New Algorithms for Constant Coefficient Multiplication in Custom Hardware

NEW ALGORITHMS FOR CONSTANT COEFFICIENT MULTIPLICATION IN CUSTOM HARDWARE

BY

JASON THONG, B. Eng.

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2009) McMaster University

(Electrical & Computer Engineering) Hamilton, Ontario, Canada

TITLE:	New Algorithms for Constant Coefficient Multiplication in Custom Hardware
AUTHOR:	Jason Thong B. Eng. (Electrical and Biomedical), McMaster University, Hamilton, Ontario, Canada
SUPERVISOR:	Dr. Nicola Nicolici
NUMBER OF PAGES:	xxv, 237

Abstract

Multiplying by known constants is a common operation in many digital signal processing (DSP) algorithms. High performance DSP systems are implemented in custom hardware, in which the designer has the ability to choose which logic elements will be used to perform the computation. By exploiting the properties of binary multiplication, it is possible to realize constant multiplication with fewer logic resources than required by a generic multiplier. In this thesis, we present several new algorithms for solving the constant multiplication problem - *given* a set of constants, *find* a low-cost logic circuit that realizes multiplication by each of the constants.

In this thesis, a thorough analysis of the existing algorithms, the underlying frameworks, and the associated properties is provided. We also propose new strategies which are fundamentally different from the existing methods, such as the integration of a heuristic algorithm within an optimal algorithm. In our proposed optimal exhaustive algorithms, we introduce aggressive pruning methods to improve the compute efficiency (compared to existing optimal exhaustive algorithms). Our proposed heuristics attempt to address the weaknesses of the existing heuristics. By extending the analysis of prior work and providing new insight, we are often able to improve both the run time and the performance (in terms of minimizing logic resources).

Acknowledgements

I am deeply grateful to all who have contributed to this thesis. Without the support of faculty, friends, and family, my work would not have been nearly as proficient. Many people have helped me mature as a researcher and as a person in some way or another; I would like to acknowledge those who have played a key role in my development.

I am eternally indebted to my parents, who have given me so much love and continually provided an excellent environment for me to learn and work. Because of your support, I have the desire to do good research and to keep improving my skills.

I admire the no-nonsense approach of my supervisor Dr. Nicola Nicolici, who I thank for consistently challenging me to improve in many aspects of research beyond just technical rigor. In addition to Dr. Nicolici, I also thank Dr. Capson and Dr. Szymanski for their insightful comments during my defense. The final form of my thesis and my defense presentation have been greatly assisted by my very professional friends in the Computer-Aided Design and Test group at McMaster: Adam Kinsman, Dr. Henry Ko, Zahra Lak, Phil Kinsman, and Mark Jobes. I sincerely appreciate your enthusiastic support and all of your suggestions. Another thanks goes to Adam, Mark and Dr. Nicolici for enduring several (often unstructured) discussions about my algorithms during their development. Finally, I would like to acknowledge the administrative and technical staff of the ECE department for their assistance.

Glossary

Abstraction Simplification of details, approximation of complex problems

Adder depth The maximum number of serial adder-operations from input to output

Adder distance The minimum number of adder-operations needed to construct a target by using the existing terms

Adder-operation An addition or subtraction in which the operands may be shifted

Additive decomposition A SCM or MCM decomposition where several terms can be added together in any order

ASIC Application-specific integrated circuit

BBB Best of Bernstein or BHM, an existing heuristic SCM algorithm

BDFS Bounded depth-first search, a proposed optimal MCM algorithm

BFS$_{\text{mcm}}$ Breadth-first search MCM, an existing optimal MCM algorithm

BH Bull Horrocks, an existing heuristic MCM algorithm

BHM Bull Horrocks modified, an existing heuristic MCM algorithm

BIGE Bounded inverse graph enumeration, a proposed optimal SCM algorithm

CAD Computer-aided design, tools for design automation

Complexity-n constants The set of constants in which the optimal SCM cost is n

CSD Canonical signed-digit, the SD form with no adjacent nonzero digits and the minimum number of nonzero digits

CSE Common subexpression elimination, a framework for solving SCM and MCM

Custom hardware Computing hardware that is customized for a given application, includes ASICs and FPGAs

DAG Directed acyclic graph, a framework for solving SCM and MCM

DiffAG Difference-based adder graph, an existing heuristic MCM algorithm

Digit clashing The CSE problem of disappearing patterns due to colliding digits

Division test A pruning method for exploiting multiplicative decompositions

DSP Digital signal processing

FPGA Field-programmable gate array

H(k) Heuristic with k extra nonzero digits, an existing heuristic SCM algorithm

H(k)+ODP The H(k) algorithm with ODPs, a proposed heuristic SCM algorithm

H3 A variant of Hcub with exact distance 3 tests, a proposed heuristic MCM algorithm

H3$_d$ A depth-constrained version of H3, a proposed heuristic MCM algorithm

H4 A variant of Hcub with exact distance 4 tests, a proposed heuristic MCM algorithm

Hcub Heuristic of cumulative benefit, an existing MCM algorithm

Heuristic An effective but potentially suboptimal method for solving a problem

IGT Inverse graph traversal, a DAG-based pruning method

Logic resources An abstraction of the amount of silicon required to implement a logic function

MAG Minimized adder graph, an existing optimal SCM algorithm

MCM Multiple constant multiplication, find a low-cost add-shift-subtract realization of multiplication by each of the given constants

MSD Minimal signed-digit, any SD representation with the minimum number of nonzero digits

Multiplicative decomposition A SCM or MCM decomposition involving multiplication by complexity-n constants

ODP Overlapping digit pattern, a proposed technique for partially resolving the CSE digit clashing problem

Pattern (CSE) A collection of signed digits that define how existing terms are added-operated together

Pruning The removal of parts of a search tree that are no longer of interest

Pseudo-ODP An ODP which provides no benefit now but may provide more benefit on a future iteration

RAG-*n* n-dimensional reduced adder graph, an existing heuristic MCM algorithm

Ready set The set of existing terms constructed so far

Redundancy between constants The reuse of terms between different constants when constant multiplication is decomposed

Redundancy within constants The reuse of terms within each constant when constant multiplication is decomposed

Ripple-carry adder A multi-bit adder composed of several single-bit adders connected serially

SBAC Single-bit adder cost, a proposed optimal SCM algorithm

SCM Single constant multiplication, same as MCM but for a single constant

SD representation Signed-digit representation, used in the CSE framework

Search space The set of all solutions that can be found by an algorithm, this is smaller than the solution space for heuristic algorithms

Solution space The set of all feasible solutions

Successor set The set of terms that can be constructed with one adder-operation

Target set The given set of constant(s) for which we must realize multiplication by

Vertex reduction A pruning method for exploiting additive decompositions

Notation

Notation	Meaning	Def. in Section
R	Ready set, the set of existing terms constructed so far, this contains the solution when the algorithm ends	2.3.1
S	Successor set, the set of terms that can be constructed with one adder-operation (with respect to R)	3.2.2
S_2	2^{nd} order successor set, the set of terms that can be constructed with two adder-operations	3.2.6
T	Target set, the given set of constant(s) for which we must realize multiplication by	2.3.1
T'	Remaining target set, targets not yet constructed	4.2
C_n	Complexity-n constants, the set of constants in which the optimal SCM cost is n	3.2.3
$\text{dist}(R,t)$	Adder distance from R to a target t	3.2.3
$\mathcal{A}(x,y)$	Adder-operation between elements x and y	2.3.2
$\mathcal{A}(x_1,\ldots,x_n)$	Vertex-reduced adder-operation between all of the elements $x_1,\ldots,x_n$	3.2.5
$\lvert U\rvert$	The cardinality of the set U	2.3.2
$\lvert u\rvert$	The absolute value of the element u	2.3.2
$U \cdot V$	Set of all products	3.2.4
U/V	Set of all divisions with zero remainder	3.2.4
$\overline{1}$	In signed-digit notation, this represents -1	2.1.2
$\overline{A}$	In CSE, this represents the negative of pattern A	3.1.2
$B = A0\overline{A}$	An example of CSE notation, pattern B is defined in terms of two instances of the existing pattern A	3.1.2

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Multiplying a variable by a set of known constant coefficients is a common operation in many digital signal processing (DSP) algorithms. Compared to other common operations in DSP algorithms, such as addition, subtraction, using delay elements, etc., multiplication is generally the most expensive. There is a trade-off between the amount of logic resources used (i.e. the amount of silicon in the integrated circuit) and how fast the computation can be done. Compared to most of the other operations, multiplication requires more time given the same amount of logic resources and it requires more logic resources under the constraint that each operation must be completed within the same amount of time.

A general multiplier is needed if one performs multiplication between two arbitrary variables. However, when multiplying by a *known constant*, we can exploit the properties of binary multiplication in order to obtain a less expensive logic circuit that is functionally equivalent to simply asserting the constant on one input of a general multiplier. In many cases, using a cheaper implementation for only multiplication still results in significant savings when considering the entire logic circuit because

multiplication is relatively expensive. Furthermore, multiplication could be the dominant operation, depending on the application.

In this thesis, we will propose several algorithms which *run in software*, but the solutions that these algorithms produce enable one to efficiently implement constant coefficient multiplication *in hardware*. *Given* the set of constant coefficients, said algorithms *search* for good hardware realizations.

The remainder of the introduction is organized as follows. In section 1.1, we will present some DSP algorithms and applications that require multiplication by a set of constants. In section 1.2, the advantages of using a ''multiplierless'' implementation of constant coefficient multiplication in custom hardware are discussed. Finally, the contributions and the organization of the thesis are summarized in section 1.3.

1.1 Applications of Constant Multiplication

In this section, we will provide a few examples of applications which require multiplication by a set of constants. When *designing the hardware* to implement these applications, the algorithms in this thesis can be used to provide good solutions for the constant coefficient multiplication part of the logic circuit.

