to evaluate our CP model, we focus on a widely used benchmark extracted from a collection of digital filter designs and compare ourselves with state-of-the-art ILP and SAT models. We show that our CP approach is less efficient on some easy instances, but more efficient on hard instances. We also introduce a pseudo-polynomial time algorithm which is able to solve some instances, and show that using this algorithm during a preprocessing step improves the solution process.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Constraint Programming, Multiple Constant Multiplication, Hardware Optimization

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

1.1 Motivation

Electronic devices with embedded computations are everywhere: in hearing-aids, smart-watches, cars, or drones, for example. The strict resource and power constraints call for highly optimized implementations. Instead of using standard data formats and generic arithmetic units, hardware designers often use code generators to implement individual or compound operators tailored to the application's requirements.

In this context, a common type of arithmetic operator is Multiplication by Multiple Constants (MCM), where an integer number is multiplied by several target constants known at the design time. The idea is that by exploiting the a priori knowledge of the constants, one can avoid costly generic multiplier circuits and design a custom optimized computational implementation for a given target constant set. MCM is used in digital signal processing for


Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

linear digital filter evaluation [1,13,19,22,26], Discrete Transforms (Discrete Cosine Transform, Fast Fourier Transform) [15, 32], as well as for inference of Deep Neural Networks [10, 17].

Multiplications by constants are commonly done using additions/subtractions and bit-shifts: shifting a number X by k bits on the left (or the right) is equivalent to multiplying X by 2^k (or 2^{-k}). This is called a *shift-and-add* approach. Shifts are basically wiring that is free in hardware, hence designers usually only minimize the number of additions/subtractions (which we simply refer to as adders). This is a good high-level metric of the final hardware cost, though other low-level ones that count logic gates exist.

Let us first illustrate our problem on a single constant multiplication (SCM) problem. Consider the multiplication of an integer variable X by 93, written as 1011101 in binary. Hence, $93X = 2^6X + 2^4X + 2^3X + 2^2X + X$ and a naive implementation of a shift-and-add approach for $93X$ is:

$$80X = 2^6X + 2^4X; \quad 88X = 80X + 2^3X; \quad 92X = 88X + 2^2X; \quad 93X = 92X + X$$

However, this idea does not take into account the possibility of making subtractions instead of additions. Thanks to the *Canonical Signed Digit* representation of a number [18,31], we obtain that $93X = 2^7X - 2^5X - 2^2X + X$ so a second implementation of a shift-and-add approach for $93X$ is:

$$96X = 2^7X - 2^5X; \quad 92X = 96X - 2^2X; \quad 93X = 92X + X$$

Unfortunately, neither of these approaches are guaranteed to minimize the number of additions. Indeed, the optimal implementation is:

$$31X = 2^5X - X; \quad 93X = 2^1(31X) + 31X$$

You can notice that previous approaches went wrong because they forced every addition to have X as one of the inputs, therefore forbidding additions between two previously computed multiplications.

In MCM, several constants must be multiplied by the same input X. For example, consider the target set of constants $\{7, 19, 31\}$.

A naive implementation of the MCM could consist in optimizing each constant individually:

$$\begin{aligned} 7X &= 2^3X - X \\ 15X &= 2^4X - X; \quad 19X = 15X + 2^2X \\ 31X &= 2^5X - X \end{aligned}$$

However, optimizing each multiplication separately does not ensure that the total number of additions is minimized, since intermediate terms can actually be shared. Indeed, the optimal shift-and-add implementation for this target set is:

$$7X = 2^3X - X; \quad 31X = 2^5X - X; \quad 19X = 2^{-1}(7X + 31X) \tag{1}$$

Note that here we used a right-shift to divide the even value $38X$ by 2 to obtain $19X$.

Given a set of target constants, the MCM Problem aims at finding a shift-and-add implementation that produces every target constant and minimizes a given metric.

There are a lot of MCM non-exclusive variations relying on different metrics used as objective functions to minimize [2, 3, 12, 23]. In this paper, we will focus on the simplest metric: minimizing the number of adders [11, 25]. Although it does not perfectly reflect the actual implemented hardware cost, this high-level metric is a good proxy for a cost function.

It is generally accepted that MCM is an $\mathcal{NP}$-hard problem from [6]. Historically, first papers to address the MCM problem were using heuristics, most of the time based on greedy algorithms [3, 4, 8, 24]. Then Integer Linear Programming (ILP) models were implemented, first by tabulating all possible factors that may occur in an optimal solution in preprocessing computations [3, 16, 20]. However, the size of these number tables grows exponentially with the word-length of target constants, then leading to models requiring highly demanding calculations.

More general new ILP models approached the MCM problem by initializing a lower bound k to the number of target constants, and by iteratively solving the decision problem "Can the MCM be solved with exactly k adders?" until reaching the smallest k for which the answer is yes [21]. This is called an "outer loop process". Instead of solving a sequence of decision problems, authors of [11, 21] introduced a model that directly minimizes the number of adders (while initializing k to an upper bound computed with a good heuristic). Due to the linearization necessity implied by the use of ILP models, these methods suffer from a lack of expressiveness, which translates into really complicated models and also undergo numerical instabilities when the target constants become large.

In the recent paper [9] was proposed a SAT model solving the MCM problem using an outer loop process. Finally, in [5] were introduced SAT models minimizing the number of full/half adders, another lower-level metric, to solve the Single Constant Multiplication Problem. Nevertheless, in this paper, we propose for the first time an approach for the MCM problem from the perspective of Constraint Programming.