Multiplication by a set of constants occurs when multiplying by a constant vector or a constant matrix. For example, the dot product $a \cdot b$ gives the scalar projection of a onto b (or vice versa). Multiplication by a constant matrix is nothing more than performing the dot product between several constant vectors (which collectively form a matrix) and a variable vector (the elements of this vector are the inputs). Multiplying by a constant matrix can thus be regarded as a linear transformation of coordinates, which is used in many applications. For example, the conversion from the RGB (red,

green, blue) color space to the YUV color space (Y represents brightness, U and V represent chroma) involves multiplication by a constant 3x3 matrix. Because the human eye is more sensitive to brightness than coloring (chroma), we can compress the information in the U and V components with only a minor perceived distortion. This is exploited in JPEG and MPEG for compressing images and video, respectively. However, most display devices require color to be provided in the RGB format, hence the need for color space conversion. Any application which involves a predefined linear transformation of coordinates will use multiplication by a set of constants.

Many discrete linear signal transforms involve multiplication by constants, such as the Discrete Fourier Transform (DFT) and other Fourier-related transforms like the Discrete Cosine Transform. As commonly known, the DFT and Fourier-related transforms can be used for spectral analysis, i.e. to identify what proportion of the input signal was contributed by a sinusoid at frequency f (for different f). These transforms can also be regarded as a linear transformation of coordinates and as such, they can be used for signal compression. The objective is to transform the input signal into a coordinate basis such that the majority of the energy (or information) in the signal is concentrated into only a few components. We can then compress or even completely disregard the low energy components with only a minimal distortion. In communications, signal compression is key to minimizing the amount of energy needed to transmit information. This is particularly important in mobile devices, in which battery life is a major concern.

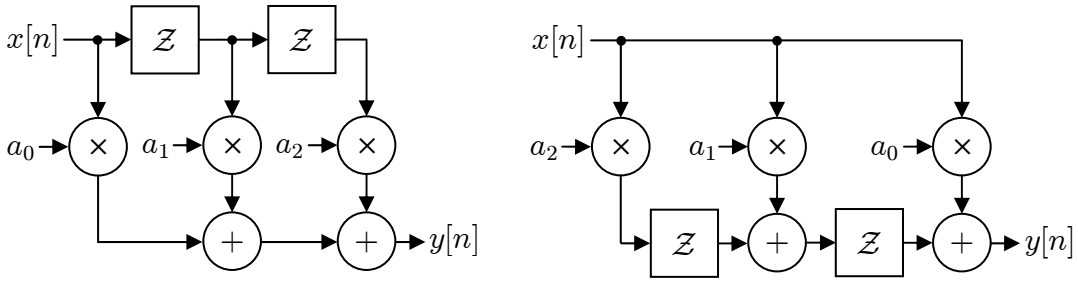
As an example of signal compression, many real-life images (as opposed to randomly generated pixels) are typically composed of mostly low frequency content. By expressing an image in its frequency domain, we can compress the high frequency

information without much distortion. This is also exploited in JPEG and MPEG. Likewise, the human ear is less sensitive to high frequency sounds. MP3 exploits this by using less compression for low frequencies and more compression for high frequencies. Nonetheless, we first need to represent the signal in the frequency domain before we can compress. When decompressing (to listen to the audio), after we undo the compression, we need to transform from the frequency domain to the time domain, which again involves a linear signal transform with constant multiplication.

Many linear signal transforms are computed in a manner such that intermediate terms are reused, such as done in the Fast Fourier Transform (FFT). For an input signal of length n, this reduces the number of multiplications from $O(n^2)$ to $O(n \log n)$. Even in this form, this still involves multiplication by constants. For example, each ''butterfly'' in the FFT involves a multiplication by a constant 2x2 matrix (assuming complex numbers are represented in rectangular coordinates, since addition is complicated in polar coordinates).

The DFT can be used for solving partial differential equations numerically (to estimate the solution). By approximating the Fourier Transform of the original equations with the DFT, the equations are expressed as a sum of complex exponentials. Differentiation now becomes multiplication ($\frac{d}{dx}e^{ix} = xe^{ix}$), which is much easier to solve. The product of polynomials involves the convolution of the polynomial coefficients. Assuming the polynomials are n^{th} order, $O(n^2)$ multiplications are needed, yet by using a FFT-based method, this can be reduced to $O(n \log n)$. In both examples, an inverse Fourier Transform is also computed at the end.

Finite impulse response (FIR) filters and infinite impulse response (IIR) filters are commonly used in DSP algorithms. Assume N and M are finite, $x[n]$ is the input,



(a) Three separate single constant multiplications.

(b) Three constants that multiply the same multiplicand.

Figure 1.1: Two realizations of a FIR filter with $N = 3$ ($\mathcal{Z}$ denotes a delay element).

$y[n]$ is the output, and a_i and b_i are constant coefficients, then every IIR filter has the form $y[n] = \sum_{i=0}^{N-1} a_i \cdot x[n-i] - \sum_{i=1}^{M} b_i \cdot y[n-i]$. In a FIR filter, all of the $b_i = 0$. Two realizations of a FIR example with $N = 3$ are shown in Figure 1.1. Three separate single constant multiplications are needed in Figure 1.1(a) whereas one instance where the same multiplicand is multiplied by three constants is used in Figure 1.1(b). The delay elements $\mathcal{Z}$ have a larger bit width in Figure 1.1(b), but when decomposing multiple constant multiplication, intermediate terms can be shared *between* constants to reduce logic resources, as illustrated in section 2.2.

FIR and IIR filters are arguably the simplest means of performing digital filtering (and therefore likely the cheapest to implement). Filtering is used to attenuate or gain each frequency in the input signal by a desired amount, thus it can be used to alter the frequency response characteristics in a system. For example, a communication channel may attenuate some frequencies more than others, but by having an estimate of the channel's frequency response, we can apply a filter with the inverse attenuation of each frequency at the output of the channel in order to restore the original signal. In communications, FIR/IIR filters are used for equalization, echo suppression, etc.

FIR filters are typically used for upsampling and downsampling digital signals. In both cases, an interpolation filter is typically used to smooth the newly sampled signal. This has many applications in audio and video processing (for example, resizing a video or playing audio at different speeds requires a change in the sampling rate). FIR filters are also used in image processing. For example, the Laplacian operator can be used for edge detection, and one of the simplest means of implementing this is with a 2-dimensional FIR filter.

The work in this thesis is primarily intended for DSP systems, where the benefits of finding low-cost implementations of constant multiplication are considerable since it is frequently used. Even so, multiplication by universal constants can arise in scientific computing, for example. However, when solving a set of equations, it may be possible to collect the constants in order to reduce the total number of multiplications.

1.2 Custom Hardware

1.2.1 A Comparison of Custom Hardware and Instruction-Based Processors

In order to achieve higher computational throughput and/or lower energy consumption, many DSP algorithms are implemented in custom hardware. Examples of custom hardware include application specific integrated circuits (ASICs) and field-programmable gate arrays (FPGAs). These benefits are further enhanced by finding lower cost implementations of constant coefficient multiplication, as smaller and cheaper implementations of DSP systems are obtained. In ASICs, a smaller logic circuit results in less material costs and lower energy consumption *ceteris paribus*.

Alternatively, for the same amount of silicon, a higher computational throughput can be obtained by placing more computational units in parallel. FPGAs typically have some dedicated multipliers, but a large DSP design could require more multiplications than available multipliers, in which case one must build *soft* multipliers (i.e. using the programmable logic elements). In FPGAs, finding an implementation that requires less logic elements may enable one to fit the system on a smaller and likely cheaper FPGA, among various other benefits. The benefits manifest when computational units are used in parallel. For example, the total computational throughput may need to be high enough to satisfy a real-time constraint (such as a video decoder which must output a frame of video of size x by y pixels every z seconds).

Many DSP applications only require the computational unit to perform one task. For example, a video decoder inside a television receives encoded video in a pre-defined format and outputs decoded video in another pre-defined format. In this case, the flexibility to perform different tasks, which is offered by instruction-based processors, is not needed. In custom hardware, no time and no parts of the logic circuit are wasted for fetching and processing instructions if this is not needed, thus this extra time and circuit area can be used to perform useful computations.

Most of the instruction-based processors in the DSP systems of today have multipliers since multiplication is a common task. Furthermore, in most compute-intensive DSP systems, the multipliers are pipelined to have high throughput. It will be shown in section 2.1 that constant coefficient multiplication is decomposed into a set of addition, subtractions, and binary shifts. In general, fewer multiply operations are required than add, subtract and shift operations in order to compute multiplication by a set of constants. Thus, if multipliers are already available in an instruction-based

processor, they may as well be used to minimize the execution time. However the size of the add-subtract-shift logic circuit which implements constant multiplication is generally smaller than if multipliers were used. Unlike instruction-based processors which already have multipliers available to use, in custom hardware, the designer *has the ability to choose what hardware will be used* to perform a desired computation. Conclusively, the application of constant multiplication decomposition is *primarily intended for custom hardware.*

On a different note, some small instruction-based processors (which could be used in embedded systems for instance) may not have multipliers. An example of such is the NIOS II/e processor from Altera [3]. In this case, multiplication has to be done with additions, subtractions and shifts anyways, thus minimizing the number of operations results in less execution time. Even so, this particular application is not our main focus. Such processors are typically not used for large amounts of computation, thus multiplication within this processor is unlikely to be on the critical path of a system-wide task.

1.2.2 Logic Resources

The objective of the work in this thesis is to design algorithms to efficiently search for good decompositions of constant coefficient multiplication -- *given* the set of constants, *find* the set of additions, subtractions, and shifts that realize the constant coefficient multiplication with the minimum amount of logic resources.

In custom hardware, shifts incur no cost since they are hardwired. Floating point numbers have a finite size mantissa, so without loss of generality, when we search for constant multiplication decompositions, we can assume that all of the given constant

coefficients will be integers because any fractional number with finite precision can be left shifted until it becomes an integer. This process incurs no cost and is reversible, i.e. after constant multiplication has been implemented, we can shift the result to correct for any factor of 2^n, where n is an integer.

We use the number of additions or subtractions as the metric for the amount of logic resources. Our targeted implementation is custom hardware, in which addition and subtraction require approximately the same amount of logic resources and shifts incur no cost since they are hardwired. It is assumed that ripple-carry adders are used since the objective is to minimize the amount of logic. Minimization for other types of adders, such as carry-save adders, is beyond the scope of this thesis. Although the number of additions or subtractions is an abstraction of the amount of silicon required to implement the logic circuit, it is conjectured that finding good solutions with this metric typically results in good solutions in terms of minimizing the amount of silicon. This metric is the most commonly used metric in this area of research.

Some abstraction is needed in order to solve problem sizes of the most practical importance within a reasonable amount of time. This claim is supported by our experimental results. Instead of the number of additions or subtraction, the number of single bit adders or subtractors can be used as a more accurate metric (note even this still has some abstraction from the amount of silicon). As shown in our experimental results in section 5.3.3, a significant amount of extra time is required to solve the same problems using this more accurate metric. This translates into needing impractical amounts of time to solve larger but still real-sized problems. This also implies that further increasing the metric accuracy will result in longer run times, meaning only smaller problem sizes (which are less relevant in practice) can be solved

within reasonable amounts of time. Furthermore, using the amount of silicon as the metric makes the solution dependent on the implementation fabric of the logic circuit, thus the solution would be non-portable and also dependent on the performance of many other computer-aided design (CAD) tools, which perform logic synthesis, place and route, etc.

A ''reasonable'' amount of time is difficult to quantify because it depends on the design flow of the system. For example, if the system is intended to satisfy an existing standard, the constants will be defined and thus each constant multiplication problem only needs to be solved *once* (even if other parts of the system are modified). In this case, one may be willing to wait hours or days for each problem instance. Conversely, a faster algorithm is needed if the design specifications are not finalized (for example, the constants may need to be updated as other parts of the system are modified). Depending on how finalized the design is, one may only be willing to wait a few seconds for each constant multiplication problem, for example. If a large part of the system involves constant multiplication, it is sensible to allocate a large portion of the time in the total CAD flow to solving constant multiplication. Conclusively, a ''reasonable'' amount of time is highly application specific, as discussed above.

1.3 Thesis Contributions and Organization

In chapter 2, we will illustrate how constant multiplication can be decomposed, which facilitates the search for low-cost implementations. We will also introduce the formal problem definition. In chapter 3, we will present the algorithmic frameworks and several of the associated properties that are exploited by many algorithms. By discussing this *before* the existing algorithms in chapter 4, we are able to provide a

unified discussion of the existing algorithms, which helps to illustrate the similarities and differences in the *underlying intuition* of these algorithms.

Our proposed single constant multiplication (SCM) and multiple constant multiplication (MCM) algorithms are described in detail in chapters 5 and 6, respectively. The best methods for solving SCM are fundamentally different from the best techniques for solving MCM, hence we have considered these problems separately. We propose new heuristics and optimal (exhaustive) algorithms. By extending the analysis of prior work and providing new insight, we are often able to improve the run time and the performance (in terms of minimizing logic resources).

We also address two additional problems. In section 5.3, we show that considering a more accurate (less abstracted) resource metric results in solutions with less logic resources. In section 6.2, we propose a depth-constrained MCM algorithm. By reducing the depth, the computational throughput of the logic circuit increases. Finally, the contributions and ideas for future work are summarized in chapter 7.

Chapter 2

Problem Background

We will begin by illustrating how multiplication by a set of constants can be decomposed into a set of additions, subtractions, and binary shifts. In section 2.2, we will demonstrate that reusing intermediate terms in the decomposition can further reduce the size of the logic circuit. Many examples are provided to illustrate these concepts. In section 2.3, several formal problem definitions are provided. Some simplifying assumptions will be discussed and then incorporated into the problem definitions.

2.1 Integer Multiplication

2.1.1 General Multiplication and Constant Multiplication

As stated in section 1.2.2, since shifts are free, we can assume that all of the constant coefficients will be integers, as floating point numbers can be shifted until they become an integer (this requires no cost and this process is reversible). Therefore we will only consider integer multiplication.

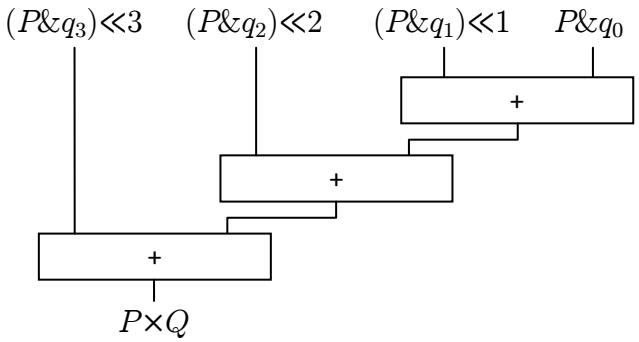


Figure 2.1: A general 4-bit multiplier. A left shift by n bits is denoted by $\ll n$. Q has a binary representation of $q_3q_2q_1q_0$, thus $q_i \in \{0, 1\}$ for $i = 0, 1, 2, 3$. Note that $P\&q_i$ is a short-form representation for using an AND gate between q_i and each bit of P, so if P is m bits, then $P\&q_i$ is also m bits.

In any radix, multiplication can be decomposed into several smaller multiplications and additions. Let us illustrate with an example. In radix 10, to compute 456×789, we can decompose this into $(456 \times 7) \times 10^2 + (456 \times 8) \times 10^1 + (456 \times 9) \times 10^0$, which is equivalent to $((456 \times 7) \ll 2) + ((456 \times 8) \ll 1) + (456 \times 6)$, where $\ll n$ denotes a left shift by n digits. The multiplicand (456 in this example) is multiplied by each digit of 789, is left shifted by the appropriate amount, and then these shifted products are added. In radix 2 (binary), each digit is 0 or 1, so the multiplicand is only ever multiplied by 0 or 1 (but multiplication by zero is zero and multiplication by one is the original number). Therefore in radix 2, multiplication can be decomposed into a set of additions and left shifts (multiplication is *not* needed).

Let us multiply two arbitrary binary integers P and Q. Assume Q can be represented on N bits, thus Q has a binary representation of $q_{N-1}q_{N-2}q_{N-3}\cdots q_2q_1q_0$, where each bit $q_i \in \{0, 1\}$. Let $R = P \times Q$, then $R = \sum_{i=0}^{N-1}(P \times q_i) \ll i$. Obviously $P \times q_i = 0$ if $q_i = 0$ and $P \times q_i = P$ if $q_i = 1$. To implement this, an AND gate is used between q_i and each bit of P. A general N-bit multiplier must have $N - 1$ adders

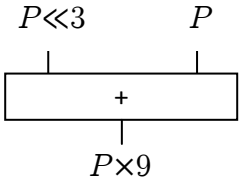


Figure 2.2: Multiplication by 9 (1001 in binary) requires fewer adders than a general 4-bit multiplier. Also note the AND gates are no longer needed.

since it is possible that every q_i could be 1, in which case N instances of the $P \ll i$ must be added. A general 4-bit multiplier is shown in Figure 2.1.

Let $S = \sum_{i=0}^{N-1} q_i$, which is the number of bits in Q that are 1. Clearly, $S \leq N$. If Q is a constant, then multiplication by Q requires only $S - 1$ adders (to add the S instances of $P \ll i$). In other words, if we know that a particular bit q_i is 0, we no longer need the adder that would have been needed in a general multiplier to add $P \ll i$ in the sum to compute $P \times Q$. In a general multiplier, Q can take any arbitrary value, so $N - 1$ adders are needed for the worst case. Conversely, a constant coefficient multiplier needs *up to* $N - 1$ adders. Even so, a constant multiplier always requires less logic resources than a general multiplier since the AND gates to multiply $P \times q_i$ are no longer needed. An example of a constant multiplication by 9 (1001 in binary) is shown in Figure 2.2.

2.1.2 Using Subtraction in Constant Multiplication

A smaller logic may be obtained by considering subtraction. As an example, consider multiplying an arbitrary number x by 7 (111 in binary). From section 2.1.1, it follows that $7x = (x \ll 2) + (x \ll 1) + x$. Now consider decomposing $7x$ as $8x - x = (x \ll 3) - x$. The latter decomposition is better since addition and subtraction require roughly the

same amount of logic resources in custom hardware. To simplify the discussion on logic resources, both adders and subtractors are simply referred to as ''adders'' for the remainder of this thesis unless stated otherwise.

Recall from section 2.1.1 the definition of $R = P \times Q = \sum_{i=0}^{N-1}(P \times q_i) \ll i$. In a binary representation, $q_i \in \{0, 1\}$ is inherently enforced, however if we let $q_i = -1$, subtraction will be used since $P \times q_i$ would then equal $-P$. Adding a negative term in a summation is equivalent to using subtraction.

In the remainder of this thesis, we will use $\overline{1}$ to represent -1. Allowing $q_i \in \{\overline{1}, 0, 1\}$ facilitates the use of both addition and subtraction. If a constant Q can be represented by n *nonzero digits* (where each digit $q_i \in \{\overline{1}, 0, 1\}$), then multiplication by Q requires $n - 1$ adders (to add the n terms that the n nonzero digits represent).

By also allowing subtraction, we may be able to reduce the number of nonzero digits in Q. A run of consecutive ones in the binary representation of the constant requires many additions, but only one subtraction. Notice that $\sum_{i=0}^{N-1} 2^i = 2^N - 1$. For example, revisiting the example at the beginning of this section, 111 can be replaced by $100\overline{1}$. Likewise, $1111 \rightarrow 1000\overline{1}$, $11111 \rightarrow 10000\overline{1}$, and so on. Obviously $1 \rightarrow 1\overline{1}$ is a valid transform, but it is of no benefit since it increases the number of nonzero digits. The smallest transform we will consider is $11 \rightarrow 10\overline{1}$.