1.2 Overview of the paper

In Section 2, we go deeper into some existing models and properties of the MCM problem. In Section 3, we introduce our CP model. In Section 4, we experimentally evaluate our CP model and show that it outperforms the ILP approach of [11] and is competitive with the SAT approach of [9]. In Section 5, we consider a simpler decision problem, that aims at deciding whether their exists a solution when the number of adders is fixed to the number of constant targets. We introduce a pseudo-polynomial time algorithm for this problem, and show that half of the benchmark instances can be solved with this algorithm because the answer is yes. We also show that using this algorithm during a preprocessing step significantly improves the solution process.

2 Problem statement and properties

The first input of an MCM instance is the set T of target constants. Intermediate values used to produce these target constants are called *fundamentals*. Constants that are not used to produce other constants are called *free constants*. For example, in Eq. 1 target constants are 7, 19, and 31. While 19 is a free constant, 7 and 31 are not free as they are used to produce 19.

The second input of an MCM instance is the word-length, *i.e.*, the number of bits w used to encode fundamentals. In this paper, we fix $w = \max_{j \in [\![1,|T|]\!]} \lceil \log_2(T_j) \rceil$.

We also assume, based on experimentation, that intermediate values of fundamentals can be represented with $w + 1$ bits. Indeed, optimal solutions can often pass by a fundamental that is larger than the largest target constant. For example, in Eq. 1, the target set is representable with 5 bits but the last adder requires 6 to represent the fundamental $7 + 31 = 38 > 31 = 2^5 - 1$.

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2.1 Adder modeling

For each adder a, we note $c_{a,l}$, $c_{a,r}$, and c_a the left operand, the right operand, and the result. As each operand is shifted before being added, a first possibility is to use two shift variables for representing the number of bit-shifts that must be performed on the left and right operands, respectively. As shifts may either be to the left (to multiply by positive powers of 2) or to the right (to multiply by negative powers of 2, i.e. divide), these variables are assigned to positive values when the shift is to the left, and to negative values when it is to the right.

We can reduce the domain of shift variables by considering a theorem of [8] according to which we can impose that either the shift on the left operand is positive and there is no shift on the right operand, or both shifts are negative and equal, so that they can be applied after the addition. Then we obtain [11]:

$$c_a = 2^{-s_a^r}[(-1)^{\sigma_{a,l}} 2^{s_a^l} c_{a,l} + (-1)^{\sigma_{a,r}} c_{a,r}] \tag{2}$$

where s_a^l is the left shift applied to the left input, s_a^r is the right shift applied to the sum of the left and right inputs, and $\sigma_{a,l}$ and $\sigma_{a,r}$ are signs of the left and right inputs. Eq. 2 is equivalent to:

$$2^{s_a^r} c_a = (-1)^{\sigma_{a,l}} 2^{s_a^l} c_{a,l} + (-1)^{\sigma_{a,r}} c_{a,r} \tag{3}$$

Note that $s_a^l, s_a^r \in \mathbb{N}$ instead of $\mathbb{Z}$. These notations are summed up in Fig. 1.

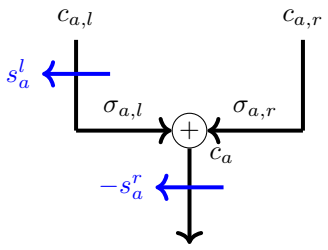


Figure 1 Optimized modeling of adders

▶ **Example 2.1.** Let us consider the shift-and-add solution Eq. 1. For the adder producing $7 = 2^3 - 1$ we have $c_a = 7$, $c_{a,l} = c_{a,r} = 1$, $s_a^l = 3$, $s_a^r = 0$, $\sigma_{a,l} = 0$ and $\sigma_{a,r} = 1$. For the adder producing $19 = 2^{-1}(31 + 7)$, we have $c_a = 19$, $c_{a,l} = 7$, $c_{a,r} = 31$, $s_a^l = 0$, $s_a^r = 1$, $\sigma_{a,l} = \sigma_{a,r} = 0$.

2.2 Properties of optimal solutions

Here is a basic property of optimal adders:

▶ **Property 1.** *In optimal implementations, every free constant is a target constant.*

Proof. A free constant is not used to produce other constants. If it is not a target constant, then we can remove it from the circuit. But the implementation is supposed to be optimal, so this is a contradiction. ◀

Keep in mind that the converse is false: a target constant may be used to compute other target constants (for example, in Eq. 1, 7 and 31 are not free constants).

Also, if we denote MCM-Adders(T, w) the minimal number of adders required to produce the target constants set T on w bits, we have:

▶ **Property 2.** *MCM-Adders*$(T, w) \geq |T|$

Proof. This is simply because we need an adder output producing each target constant. ◀

2.3 Preprocessing of target constants

The target constant set T is preprocessed before launching the solving process, as proposed in [8]. First, we replace every target constant by its absolute value, as the sign may be adjusted later. Second, we divide every target constant by 2 until it becomes odd, as a shift can be applied to retrieve the original even-valued constant. Finally, we remove 1 from the resulting set of target constants, as it can be obtained for free from the initial input. More formally, T is replaced with $\{odd(|T_j|), T_j \in T\} \setminus \{1\}$ where the operator odd is defined as:

$$odd(x) = \frac{x}{\max\limits_{n \geq 0}(gcd(x, 2^n))}$$

As a consequence, every preprocessed target constant is odd and greater than 1.