The above transform can be applied repeatedly to a constant until there are no more consecutive nonzero digits. The resulting signed-digit representation is known as the canonic signed digit (CSD) form. An example with $Q = 7523$ is shown in Figure 2.3. Notice the transform proceeds from right to left. On each step, the rightmost group of consecutive ones is eliminated. Each newly created $\overline{1}$ always has no adjacent nonzero digits (we do not consider $1 \rightarrow 1\overline{1}$ and it is impossible to create $\overline{11}$ or $\overline{1}1$

$$
\begin{array}{r}
1110101100011 \\
111010110010\overline{1} \\
11101101\overline{1}0010\overline{1} \\
1110\overline{1}0\overline{1}0010\overline{1} \\
100\overline{1}0\overline{1}0\overline{1}0010\overline{1}
\end{array}
$$

Figure 2.3: The canonical signed digit (CSD) transform applied to the constant $Q = 7523$. The transform $011\cdots111 \rightarrow 100\cdots00\overline{1}$ is repeatedly applied from right to left until no more consecutive nonzero digits remain.

with the $011\cdots111 \rightarrow 100\cdots00\overline{1}$ transform). The next group of consecutive ones to transform is always to the left of the current group, so only one pass through the constant is needed. Thus the CSD transform requires linear run time with respect to the bit width of the constant). The transform happens in a deterministic manner, so a unique CSD representation exists for each number. An algorithm for computing the CSD transform is shown in detail in [4].

The CSD representation has the minimum number of nonzero digits among any signed digit representation [5]. If the binary form of a constant can be represented on b bits, the CSD form can be represented on at most $b+1$ CSD digits ($\overline{1}$, 0, and 1). If we transform the leftmost group of consecutive ones with $011\cdots111 \rightarrow 100\cdots00\overline{1}$, the bit width enlarges by one, but now the leftmost 1 is isolated, thus no more transforms can be done. This ‘‘carry-out’’ behavior is illustrated in the last step of the CSD transform in Figure 2.3. Since the CSD form has no adjacent nonzero digits, it may have up to $\lceil \frac{b+1}{2} \rceil$ nonzero digits. Therefore realizing multiplication by a constant of binary bit width b requires $\lceil \frac{b+1}{2} \rceil - 1 = \lceil \frac{b-1}{2} \rceil$ adders in the worst case. This is much better than the $b-1$ bound from using addition only. In [4], it was proven that the average number of nonzero digits in the CSD form is $\frac{b}{3} + \phi$, where ϕ is approximately 0.5 (refer to [4] for more details).

The CSD transform as described above cannot be directly applied to negative constants in 2's complement. However, finding the negative of a number in the CSD form is trivial, as inverting the sign of each digit inverts the sign of the number that these digits represent (we can do this since both 1 and $\overline{1}$ exist). Thus, given a negative constant, first take the absolute value (2's complement to unsigned binary), then do the CSD transform, then invert the sign of all CSD digits. Since Q and $-Q$ have the same number of nonzero digits, the average and worst case number of nonzero digits remain the same. However, sometimes we will need an extra adder. For example, $\overline{1}0\overline{1}0\overline{1}$ requires 3 adders, not 2.

2.2 Sharing Intermediate Terms

To the best of our knowledge, the CSD bound on the average and worst case number of adders is the best analytical upper bound. However, it is easy to show that the CSD implementation of constant multiplication is not always optimal in terms of using the minimum number of adders. Consider multiplication by 45, which has CSD representation of $10\overline{1}0\overline{1}01$. Since it has 4 nonzero digits, 4 instances of the multiplicand must be added (or subtracted), thus the cost is 3 adders. However, $45x$ can be decomposed as $45x = (16-1)(3x) = ((3x) \ll 4) - (3x)$, where $3x = 4x - x = (x \ll 2) - x$. This decomposition is better because it only needs 2 adders.

It is possible to use less adders than CSD by reusing intermediate terms. The CSD implementation of $45x$ is $(x \ll 6) - (x \ll 4) - (x \ll 2) + x$. This can be factored into $(((x \ll 2) - x) \ll 4) - ((x \ll 2) - x)$. Once the $(x \ll 2) - x$ is computed, there is no point using 2 more adders to add the $x \ll 6$ and $x \ll 4$ terms because adding a shifted version of $(x \ll 2) - x$ achieves the same effect with only 1 more adder.

If we need to multiply the multiplicand by several constant coefficients, sharing intermediate terms *between* different constants can save adders. For example, consider multiplying an arbitrary number x by both 13 and 25. It can be shown that multiplication by 13 alone requires at least 2 adders and multiplication by 25 alone also requires at least 2 adders. One optimal solution for 13 is $13x = (x \ll 3) + (x \ll 2) + x$, and one optimal solution for 25 is $25x = (x \ll 4) + (x \ll 3) + x$. Notice that both use the intermediate term $9x = (x \ll 3) + x$. By sharing $9x$ (which costs one adder to make), we can implement both $13x = (9x) + (x \ll 2)$ and $25x = (9x) + (x \ll 4)$ with only 2 more adders, thus a total of 3 adders are needed.

Clearly, reusing the appropriate intermediate terms can save adders. However, the challenge is to *find* which intermediate terms are ''useful'' *given* the set of constant coefficients (this ''usefulness'' is quantified in different ways by different algorithms in the later chapters). Small problem instances can be solved by hand, but for practical reasons it is necessary to use a search algorithm for large but still real-sized problems.

2.3 Formal Problem Definitions

2.3.1 An Informal Introduction

Throughout the thesis, we will reuse some of the notation introduced in [6] since it provided a unifying framework for constant multiplication problems. We will first introduce two sets used in [6] that are applicable to *any* constant multiplication algorithm and then we will formally define the constant multiplication problem.

Definition 1. *The ''ready'' set R contains all of the terms that have been constructed in order to realize the constant multiplication.*

Definition 2. *The ''target'' set T is comprised of all of the given constant coefficients in a constant multiplication problem, however only unique and nonzero constants are considered (duplicates are ignored).*

Each element in R represents the realization of multiplying an arbitrary multiplicand by this element (i.e. if $R = \{1, 5, 37\}$, for any arbitrary x, we have implemented $1x$, $5x$, and $37x$). Every algorithm starts with $R = \{1\}$ since the multiplicand multiplied by 1 comes for free. Elements are added to R until $T \subseteq R$, at which point the algorithm stops since all of the constant multiplications have been realized. How R gets constructed is decided by the algorithm. A cost of one adder is incurred every time one element is added to R. When the algorithm finishes, the number of adders required to realize the constant coefficient multiplication is the cardinality of $R \setminus \{1\}$ (R with the element 1 removed from it), thus the objective is to minimize the cardinality of R. The rules governing how R can be constructed are described in Definition 3. Note that Definition 3 reuses most of the formal problem definition from [6], but without explicitly stating how the adder-operation $\mathcal{A}(x, y)$ is computed.

2.3.2 Definition of the Constant Multiplication Problem

In this thesis, $|R|$ denotes the *cardinality* of the set R whereas $|r|$ denotes the *absolute value* of the element r.

Definition 3. *The constant multiplication problem:* given *a set of unique and nonzero constant coefficients T,* find *a set $R = \{r_0, r_1, \ldots, r_n\}$ such that $T \subseteq R$, $r_0 = 1$, and for each $r_k \in R$ with $1 \leq k \leq n$, there exists elements $r_i, r_j \in R$ with $0 \leq i, j < k$ that satisfy $r_k \in \mathcal{A}(r_i, r_j)$. The* objective *is to minimize $|R|$.*

Definition 4. *The most general definition of $\mathcal{A}(x, y)$, an adder-operation of the elements x and y, is the set of all possible numbers that can be created using x, y, and one ''adder''. Subtraction and/or shifts may or may not be allowed by the ''adder''.*

We deliberately did not explicitly define the adder-operation $\mathcal{A}(x, y)$ because slightly different (but somewhat related) problems can be derived simply by allowing or disallowing subtraction and/or shifts (and without modifying Definition 3).

2.3.3 Simplifying Assumptions and the Derivation of the Adder-Operation for Custom Hardware

As explained in the introduction, in custom hardware, shifts have no cost and addition and subtraction require approximately the same amount of logic resources. Shifting is equivalent to multiplying by an integer power of 2, thus the most general form of the adder-operation *for custom hardware* is given in (2.1).

$$\mathcal{A}(x, y) = \{z \mid z = 2^m x \pm 2^n y, \text{integer } m \text{ and } n\} \tag{2.1}$$

As explained in section 1.2.2, we can assume all targets are integers. Because shifts are free, factors of 2^n (for integer n) can be corrected anywhere this is needed. Thus without any loss of generality, we can enforce *all* terms in R to be odd integers. To facilitate this, we modify the definition of the adder-operation to (2.2). Algorithms can use this property to remove redundancy within their search space.

$$\mathcal{A}(x, y) = \{z \mid z = 2^m x \pm 2^n y, m \in \mathbb{Z}, n \in \mathbb{Z}, z \in \mathbb{Z}, \frac{z}{2} \notin \mathbb{Z}\} \tag{2.2}$$

In many of the applications of constant multiplication, after the multiplications are done, these products are added. Thus for negative constants, we could multiply by

the absolute value but then use subtraction in the summation of products. In many cases, the sign of the constants can be adjusted elsewhere. To simplify the problem, we will only consider positive constant coefficients. Combining this with the free-shift property above, we can enforce all terms to be *odd and positive integers*. Most of the work in this area of research also uses this assumption.

Under the constraint that all targets are positive, if R is a valid solution to the constant multiplication problem in Definition 3, then so is R', where the elements in R' are the absolute values of the elements in R. In other words, subtraction can be propagated through the logic circuit of a constant coefficient multiplier.