3 Constraint Programming model

In this section, we first introduce a CP model for the decision problem where the number of adders is given. Then, we show how to use this model to solve the optimization problem.

3.1 Fixed Number of adders Decision Problem

In this section, we assume that the number of adders is fixed to N.

Variables

For each adder $a \in [\![1, N]\!]$, we introduce variables to represent its operands and output:

- $c_a \in [\![1, 2^w - 1]\!]$ is the output constant;
- $c_{a,l} \in [\![1, 2^w - 1]\!]$ and $c_{a,r} \in [\![1, 2^w - 1]\!]$ are the left and right inputs.

We also introduce a variable c_0 which models the initial input and which is constrained to be equal to 1.

Variables $c_a, c_{a,l}$, and $c_{a,r}$ are related with sign values ($\sigma_{a,l}$ and $\sigma_{a,r}$) and shift values (s^l_a and s^r_a) as defined in Eq. 3. Instead of introducing sign and shift variables, as done in ILP models, we gather them in k_a, $k_{a,l}$, and $k_{a,r}$ variables, which are further gathered in c^{nsh}_a, $c^{sh,sg}_{a,l}$ and $c^{sh,sg}_{a,r}$ variables as displayed below:

$$\underbrace{\underbrace{2^{s^r_a}}_{k_a} c_a}_{c^{nsh}_a} = \underbrace{\underbrace{(-1)^{\sigma_{a,l}} 2^{s^l_a}}_{k_{a,l}} c_{a,l}}_{c^{sh,sg}_{a,l}} + \underbrace{\underbrace{(-1)^{\sigma_{a,r}}}_{k_{a,r}} c_{a,r}}_{c^{sh,sg}_{a,r}} \tag{4}$$

Hence, for each adder $a \in [\![1, N]\!]$, we introduce the following variables to store intermediate results:

- $k_a \in \{2^s \mid s \in [\![0, w]\!]\}$;
- $k_{a,l} \in \{-2^s, 2^s \mid s \in [\![0, w]\!]\}$;
- $k_{a,r} \in \{-1, 1\}$;
- $c^{nsh}_a \in [\![0, 2^{w+1}]\!]$;
- $c^{sh,sg}_{a,l} \in [\![-2^{w+1}, 2^{w+1}]\!]$ and $c^{sh,sg}_{a,r} \in [\![-2^w + 1, 2^w - 1]\!]$.

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

C1: $\forall\ a \in [\![1, N]\!],\ k_a \times c_a = c_a^{nsh}$
C2: $\forall\ a \in [\![1, N]\!],\ k_{a,l} \times c_{a,l} = c_{a,l}^{sh,sg}$ and $k_{a,r} \times c_{a,r} = c_{a,r}^{sh,sg}$
C3: $\forall\ a \in [\![1, N]\!],\ c_{a,l}^{sh,sg} + c_{a,r}^{sh,sg} = c_a^{nsh}$
C4: $\forall\ a \in [\![1, N]\!],\ k_{a,l} > 0$ or $k_{a,r} > 0$
C5: $\forall\ a \in [\![1, N]\!],\ k_a + k_{a,l} \neq 2$ and $k_a + k_{a,l} \neq 0$
C6: $T \subseteq \{c_a \mid a \in [\![1, N]\!]\}$
C7: $c_0 = 1$
C8: $\forall\ a \in [\![1, N-1]\!],\ \{a' \in [\![k+1, N]\!] \mid ind_{a',l} = a\ \vee\ ind_{a',r} = a\} = \emptyset \implies c_a \in T$
C9: $\forall\ a \in [\![1, N]\!],\ c_{a,l} = c_{ind_{a,l}}$ and $c_{a,r} = c_{ind_{a,r}}$
C10: allDifferent$((c_a)_{a \in [\![0,N]\!]})$
C11: $\forall\ a \in [\![1, N]\!],\ c_a \equiv 1[2]$

Figure 2 Constraints for the Decision problem, when the number of adders is fixed to N

Finally, for each adder $a \in [\![1, N]\!]$, we introduce variables for linking the left and right inputs of a with the output of another adder. To ensure that there is no cycle in this dependency relation, the domain of these variables only contains values smaller than a:

- $ind_{a,l} \in [\![0, a-1]\!]$ and $ind_{a,r} \in [\![0, a-1]\!]$

Constraints

Constraints are listed in Fig. 2.

Constraints C1 to C3 are straightforward consequences of Eq. 4.

As fundamentals cannot have negative values, we know that $k_{a,l}$ and $k_{a,r}$ cannot be both negative. This is expressed by Constraint C4.

As stated in Section 2.1, whenever there is no right shift, *i.e.*, $k_a = 1$, then there is a left shift, *i.e.*, $k_{a,l} \notin \{-1, 1\}$. This is ensured by Constraint C5.

To be a valid MCM implementation for the given target set T, adders must produce every target constant. This is realized by Constraint C6.

Constraint C7 initializes the origin input.

According to the Property P. 1, free constants are target constants. First member of the implication of Constraint C8 is equivalent to: "If adder a produces a free constant..." and second member is equivalent to "...then it is a target constant.".