Using (2.2) as the definition of the adder-operation, we will show that if $z > 0$ and $-z \in \mathcal{A}(x, y)$, then $z \in \mathcal{A}(|x|, |y|) \cup \mathcal{A}(|y|, |x|)$. Stating that $-z \in \mathcal{A}(x, y)$ is equivalent to stating that there exists integers m and n such that $-z = 2^m x \pm 2^n y$. Let us examine all combinations of positive or negative x and y with the use of addition or subtraction. Without loss of generality, assume $x, y, z > 0$, then for some integers m and n:

$$-z \neq 2^m x + 2^n y \text{ and } -z \neq 2^n x - 2^m(-y) \text{ because } 2^m x + 2^n y > 0$$

$$-z = 2^m x - 2^n y \text{ or } -z = 2^m x + 2^n(-y) \implies z = 2^n y - 2^m x \in \mathcal{A}(y, x)$$

$$-z = 2^m(-x) + 2^n y \text{ or } -z = 2^m(-x) - 2^n(-y) \implies z = 2^m x - 2^n y \in \mathcal{A}(x, y)$$

$$-z = 2^m(-x) - 2^n y \text{ or } -z = 2^m(-x) + 2^n(-y) \implies z = 2^m x + 2^n y \in \mathcal{A}(x, y)$$

If the first k elements of R are positive, taking the absolute value of the $k + 1^{th}$ element in R does not affect the validity of the solution R (because if $z > 0$ and $-z \in \mathcal{A}(x, y)$, then $z \in \mathcal{A}(|x|, |y|) \cup \mathcal{A}(|y|, |x|)$). Notice that the constraints in Definition 3 are still satisfied if we apply the substitutions $x = r_i$, $y = r_j$, and $z = r_k$ (also, the interchanging of r_i and r_j is allowed). Because $r_0 = 1 > 0$, by induction,

each element of R can be made positive. The only reason an element in R would need to be negative is if an element in T were negative (to satisfy $T \subseteq R$), however we have constrained all targets to be positive. Since we can enforce *all* terms in R to be *odd and positive integers*, we can again modify the definition of the adder-operation to (2.3) to encompass this.

$$\mathcal{A}(x,y) = \{z \mid z = |2^m x \pm 2^n y|, m \in \mathbb{Z}, n \in \mathbb{Z}, z \in \mathbb{Z}, \frac{z}{2} \notin \mathbb{Z}\} \tag{2.3}$$

Notice that (2.3) is symmetric (i.e. $\mathcal{A}(x,y)$ and $\mathcal{A}(y,x)$ produce identical sets) whereas (2.2) is not. When addition is used, clearly $\mathcal{A}(x,y)$ and $\mathcal{A}(y,x)$ produce identical sets. If subtraction is used, $|2^m x - 2^n y|$ can be rewritten as $|-(2^n y - 2^m x)|$, which is simply $|2^n y - 2^m x|$. Algorithms can take advantage of this by only performing subtraction in one direction. Also, the cardinality of $\mathcal{A}(x,y)$ will be smaller if only positive integers are allowed. Conclusively, search algorithms will require less computational effort if (2.3) is used as the definition of the adder-operation.

By induction, we can expect x and y to be odd and positive integers. To ensure $z \in \mathcal{A}(x,y)$ is an odd and positive integer, there are 3 cases of m and n to consider: $m > 0$ *and* $n = 0$, $m = 0$ *and* $n > 0$, and $m = n < 0$. The first two cases correspond to adding an odd and even integer. In the third case, adding two odd integers produces an even integer, but this can be divided by 2 until it becomes odd. Only one shift is needed per adder-operation, since the $m = n < 0$ case is computed as $z = 2^n|x \pm y|$.

For the convenience of notation, let $\mathcal{A}(X,y) = \{z \mid z \in \mathcal{A}(x,y), x \in X\}$ for a set X and an element y. Likewise, let $\mathcal{A}(X,Y) = \{z \mid z \in \mathcal{A}(x,y), x \in X, y \in Y\}$ for two sets X and Y. These conventions will be applied to all operators of sets throughout this thesis.

2.3.4 Definition of the SCM and MCM Problems

All of the elements are now in place to formally define the problems that are commonly known as the single constant multiplication (SCM) problem and the multiple constant multiplication (MCM) problem. In the remainder of this thesis, unless stated otherwise, SCM and MCM refer to the problems in Definitions 5 and 6, respectively, and (2.3) is used as the definition of the adder-operation.

Definition 5. *The single constant multiplication (SCM) problem is an instance of the problem given in Definition 3, but under the constraint that all of the targets are odd and positive integers, the definition of the adder-operation from (2.3) is used, and* $|T| = 1$.

Definition 6. *The multiple constant multiplication (MCM) problem has the same specifications as the SCM problem except that* $|T| \geq 2$.

2.3.5 Related Problems

Solving the MCM problem with an adder-operation that does not allow shifts is equivalent to solving the multiple exponentiation problem, that is, given a set of integer coefficients T, for an arbitrary x, find the minimum number of multiplications or divisions to compute the set $\{x^t \mid t \in T\}$. Further disallowing subtraction from the adder-operation is equivalent to not allowing division in the exponentiation (in which case T must contain only positive integers for a solution to exist).

Consider solving an instance of the problem given in Definition 3 under the constraint that all targets are positive integers (but not necessarily odd), $|T| \geq 2$, and we use an adder-operation that only allows addition (no subtraction or shifts,

i.e. $\mathcal{A}(x, y) = \{z \mid z = x + y\}$). The corresponding yes-no decision problem (does a solution exist for the given problem with up to n adders?) has been shown to be NP-complete [7, 8]. Both of these proofs use local replacement to obtain a reduction from the known NP-complete problem ''ensemble computation'' in [9]. It follows that finding the minimum number of adders for the given problem is NP-hard.

In the addition-shift-sequence problem presented in [10], the adder-operation is defined very differently. For the cost of one ''adder'', either one addition or one left shift is allowed. The shift can be by an arbitrary number of places and subtraction is not permitted. This problem is NP-hard since the corresponding decision problem was proven NP-complete in [10] by using a reduction from the known NP-complete problem ''vertex cover'' in [9].

To the best of our knowledge, no proof of the NP-completeness has been established for either of the two decision problems that correspond to the SCM problem and the MCM problem. Although many *variants* of said decision problems have been proved NP-complete, these proofs do not extend to encompass addition, subtraction, and shifts. When constrained to addition and left shifts only, all intermediate terms must build *up to* each target. An intermediate term that is larger than a target cannot help to construct this target. By introducing subtraction and right shifts, this fundamental limitation disappears. Unfortunately, many references incorrectly cite [7] and/or [8] for proof that the SCM and MCM problems are NP-hard. Even so, it can at least be conjectured that both the SCM and MCM problems are NP-hard, as they are more generalized versions of known NP-hard problems. We support this conjecture and to the best of our knowledge, nobody has formally challenged this conjecture since the early 1990s when the SCM and MCM problems emerged.

Chapter 3

Algorithmic Frameworks

We will *not* discuss any constant multiplication algorithms in this chapter, the existing algorithms are presented in chapter 4. However, it is necessary to first introduce the algorithmic frameworks that most of the existing algorithms use. The solution of a constant multiplication problem (the add-subtraction-shift decomposition) can be represented in many different ways. Each algorithmic framework provides a representation that enables algorithms to efficiently search for the solution. We will illustrate the beneficial properties of each framework and provide insight on how an algorithm *could* search for a solution by exploiting said properties, but we will not discuss any particular algorithm. Many examples will be provided to better illustrate the concepts. The algorithmic frameworks are presented in section 3.1 and several useful properties are discussed in section 3.2. By using the properties presented in this chapter, we will provide a more unified discussion of the existing algorithms in chapter 4. Furthermore, these will help to illustrate the similarities and differences *in the underlying intuition* of many existing algorithms.

3.1 DAG and CSE Framework Notation

In this section, we will illustrate how to represent constant coefficient multiplication with directed acyclic graphs (DAGs) and common subexpression elimination (CSE). As shown in the survey papers [11, 12], these are the two most common frameworks.

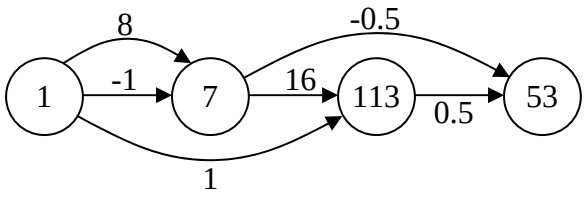
3.1.1 Directed Acyclic Graph Notation

Assume R is a valid solution to a constant multiplication problem and thus satisfies the constraints of Definition 3 (section 2.3.2). Except for $r_0 = 1$, one adder is needed to construct each element in R (recall an adder-operation may involve subtraction). For each index $k \geq 1$, the interpretation of the constraint that $r_k \in \mathcal{A}(r_i, r_j)$ where $i, j < k$ is that when a new element r_k is added to R, we must use two existing elements in R (r_i and r_j in this case) as the operands of the adder that will construct r_k. Thus there is a unidirectional dependency among the elements in R.

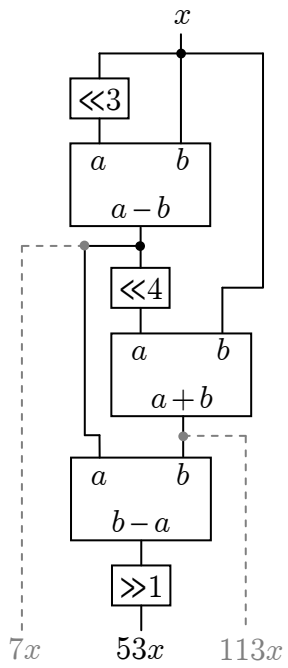
The dependencies in R are easy to encompass with a directed acyclic graph. Each node in the DAG is labeled by the value of the element in R that it represents. Except for $r_0 = 1$, each element r_k has a dependency on two elements r_i and r_j. This is represented by two directed edges from the r_i and r_j nodes to the r_k node (if $r_i = r_j$, there will be two directed edges from node r_i to node r_k). Only the $r_0 = 1$ node has no edges coming to it. This is known as the *source node* since 1 multiplied by the multiplicand comes for free. Using (2.3) as the definition of the adder-operation, if $r_k \in \mathcal{A}(r_i, r_j)$, then there exists some integers n and m such that $r_k = |2^m r_i \pm 2^n r_j|$. The shifts in the adder-operation (which are expressed as multiplication by an integer power of 2) are also encompassed by the DAG. The directed edge from r_i to r_k is labeled with the value 2^m and the edge from r_j to r_k is labeled with $\pm 2^n$.

$$7x = 8x - x = (x \ll 3) - x$$
$$113x = 16(7x) + x = ((7x) \ll 4) + x$$
$$53x = 0.5(113x) - 0.5(7x) = ((113x) - (7x)) \gg 1$$

(a) Mathematical expression



(b) Directed acyclic graph



(c) Adder tree

Figure 3.1: Three representations of the same implementation of multiplication by 53. Shifting units in the adder tree describe the routing of wires between the adders and thus incur no cost. The gray dotted lines represent constant multiplication by other coefficients that were used to construct 53 (these could be used as outputs if an MCM instance with $T = \{7, 53, 113\}$ was used).