Constraint C9 links the left and right inputs of each adder a with the output of adders $ind_{a,l}$ and $ind_{a,r}$, respectively.

Finally, constraints C10 and C11 ensure that the generated values are all different and odd, respectively.

Search strategy

Random exploration of the search space in the MCM problem rarely produces valid or meaningful solutions. Fixing the fundamental of the second adder before the first adder generally results in failure. In contrast, constructing the adder graph incrementally, by adding one adder at a time in a well-defined order, is more likely to yield a valid solution. Thus, we impose a Lower Bound First strategy, and the variable ordering heuristic is detailed in Fig. 3.

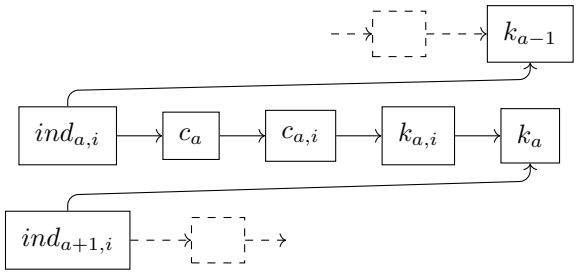


Figure 3 Branching priority for Constraint Programming modeling

3.2 Minimization Problem

To solve the MCM minimization problem, we solve a sequence of decision problems: we initialize N to $|T|$, and then we iteratively solve the model defined in Section 3.1 and increment N until we find a solution.

4 Experimental Evaluation

Our CP model has been implemented in Java with Choco-solver [30]. Experiments were conducted on a machine equipped with an Intel® Xeon® Gold 6444Y processor (16 cores, 32 threads, 45 MB L3 cache) with 256 GB RAM, running on an x86_64 architecture with support for AVX-512 and related vector instruction sets. We call this approach CP – Choco.

In addition to CP – Choco, we propose one more model using Minizinc [28]. As we heard how efficient SAT solvers (directly written in SAT) were for the MCM problem resolution, we used Minizinc with CP models in front-end, PicatSAT [33] in middle-end (to translate CP into SAT) and KisSAT in back-end. We denote as CP – PicatSAT our CP model with this approach.

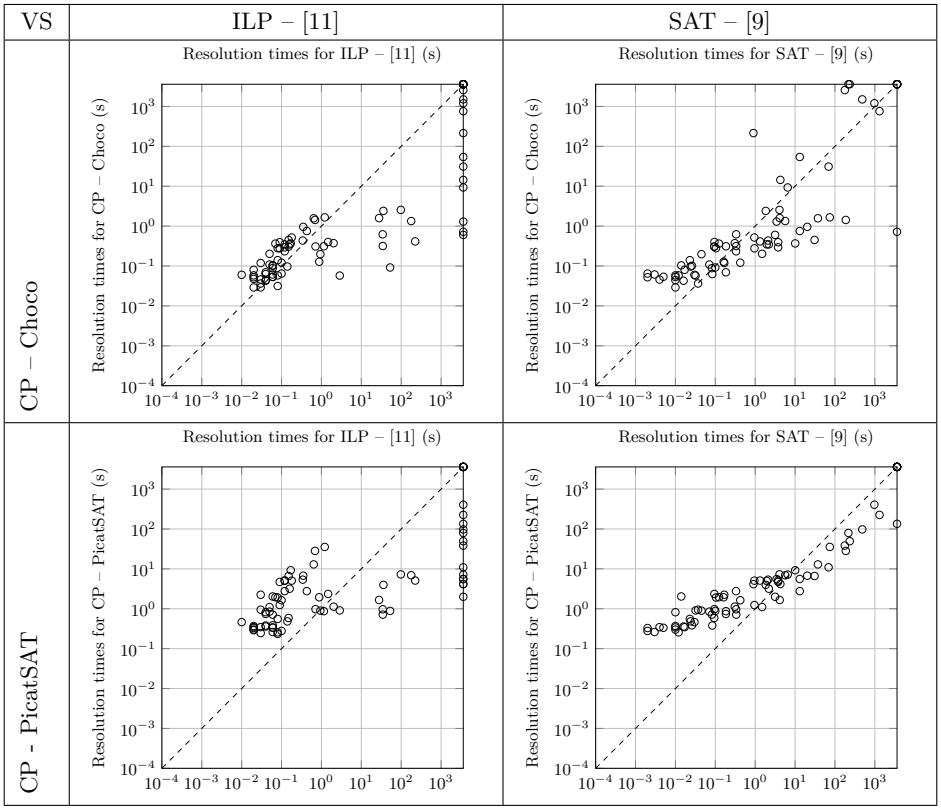
The code written for these models is available as open source[1].

The benchmark we have worked on is extracted from a collection of linear digital filters [27] that can be implemented with the MCM problem. It has been widely used in the literature and for the ILP-based approaches in particular [11, 21, 24]. It contains 83 instances with $|T|$ ranging from 1 to 51 constants and w from 4 to 19 bits.