DAGs offer a natural representation for constant multiplication. Each node in the DAG represents one addition or subtraction based on whether an edge coming to the node is labeled with 2^n or -2^n, respectively. If a directed edge from node u to node v has a weight of w, this means u is one of the two operands that are added or subtracted in order to construct v, and the left shift applied to the operand u is $\log_2 |w|$ (if $\log_2 |w| < 0$, then a right shift by $-\log_2 |w|$ is applied). Due to the one-to-one mapping between the DAG and the set of adder-operations in a valid solution, it follows that there exists a DAG representation for every valid solution.

An example of one possible way to implement multiplication by the constant coefficient 53 is given in Figure 3.1. The same constant multiplication decomposition is shown by a mathematical representation, a DAG, and an adder tree. The objective of Figure 3.1 is to illustrate the similarities between the three representations. Note that the shifting units in the adder tree in Figure 3.1(c) are used to describe the routing of wires between the adders and thus incur no cost. The adders have labeled inputs and outputs so that when subtraction is used, there is no ambiguity as to which input should be negated. The gray dotted lines in the adder tree represent constant multiplication by other coefficients that were used to construct 53. These could be used as outputs if an MCM instance with $T = \{7, 53, 113\}$ was used, for example. Note the same DAG can be used to represent the solution to a SCM instance with $t = 53$ or an MCM instance with $T = \{7, 53, 113\}$. In a constant multiplication problem with $|T|$ targets, there will be $|T|$ extraction points from within the adder-tree to serve as the constant multiplication outputs (see Figures 3.1 and 3.2). These extraction points are not always at the bottom of the adder-tree because targets can be built off of other targets in the DAG, as illustrated in Figure 3.2.

As illustrated in Figure 3.1(c), in each add-and-shift unit, the shifter can come before or after the adder. There are two ways to obtain an odd integer if odd and positive integers are used as the arguments of the adder-operation in (2.3). Adding an odd integer and an even integer produces an odd integer, however since the operands to the adder-operation are odd, one and only one of the operands must be left shifted to be made even. In this case, the shifting must be done before the adder since there is a nonzero relative shift between the two operands. The other way to obtain an odd integer is to add two odd integers and then divide the result by 2 until the result becomes odd. In this case, the shifting must be done after the adder. If $(x+y) \gg n$ is an integer, then the n least significant bits of $x+y$ are necessarily zero, however the addition of the n least significant bits of x and y may produce a carry-out that affects the result $(x+y) \gg n$. Thus for the n least significant bits in the adder, only the carry-out bits (and not the sum bits) need to be computed.

3.1.2 Common Subexpression Elimination Notation

Signed digit notation was introduced in section 2.1.2 to illustrate the CSD transform. The notation for common subexpression elimination is simply an extension of SD notation. With CSE notation, a new digit is used to define a pattern (a set of existing digits). Every instance of the pattern can then be substituted by this new digit. Let us illustrate with an example. Consider multiplication by 45, which has a CSD representation of $10\overline{1}0\overline{1}01$. Let A represent the arbitrary multiplicand that we want to multiply 45 by, then $45 \times A = A0\overline{A}0\overline{A}0A = (A \ll 6) - (A \ll 4) - (A \ll 2) + A = (2^6 - 2^4 - 2^2 + 2^0) \times A$. In section 2.1.2, we introduced $\overline{1}$ to denote -1, which means the corresponding term will be subtracted in the summation. Extending this to other

digits, $\overline{A}$ represents $-A$, for example. Let us define a new pattern $B = A0\overline{A} = (A \ll 2) - A = 3 \times A$. Thus we also have $\overline{B} = \overline{A}0A = -3 \times A$. In general, the negative of a pattern is obtained by inverting the sign of every digit in the pattern. Notice the pattern $A0\overline{A}$ or its negative occurs twice in $A0\overline{A}0\overline{A}0A$, thus B or $\overline{B}$ can be substituted in two locations to get $00B000\overline{B}$ ($B = A0\overline{A}$ is equivalent to $00B = A0\overline{A}$). Substituting $C = B000\overline{B}$, we get $000000C$, thus $45 \times A = C = (B \ll 4) - B = 15 \times B = 15 \times 3 \times A$. Note the zeros in a pattern are placeholders, i.e. we could substitute $D = A000\overline{A}$ in $A0\overline{A}0\overline{A}0A$ to get $0000D0\overline{D}$.

Signed digit patterns are a way of representing which existing terms are shifted and added to created the new term. For example, $B = \overline{A}000A$ means $B = A - (A \ll 4)$. A new pattern must consist of existing digits and have at least 2 nonzero digits. Note that $A0A0A$ only contains *one* occurrence of $A0A$ since it can only be substituted once, although this can be done in two different ways. If $B = A0A$, then we could get $A000B$ or $00B0A$, but clearly $00B0B \neq A0A0A$.

Given n nonzero digits in the CSE representation of a constant, $n - 1$ adders are needed to add the n terms. Likewise, it costs $m - 1$ adders to create a pattern with m nonzero digits. The objective in the SCM and MCM problems is to minimize the number of adders, so it is beneficial to make a pattern if it can be substituted enough times so that the number of adders saved (by reducing the number of nonzero digits) is at least as large as the number of adders needed to construct the pattern.

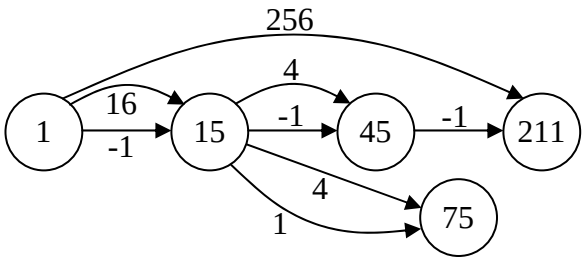
The translation from a set of CSE substitutions to the adder-tree implementation is straightforward. In Figure 3.2, we show one possible solution to the MCM problem with $T = \{45, 75, 211\}$, this same solution is illustrated with a set of CSE substitutions, a mathematical expression, a DAG, and an adder-tree implementation.

$$15x = 16x - x = (x \ll 4) - x$$
$$45x = 4(15x) - (15x) = ((15x) \ll 2) - (15x)$$
$$75x = 4(15x) + (15x) = ((15x) \ll 2) + (15x)$$
$$211x = 256x - (45x) = (x \ll 8) - (45x)$$

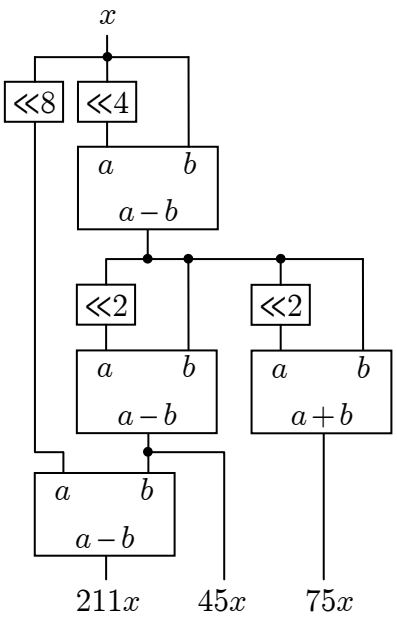
(a) Mathematical expression

Substitution	$45 \times A$	$75 \times A$	$211 \times A$
	$A0\overline{A}0\overline{A}0A$	$A0A0\overline{A}0\overline{A}$	$A0\overline{A}0A0A0\overline{A}$
$B = A000\overline{A}$	$0000B0\overline{B}$	$0000B0B$	$A00000\overline{B}0B$
$C = B0\overline{B}$	$000000C$	$0000B0B$	$A0000000\overline{C}$
$D = B0B$	$000000C$	$000000D$	$A0000000\overline{C}$
$E = A0000000\overline{C}$	$000000C$	$000000D$	$00000000E$

(b) CSE substitutions



(c) Directed acyclic graph



(d) Adder tree

Figure 3.2: Four equivalent representations of one possible solution to the MCM problem with $T = \{45, 75, 211\}$.

In Figure 3.2(b), the CSD forms of 45, 75, and 211 were used as the starting representations for the CSE substitutions. However, it is not possible to generate the same solution if we instead use the binary forms as the starting representations. For instance, in Figure 3.2(b), $45 \times A$ starts as $A0\overline{A}0\overline{A}0A$, then $B = A000\overline{A} = 15 \times A$ is substituted to give $0000B0\overline{B}$, and finally $C = B0\overline{B}$ is substituted. Now let us start with the binary representation $A0AA0A$. This still has 4 nonzero digits, so we are not starting with a representation that needs more adders than CSD form. However, there are no instances of $15 \times A = AAAA$ in $A0AA0A$ and thus this cannot be substituted. There is no need to consider any representation of $15 \times A$ that has negative digits (such as $A000\overline{A}$) since $A0AA0A$ does not contain negative digits. Clearly, the patterns that can be substituted depend on the initial representation of each constant.

Signed digit notation was first introduced in [13]. Any integer I can be expressed in the form of

$$I = \sum_{i=0} x_i 2^i \tag{3.1}$$

where $x_i \in \{\overline{1}, 0, 1\}$. For unsigned numbers, both the binary and CSD forms are valid SD forms. However, for any given integer, there are an infinite number of SD forms, as the leftmost 1 (the most significant nonzero digit) can be continually replaced by $1\overline{1}$, hence no upper limit in the summation in (3.1) is provided. For negative numbers, unlike the binary representation in 2's complement where there is a sign bit, in SD notation, the most significant nonzero digit will simply be negative. Likewise, the leftmost $\overline{1}$ can be replaced with $\overline{1}1$ repeatedly. A general algorithm for generating SD representations for a given constant is described in section 4.5.3.2.