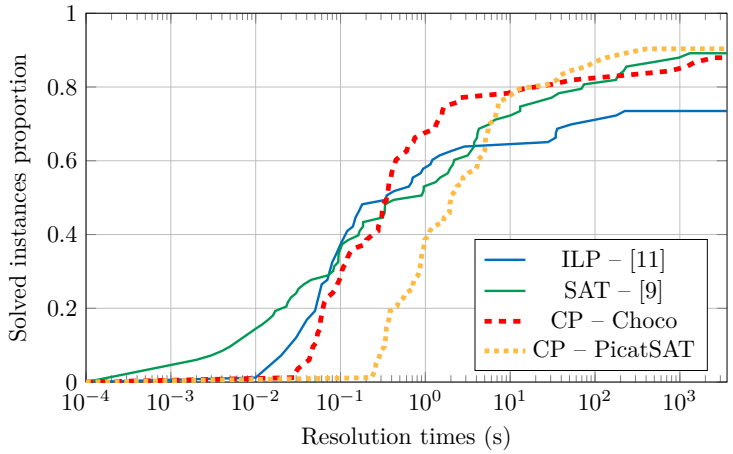
In Fig. 4(a), we compare our CP-based approach with the ILP approach of [11] and the SAT approach of [9] on a per instance basis. ILP – [11] and SAT – [9] are faster on "easy" instances, solved in less than 1s by both approaches. However, our approaches are faster on harder instances. While, in a run time limit of 3600s, CP – Choco and CP – PicatSAT are respectively able to solve 73 and 75 out of 83 instances, ILP – [11] and SAT – [9] respectively solved 61 and 74 out of 83 instances. To give more details, whenever CP – Choco or CP – PicatSAT resulted with a Time-Out, ILP – [11] resulted with a Time-out and whenever CP – PicatSAT resulted with a Time-Out, SAT – [9] resulted with a Time-out. When looking at Fig. 4(b), that displays the evolution of the cumulative percentage of solved instances with respect to time, we note that ILP – [11] and SAT – [9] are more successful than CP – Choco and CP – PicatSAT when the time limit is smaller than 0.1s, whereas SAT – [9], CP – Choco and CP – PicatSAT is more successful than ILP – [11] for longer time limits.

In addition, we can see how much better suited CP – Choco is to the combinatorial

[1] `https://gitlab.inria.fr/emeraude/mcm-cp`



(a) Comparing CP – Choco and CP – PicatSAT to ILP – [11] and SAT – [9]



(b) Distribution of the solved instances proportion through time

Figure 4 Comparative analysis of ILP and CP resolution times.

aspects of the MCM problem than ILP – [11] by comparing the number of variables in the respective models. ILP – [11] uses $2 + N(10 + |T| + 2w + (N+1)/2)$ variables whereas CP – Choco uses $11N$ variables. We can notice that unlike ILP – [11], there is no dependence on the number of target constants or on the word-length and the dependence on N is linear.

5 Preprocessing step

In this section, we study the decision problem that aims at deciding whether there exists a solution with $|T|$ adders. We show that this problem may be solved in pseudo-polynomial time, and that the solution process is improved when solving this problem in a preprocessing step.

Let us first formally define this decision problem:

Problem: T-OPT
Input: Target constant set T and word-length w
Question: Do we have MCM-Adders$(T, w) = |T|$?

Before delving into the algorithm, we first need to show how to efficiently determine if z can be produced with x and y and inputs (we denote this as $\mathsf{can}(x, y, z)$). We could do $\mathsf{can}(x, y, z) = (z \in \mathcal{A}_w(x, y))$ where $\mathcal{A}_w(x, y)$ represents the set of fundamentals written on w bits "reachable" from inputs x and y but computing $\mathcal{A}_w(x, y)$ is not optimal (the table is too large and it takes too much time to generate). Instead, we can use the operator *odd* and do:

$$\mathsf{can}(x, y, z) = \left(\exists\, (\sigma_1, \sigma_2) \in \{0, 1\}^2 \setminus (1, 1) \;\middle|\; \begin{array}{rl} & z = odd((-1)^{\sigma_1} x + (-1)^{\sigma_2} y) \\ \text{or} & (-1)^{\sigma_1} x = odd(z - (-1)^{\sigma_2} y) \\ \text{or} & (-1)^{\sigma_2} y = odd(z - (-1)^{\sigma_1} x) \end{array} \right)$$

▶ **Example 5.1.** Let us use the operator can on $T = \{7, 19, 31\}$ (cf Eq. 1). $\mathsf{can}(1, 1, 7) = true$ because $1 \times 2^3 - 1 = 7$ so $odd(7 + 1) = 1$. $\mathsf{can}(1, 1, 31) = true$ because $1 \times 2^5 - 1 = 31$ so $odd(31 + 1) = 1$. $\mathsf{can}(7, 31, 19) = true$ because $2^{-1} \times (7 + 31) = 19$ so $odd(7 + 31) = 19$.

The time complexity for computing can is logarithmic in the number of bits because of the operator *odd* [7, chapter 10].

▶ **Theorem 5.2.** *T-OPT can be solved in pseudo-polynomial time* $\mathcal{O}(log(w) \times |T|^3)$.

Proof. The pseudo-polynomial algorithm is displayed in Alg. 1. We define the depth of an adder as the length of the longest path from the initial input 1, as illustrated in Fig. 5. The idea is to process depth by depth and build R, the set of reached target constants, while Q is the queue of target constants waiting to be processed. We start from 1 and we compute $\mathsf{can}(1, 1, T_j),\ \forall T_j \in T$. We can repeat the process until we have found all target constants, as illustrated in Fig 5. The algorithm will necessarily terminate because $|T| =$ MCM-Adders(T, w). The while and for loops on lines 3, 5 and 6 of Alg. 1 run at most $|T|$ times. Hence, the time complexity of Alg. 1 is $\mathcal{O}(log(w) \times |T|^3)$. ◀

A straightforward consequence of Th. 5.2 is that any instance (T, w) such that MCM-Adders$(T, w) = |T|$ may be solved in pseudo-polynomial time. Hence, we propose to run Alg. 1 in a preprocessing step. If the algorithm returns true, then *prec* is a solution and the instance is solved; otherwise we can set the lower bound on the number of adders to $|T| + 1$, and then solve the instance with CP. We call these new approaches CP T-OPT – Choco and CP T-OPT – PicatSAT.