Although it is a limitation that a pattern needs to appear in the CSE representation of a constant in order for it to be substituted, this can also be regarded as a tool to

help CSE-based heuristics prune the search space. For example, if we choose to use the CSD form of 45, since $A0\overline{A}0\overline{A}0A$ does not contain $A00\overline{A} = 7 \times A$ or its negative, a heuristic could assume that $7 \times A$ is not a useful intermediate term.

There is one critical limitation of CSE that is independent of which initial SD forms are considered. For simplicity, assume patterns have two nonzero digits. Notice the two nonzero digits in a pattern must have a nonzero relative shift between them because the two digits cannot occupy the same location in the CSE representation. For example, $A0A$ represents $(A \ll 2) + A$, however we cannot represent two instances of $A \ll 2$ (which would be $(A \ll 2) + (A \ll 2)$) in the CSE representation. Only one digit is allowed per location in the $A0A$ type of representation. In the adder-operation from (2.3), this limitation is equivalent to restricting $m \neq n$ (recall the shifts which were expressed as multiplication by 2^m and 2^n). Since the adder-operation only produces odd integers, we lose the $m = n < 0$ case, which corresponded to adding two odd integers (with *zero relative shift* between them) and then dividing the result by 2 until it becomes odd. This argument easily extends to the case where patterns have more than two nonzero digits. Consequently, not every valid solution to the SCM or MCM problem can be expressed by CSE substitutions (such as the solution in Figure 3.1). It is difficult to add an extra mechanism to deal with this issue in the general case, as an *arbitrary* number of nonzero digits can occupy the same location.

3.2 Useful Properties

In this section, we will introduce some generalized properties associated with DAGs, CSE, and the adder-operation. We will also show that there is some symmetry in the SCM and MCM problems that can be exploited by algorithms.

3.2.1 The Reflective Property of the Adder-Operation

Recall from section 2.3.3 that the adder-operation (2.3) is symmetric. The reflective property of the adder-operation, as we have named it, was first introduced and proved in [6] as a tool for finding useful intermediate terms. More specifically, it is used to compute the adder distance (the concept of adder distance is first introduced in section 3.2.3). The reflective property of the adder-operation is summarized in Theorem 1. We also provide a more detailed proof than the proof from [6]. In the remainder of this thesis, we will use ‘‘iff’’ as the short-form for ‘‘if and only if’’.

Theorem 1. *Assume x, y, and z are odd and positive integers. Iff $z \in \mathcal{A}(x, y)$, then $y \in \mathcal{A}(x, z)$.*

Proof. Let us prove the *if* part of the above theorem. If $z \in \mathcal{A}(x, y)$, then there exists integers n and m such that $z = |2^n x \pm 2^m y|$. Consider the case when $z = 2^n x + 2^m y$, then $y = 2^{n-m}x - 2^{-m}z$. When subtraction is used between x and y, we must consider two cases. If $2^n x - 2^m y \geq 0$, then $z = 2^n x - 2^m y$ can be rearranged to $y = 2^{n-m}x + 2^{-m}z$. If $2^n x - 2^m y \leq 0$, then $z = |2^n x - 2^m y| = 2^m y - 2^n x \geq 0$ can be rearranged to $y = 2^{-m}z - 2^{n-m}$. In each case, y has the form $y = |2^{n-m}x \pm 2^{-m}z|$. Recall from section 2.3.3 that there are 3 cases of shifts to consider: $m > 0$ *and* $n = 0$, $m = 0$ *and* $n > 0$, and $m = n < 0$. It can be seen that if we let $n' = n - m$ and $m' = -m$, we obtain the same cases of shifts n' and m', which are: $m' = n' < 0$, $m' = 0$ *and* $n' > 0$, and $m' > 0$ *and* $n' = 0$, respectively. Thus $|2^{n-m}x \pm 2^{-m}z|$ is an odd and positive integer and $y \in \mathcal{A}(x, z)$. The proof for the *only if* part of the theorem is analogous. □

Lemma 1. *Since the adder-operation is symmetric, it follows that iff $z \in \mathcal{A}(x, y)$, then $x \in \mathcal{A}(y, z)$, where x, y, and z are odd and positive integers.*

Lemma 2. *Recall from section 2.3.3 that* $\mathcal{A}(X, y) = \{w \mid w \in \mathcal{A}(x, y), x \in X\}$. *If* $z \in \mathcal{A}(X, y)$, *there must exist some* $x \in X$ *such that* $z \in \mathcal{A}(x, y)$. *By Theorem 1,* $y \in \mathcal{A}(x, z)$ *and thus* $y \in \mathcal{A}(X, z)$. *It follows that iff* $z \in \mathcal{A}(X, y)$, *then* $y \in \mathcal{A}(X, z)$.

A simple example that uses the property in Theorem 1 is provided at the end of section 3.2.2.

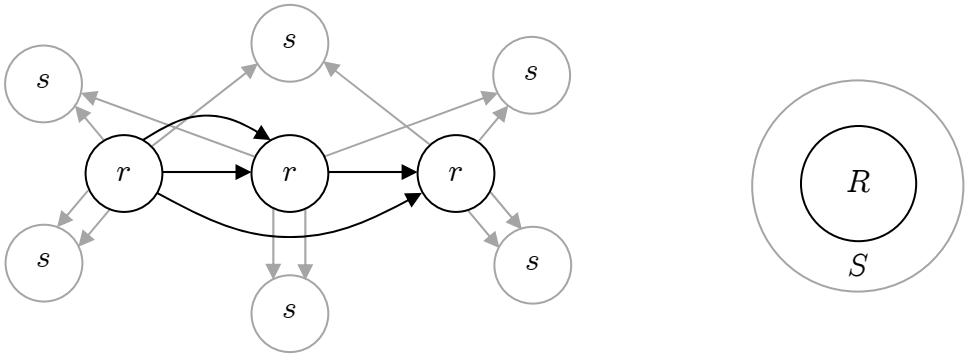
3.2.2 The Successor Set

A unifying framework for constant multiplication problems was introduced in [6]. In section 2.3.1, we introduced the ready set R and the target set T, as described in [6]. We will now introduce the successor set S from [6].

Definition 7. *The ''successor'' set* S *(with respect to* R*) contains all of the terms that can be constructed with one ''adder'' by using any two elements of* R *as the operands. Subtraction and/or shifts may or may not be allowed by the ''adder''.* S *and* R *must be mutually exclusive sets. Formally,* $S = \{s \mid s \in \mathcal{A}(r_i, r_j), r_i \in R, r_j \in R, s \notin R\}$. *By abuse of notation, we also denote this as* $S = \mathcal{A}(R, R) \backslash R$.

The exact definition of the adder-operation is deliberately *not* provided in Definition 7. Thus the successor set always has an implicit dependence on the ready set R and the type of adder-operation used. In the SCM and MCM problems, (2.3) is used as the adder-operation definition. There is no point in spending one adder to construct a term that already exists, so R and S are mutually exclusive.

Hcub, the algorithm in [6], is an iterative graph construction algorithm in which one adder is used per iteration (one term is added to R on each iteration). The successor set was introduced in [6] as a means of specifying which terms are possible



(a) Directed Acyclic Graph (b) Sets

Figure 3.3: Two representations of the successor set S with respect to the ready set R. Only the black items are already constructed, the items in gray can be constructed with one adder.

candidates for being constructed on a given iteration. Our interpretation of S is that it represents the first step along *any* possible path that leads to construction of *any* target. In other words, no matter which intermediate terms are used to construct an arbitrary target t, the first intermediate term must be in the successor set.

Constructing a new term corresponds to adding a new node in a DAG, which means we must also add two directed edges that lead to this new node. The successor set represents all possible ways of doing this. In Figure 3.3(a), given a DAG with 3 existing nodes (shown in black), all of the possible ways that one node can be added are shown in gray. Figure 3.3(b) shows the simplified representation that was introduced in [6].

In the CSE framework, with only one adder, each new pattern must have 2 nonzero digits. Thus the successor set corresponds to all possible arrangements of any two existing nonzero digits (the sign of each digit can also be chosen). However, each of these new patterns must still represent an odd and positive integer.

As an example to illustrate the reflective property of the adder-operation (from the previous section), suppose we suspect that the element $r_i \in R$ can be used as one of the operands to construct a target t with one adder. If $t \in \mathcal{A}(r_i, R)$, then there must exist a $r_j \in R$ such that $t \in \mathcal{A}(r_i, r_j)$. Thus we could check if $t \in \mathcal{A}(r_i, r_j)$ for each $r_j \in R$. By Theorem 1, this is satisfied iff $r_j \in \mathcal{A}(r_i, t)$. Instead of constructing $\mathcal{A}(r_i, R)$, it is more efficient to check for a common element between R and $\mathcal{A}(r_i, t)$, as only one adder-operation is needed. In other words, check if $R \cap \mathcal{A}(r_i, t) \neq \emptyset$.

Assume there are n elements in R and assume $t \notin R$. It follows that we can check if $t \in S$ by constructing $\mathcal{A}(R, R)$ at the expense of n^2 adder-operations, or we can check if $R \cap \mathcal{A}(t, R) \neq \emptyset$ at the expense of n adder-operations and a set intersection. However, if we want to check if *several* targets are in S, constructing $\mathcal{A}(R, R)$ may be better since it serves as a common check that can be shared by all targets.

3.2.3 An Introduction to the Adder Distance

The notion of adder distance was first introduced in [14] and later formalized in [6]. We will reuse the notation and the definition of adder distance from [6].