Algorithm 1 Pseudo-polynomial implementation of MCM on T-OPT instances

Input: Target constant set T and word-length w
Output: $(A, prec)$ such that A is the answer for T-OPT(T, w) and if A=true, then $prec$ is a predecessor function corresponding to an MCM implementation

```
function MCM-Adders_pseudo-poly(T)
    Let R = {1} and W = {1} be sets of constants
    while W ≠ ∅ do
        Pop x from W
        forall z ∈ T \ R do
            forall y ∈ R \ W do
                if can(x, y, z) then
                    prec(z) ← (x, y)
                    R ← R ∪ {z}
                    W ← W ∪ {z}
                    Break
        if R = T then
            return (True, prec)
    return (False, ∅)
```

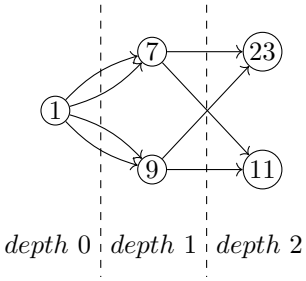
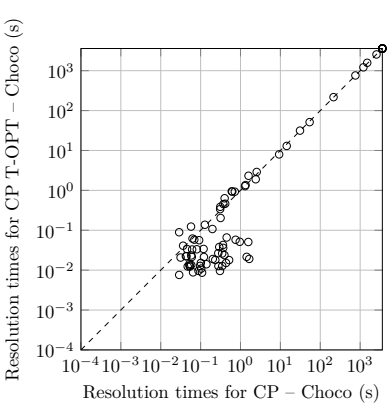


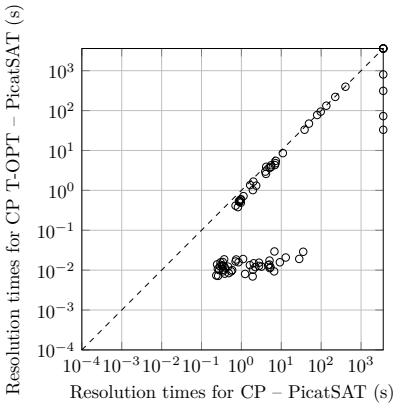
Figure 5 Illustration of Alg. 1 on the toy instance ($T = \{1, 7, 9, 11, 23\}$, $w = 5$). We have $\mathsf{can}(1, 1, 7) = \mathsf{can}(1, 1, 9) = true$ because $7 = 2^3 - 1$ and $9 = 2^3 + 1$. Hence, at the end of the first iteration of the while loop, we have $R = \{1, 7, 9\}$ and $W = \{7, 9\}$. We have $\mathsf{can}(7, 9, 11) = \mathsf{can}(7, 9, 23) = true$ because $11 = 2 \times 9 - 7$ and $23 = 2 \times 7 + 9$. Hence, at the end of the second iteration of the while loop, we have $R = \{1, 7, 9, 11, 23\}$ and $W = \{23, 11\}$. Then $R = T$ so the algorithm will return $prec = \{7 \leftarrow (1, 1), 9 \leftarrow (1, 1), 11 \leftarrow (7, 9), 23 \leftarrow (7, 9)\}$

When running Alg. 1 on our benchmark, we noticed that the answer was true for 44 out of the 83 MCM instances. In Fig. 6(a) and (b), we compare CP T-OPT – Choco with CP – Choco and CP T-OPT – PicatSAT with CP – PicatSAT. It shows us that the preprocessing step allows us to significantly reduce the solution time for many instances, those for which MCM-Adders$(T, w) = |T|$, whereas solving times are not significantly changed for the other instances (as the running time of Algo. 1 is largely negligible compared to the running time of CP). When looking at Fig. 6(c), that displays the evolution of the cumulative percentage of solved instances with respect to time, we note that CP T-OPT – Choco and CP T-OPT – PicatSAT are always better than ILP – [11] and SAT – [9]. CP T-OPT – PicatSAT is now able to solve 79 out of 83 instances of the benchmark. This improvement is a direct consequence of the preprocessing step: by avoiding unnecessary resolution of the decision

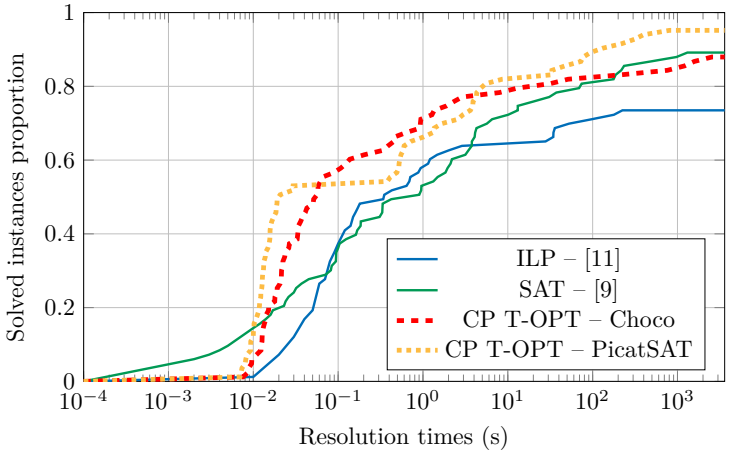
problem when the number of adders equals the number of target constants, more time remains available to solve harder instances before reaching the timeout.



(a) CP – Choco vs CP T-OPT – Choco



(b) CP – PicatSAT vs CP T-OPT – PicatSAT



(c) Distribution of the solved instances proportion through time

Figure 6 Comparative analysis of ILP – [11], SAT – [9], CP T-OPT – Choco and CP T-OPT – PicatSAT resolution times.

6 Conclusion and perspectives

In this paper, we introduced a new Constraint Programming model to solve the Multiple Constant Multiplication problem. We demonstrated that it outperforms previous ILP-based approaches and is very competitive in comparison to SAT models. We have also shown that half of the usual benchmark MCM instances do not require modeling and optimization paradigms but only a pseudo-polynomial algorithm.

Further works will focus on the improvement of the scalability with respect to the word-length (with $w > 20$ bits). Moreover, it seems that the demonstration of $\mathcal{NP}$-hardness of the MCM problem from [6] is not correct because it assumes that the MCM problem is the same as the Ensemble Computation problem proven $\mathcal{NP}$-hard in [14], which is false. Thus, a proper demonstration of the $\mathcal{NP}$-hardness of the MCM problem would be useful.

We also plan to extend our model so that it directly solves the minimization problem, instead of solving a sequence of decision problems with fixed numbers of adders. The key point is to model the fact that some adders are not used. This should allow us to improve performance of CP.

Finally, we plan to approach the MCM problem through the prism of fixed topology and decision problems. The idea is that fixing the topology of the shift-and-add implementation (the adder nodes and the arcs) facilitates the search for the corresponding shifts, signs and fundamentals. Hence, we plan to enumerate every graph topology up to a reasonable size to facilitate the optimization. In order to eliminate symmetries we plan to investigate the interest of using canonical codes based on Breadth-First Search, as proposed in [29].

References

1 Levent Aksoy, Eduardo Costa, Paulo Flores, and Jose Monteiro. Optimization of area in digital fir filters using gate-level metrics. In *2007 44th ACM/IEEE Design Automation Conference*, pages 420–423, 2007. doi:10.1109/DAC.2007.375200.

2 Levent Aksoy, Eduardo Costa, Paulo Flores, and José Monteiro. Optimization of gate-level area in high throughput multiple constant multiplications. In *2011 20th European Conference on Circuit Theory and Design (ECCTD)*, pages 588–591, 2011. doi:10.1109/ECCTD.2011.6043602.

3 Levent Aksoy, Eduardo da Costa, Paulo Flores, and JosÉ Monteiro. Exact and approximate algorithms for the optimization of area and delay in multiple constant multiplications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(6):1013–1026, 2008. doi:10.1109/TCAD.2008.923242.

4 Robert Bernstein. Multiplication by integer constants. *Software: Practice and Experience*, 16(7):641–652, July 1986. doi:10.1002/spe.4380160704.

5 Hendrik Bierlee, Jip J. Dekker, Vitaly Lagoon, Peter J. Stuckey, and Guido Tack. Single constant multiplication for sat. In Bistra Dilkina, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research – 21st International Conference, CPAIOR 2024, Proceedings*, volume 14742 of *Lecture Notes in Computer Science*, pages 84–98, Cham, Switzerland, 2024. Springer. doi:10.1007/978-3-031-60597-0_6.

6 Peter Cappello and Kenneth Steiglitz. Some complexity issues in digital signal processing. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 32(5):1037–1041, October 1984. doi:10.1109/TASSP.1984.1164457.

7 Florent de Dinechin and Martin Kumm. *Application-Specific Arithmetic*. Springer, 2024. URL: https://link.springer.com/book/10.1007/978-3-031-42808-1.

8 Andrew G. Dempster and Malcolm D. Macleod. Constant integer multiplication using minimum adders. *IEE Proceedings - Circuits, Devices and Systems*, 141(5):407–413, 10 1994.

9 Nicolai Fiege, Martin Kumm, and Peter Zipf. Bit-level optimized constant multiplication using boolean satisfiability. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 71(1):249–261, 2024. doi:10.1109/TCSI.2023.3327814.

10 Rémi Garcia, Léo Pradels, Silviu-Ioan Filip, and Olivier Sentieys. Hardware-Aware Training for Multiplierless Convolutional Neural Networks. In *ARITH 2025 - 32nd IEEE International Symposium on Computer Arithmetic*, pages 1–8, El Paso, United States, May 2025. IEEE.

11 Rémi Garcia and Anastasia Volkova. Toward the multiple constant multiplication at minimal hardware cost. *IEEE Transactions on Circuits and Systems I: Regular Papers*, pages 1–13, 2023. arXiv:hal-03784625v3, doi:10.1109/TCSI.2023.3241859.

12 Rémi Garcia, Anastasia Volkova, and Martin Kumm. Truncated multiple constant multiplication with minimal number of full adders. In *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, Austin, Texas, United States, May 2022. IEEE. doi:10.1109/ISCAS48785.2022.9937350.

13 Rémi Garcia, Anastasia Volkova, Martin Kumm, Alexandre Goldsztejn, and Jonas Kühle. Hardware-aware design of multiplierless second-order iir filters with minimum adders. *IEEE Transactions on Signal Processing*, 70:1673–1686, 2022. doi:10.1109/TSP.2022.3161158.

14 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. A Series of Books in the Mathematical Sciences. W. H. Freeman and Company, New York, NY, USA, January 1979.

15 Sidinei Ghissoni, Eduardo Costa, Cristiano Lazzari, José Monteiro, Levent Aksoy, and Ricardo Reis. Radix-2 decimation in time (dit) fft implementation based on a matrix-multiple constant multiplication approach. In *2010 17th IEEE International Conference on Electronics, Circuits and Systems*, pages 859–862, 2010. doi:10.1109/ICECS.2010.5724648.

16 Oskar Gustafsson. Towards optimal multiple constant multiplication: A hypergraph approach. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems and Computers (ACSSC)*, pages 1805–1809, Pacific Grove, CA, USA, October 2008. IEEE.

17 Tobias Habermann, Jonas Kühle, Martin Kumm, and Anastasia Volkova. Hardware-aware quantization for multiplierless neural network controllers. In *2022 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pages 541–545, 2022. doi:10.1109/APCCAS55924.2022.10090271.

18 Reid M. Hewlitt and Earl S. Swartzlander. Canonical signed digit representation for fir digital filters. In *2000 IEEE Workshop on Signal Processing Systems. SiPS 2000. Design and Implementation*. IEEE, 2000. Cat. No.00TH8528. doi:10.1109/SIPS.2000.886740.

19 Charles D. Howard, Linda S. DeBrunner, and Victor DeBrunner. Hybrid mcm implementation for fir filters in fpga devices. In *2014 IEEE International Conference on Electro/Information Technology (EIT)*, pages 1–6. IEEE, June 2014. doi:10.1109/EIT.2014.6871750.

20 Martin Kumm. *Multiple Constant Multiplication Optimizations for Field Programmable Gate Arrays*. Springer, Wiesbaden, Germany, October 2015. doi:10.1007/978-3-658-10419-7.

21 Martin Kumm. Optimal constant multiplication using integer linear programming. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 65(5):567–571, May 2018. doi:10.1109/TCSII.2017.2772922.

22 Martin Kumm, Anastasia Volkova, and Silviu-Ioan Filip. Design of optimal multiplierless fir filters with minimal number of adders. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(2):658–671, 2023. doi:10.1109/TCAD.2022.3179221.

23 Martin Kumm and Peter Zipf. High speed low complexity fpga-based fir filters using pipelined adder graphs. In *2011 International Conference on Field-Programmable Technology (FPT)*. IEEE, December 2011. doi:10.1109/FPT.2011.6132663.

24 Martin Kumm, Peter Zipf, Mathias Faust, and Chip-Hong Chang. Pipelined adder graph optimization for high speed multiple constant multiplication. In *2012 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 49–52. IEEE, May 2012. doi:10.1109/ISCAS.2012.6271996.

25 Vitaly Lagoon and Amit Metodi. Deriving optimal multiplication-by-constant circuits with a sat-based constraint engine. In *ModRef 2020 – The 19th Workshop on Constraint Modelling and Reformulation*, September 2020. September 2020.

26 Pramod Kumar Meher and Yu Pan. Mcm-based implementation of block fir filters for high-speed and low-power applications. In *Proceedings of the IEEE/IFIP 19th International Conference on Very Large Scale Integration and System-on-Chip (VLSI-SoC)*, pages 118–121. IEEE, October 2011. doi:10.1109/VLSISoC.2011.6081653.

27 Singapore Nanyang Technological University. FIRsuite: Suite of Constant Coefficient FIR Filters. https://www.firsuite.net/FIR/FromPublication, November 2023.

28 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard cp modelling language. In Christian Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007. doi:10.1007/978-3-540-74970-7_38.

29 Xiao Peng and Christine Solnon. BFS-Based Canonical Codes for Generating Graphs with Constraint Programming. In *31st International Conference on Principles and Practice of Constraint Programming (CP 2025)*, volume 340, Glasgow, United Kingdom, August 2025. URL: https://hal.science/hal-05104512, doi:10.4230/LIPIcs.CP.2025.41.

30 Charles Prud'homme and Jean-Guillaume Fages. Choco-solver: A java library for constraint programming. *Journal of Open Source Software*, 7(78):4708, 2022. doi:10.21105/joss.04708.

31 George W. Reitwiesner. Binary arithmetic. In Franz L. Alt, editor, *Advances in Computers*, volume 1, pages 231–308. Academic Press, 1960.

32 James B. Wendt, Nathaniel A. Conos, and Miodrag Potkonjak. Multiple constant multiplication implementations in near-threshold computing systems. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1071–1075. IEEE, April 2015. doi:10.1109/ICASSP.2015.7178134.

33 Neng-Fa Zhou and Håkan Kjellerstrand. The picat-sat compiler. In Marco Gavanelli and John H. Reppy, editors, *Practical Aspects of Declarative Languages (PADL 2016)*, volume 9585 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2016. doi:10.1007/978-3-319-28228-2_4.