Definition 8. *The ''adder distance'' from a set of existing terms R to a target t is denoted as* $\text{dist}(R, t)$ *and is defined as the minimum number of adders needed to construct t starting from R.*

We have an additional interpretation of Definition 8. In order to construct the target t, *at least* $\text{dist}(R, t) - 1$ intermediate terms must first be constructed (note we can always add more terms which are functionally useless in terms of helping to construct t). For example, if $\text{dist}(R, t) = 2$, at least two adders are needed to construct t. Because t *cannot* be constructed with one adder, one adder is needed to

first construct a useful intermediate term, and then using one more adder we will be able to construct t. As another example, the successor set S contains all of the terms that can be constructed with one adder, thus if $t \in S$, then t is an adder distance of 1 from R. In this case, t can be constructed without adding any intermediate terms. If $t \in R$ (t is already constructed), then t is an adder distance of 0 from R.

In order to optimally solve the SCM problem for a given target t, we need to find $\text{dist}(\{1\}, t)$. The idea of classifying every constant according to its optimal SCM cost was introduced in [6]. We will reuse their definition and notation.

Definition 9. *The set of "complexity n" constants is denoted as C_n and is defined as $C_n = \{t \mid dist(\{1\}, t) = n\}$.*

As stated in section 2.3.5, it is conjectured that the SCM problem is NP-hard. Thus computing the adder distance is not easy. Even so, many algorithms *estimate* the adder distance in order to quantify how far each target is and also to determine how useful an intermediate term is. There are many methods for estimating the adder distance with varying degrees of accuracy and computational effort; methods will be discussed with their corresponding algorithms. After we introduce some concepts in sections 3.2.4 and 3.2.5, we will then show an exact method for computing the adder distance in section 3.2.6.

Consider a SCM problem for a given target t. Assume we have already constructed some terms (which are placed in R) and we have estimated $\text{dist}(R, t)$. Now we would like to calculate how useful an intermediate term x is. If we construct x (add x to R), we can estimate $\text{dist}(R \cup \{x\}, t)$, which represents the remaining adder distance to the target. Therefore $\text{dist}(R, t) - \text{dist}(R \cup \{x\}, t)$ expresses how much closer we will move to t if x is constructed. Assume x requires m adders to be constructed. If x

is useful, we expect $m \leq \text{dist}(R,t) - \text{dist}(R \cup \{x\}, t)$. In other words, for the cost of m adders, we should move at least m adders closer to the target. Conclusively, $\text{dist}(R,t) - \text{dist}(R \cup \{x\}, t) - m$ can be used as a metric in deciding how useful an intermediate term x is.

Let us illustrate how the adder distance can be *estimated* with an example. We will use the CSE framework and constrain all patterns to have 2 nonzero digits (so it always costs one adder to construct each pattern). Assume we have a partially solved SCM problem and there are n remaining nonzero digits in the CSE form of the target t. We need $n-1$ adders to add these n terms, thus $\text{dist}(R,t) \approx n-1$. Let us construct a pattern p. If p can be substituted in k locations in the CSE form of t, $2k$ existing nonzero digits will be replaced by k new nonzero digits, thus a *net loss* of k nonzero digits. Therefore $\text{dist}(R \cup p, t) \approx n-1-k$. It costs one adder to make p, thus $\text{dist}(R,t) - \text{dist}(R \cup p, t) - 1 = k-1$ is the metric that determines how useful p is. Clearly, we should substitute the pattern that occurs the most (maximum k).

The adder distance can also be used to guide a MCM heuristic. Hcub (section 4.3.3) and our proposed algorithms H3 and H4 (sections 6.1.3 and 6.1.4, respectively) are examples of this.

3.2.4 Multiplicative Decompositions and Division Tests

We will show an important property of the set C_1. Using $R = \{1\}$, with one adder we can to create any element in $C_1 = \mathcal{A}(1,1) = \{z \mid z = |2^n 1 \pm 2^m 1|\}$. The adder-operation used in the SCM and MCM problems is constrained to produce odd and positive integers. Recall from section 2.3.3 that there were 3 cases to consider: $m > 0$ *and* $n = 0$, $m = 0$ *and* $n > 0$, and $m = n < 0$. Because of the symmetry in C_1, we can

ignore the $m > 0$ *and* $n = 0$ case. Recall the $m = n < 0$ case corresponds to adding two odd integers and then dividing the even result by 2 until it becomes odd, so in C_1 we get $1 + 1 = 2 \Longrightarrow 1$. Thus, we have

$$C_1 = \{z \mid z = 2^n \pm 1, n \in \mathbb{Z}, n \geq 1\}. \tag{3.2}$$

By an argument similar to that above, it follows that

$$\mathcal{A}(x, x) = \{z \mid z = 2^n x \pm x, n \in \mathbb{Z}, n \geq 1\} = C_1 \cdot x. \tag{3.3}$$

Note the multiplication of sets is done in the usual element-wise manner, i.e. $U \cdot V = \{u \cdot v \mid u \in U, v \in V\}$, and we interpret $C_1 \cdot x$ as $C_1 \cdot \{x\}$. Likewise, for the division of integer sets, $U/V = \{u/v \mid u \in U, v \in V, u/v \in \mathbb{Z}\}$.

Now consider a partially completed SCM problem for a given target t. Assume there are some already constructed terms in R so that $\text{dist}(R, t) = 2$. As explained in section 3.2.3, one intermediate term s is needed in order to construct t, where $s \in S$. Now assume we construct s, which means with one more adder we can construct t. There are only two ways to do this: $t \in \mathcal{A}(s, s)$ or $t \in \mathcal{A}(s, R)$. If we do not use s as one or both of the operands, then only elements of R can be used as operands, but $t \notin \mathcal{A}(R, R)$ because t is not distance 1. The first case, $t \in \mathcal{A}(s, s) = C_1 \cdot s$, is an example of a *multiplicative* decomposition, as multiplication by an element in C_n is needed (in this case $n = 1$). The second case, $t \in \mathcal{A}(s, R) = \mathcal{A}(\mathcal{A}(R, R), R)$, is an example of an *additive* decomposition since three instances of some elements in R are ‘‘added’’ to create t. We will discuss additive decompositions in the next section.

Assume there are two nodes p and q within a DAG such that $q = c_n \cdot p$, where $c_n \in C_n$. Also assume that q was constructed by adder-operating only instances of p.

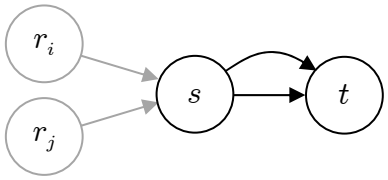


Figure 3.4: The distance 2 graph topology $t \in \mathcal{A}(s, s) = C_1 \cdot s$, where $s \in S$. Nodes s and t form a multiplicative subgraph (shown in black). Note that $r_i, r_j \in R$.

More than one adder-operation can be used, for example $q = \mathcal{A}(\mathcal{A}(p, p), p) = C_2 \cdot p$. Given this, all of the nodes between p and q (including p and q) form a multiplicative subgraph. In the $t \in \mathcal{A}(s, s) = C_1 \cdot s$ example above, the nodes s and t form a multiplicative subgraph, as shown by the black part of the graph in Figure 3.4.

Now let us consider how we would establish whether or not $t \in C_1 \cdot s$, where $s \in S$ (i.e. whether or not $t \in C_1 \cdot S$). If this is satisfied, there must exist a $c_1 \in C_1$ and a $s \in S$ such that $t = c_1 \cdot s$, or equivalently, $s = t/c_1$. This means that the only values of s that will permit the construction of t are $s \in t/C_1$, where the set $t/C_1 = \{t/c_1 \mid c_1 \in C_1, t/c_1 \in \mathbb{Z}\}$. If $c_1 > t$, then $t/c_1 \notin \mathbb{Z}$. Thus the value of t places an upper bound on the shift value n in $C_1 = \{z \mid z = 2^n \pm 1, n \in \mathbb{Z}, n \geq 1\}$. If no element in C_1 *properly divides* t (i.e. divides t with zero remainder), then $t/C_1 = \emptyset$ is sufficient proof that $t \notin C_1 \cdot S$. Since $s \in S$ and $s \in t/C_1$, we need a common element between t/C_1 and S. Conclusively, iff $t/C_1 \cap S \neq \emptyset$, then $t \in C_1 \cdot S$. This type of division test was proposed in [6] for computing the adder distance, which is explained in section 3.2.6.

A classification of directed acyclic graphs was done in [2]. As explained in [2], multiplicative graphs are composed of two independent subgraphs such that the output node of the first subgraph is the same node as the input node of the second subgraph. It is possible that t may be decomposed into a product of more than two terms, thus

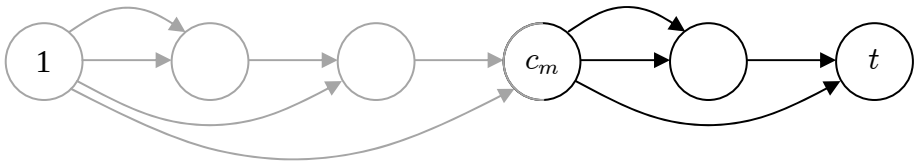


Figure 3.5: The decomposition of a multiplicative graph into two subgraphs.

the subgraphs can also be multiplicative graphs. Thus a target t can be decomposed as $t = c_m \cdot c_n$, where $c_m \in C_m$ and $c_n \in C_n$. An example with $m = 3$ and $n = 2$ is shown in Figure 3.5. The c_m node belongs to both subgraphs. The first subgraph of the DAG (left side in gray) constructs c_m while the second subgraph (in black) implements multiplication by c_n. Given an adder tree which implements $c_n \cdot x$, if we put x at the input, we receive $c_n \cdot x$ at the output. However, if we put $c_m \cdot x$ at the input, we receive $c_m \cdot c_n \cdot x = t \cdot x$ at the output.

Suppose a SCM heuristic discovered that the given target t could be decomposed as $t = a \cdot b$. One strategy could be to implement SCM for a and SCM for b, then multiplicatively combine the DAGs. There could be several decompositions of this form, so the best solution could be selected. This could also be applied recursively. If t is a prime number, we could instead try to decompose t as $t = a \cdot b + c$.

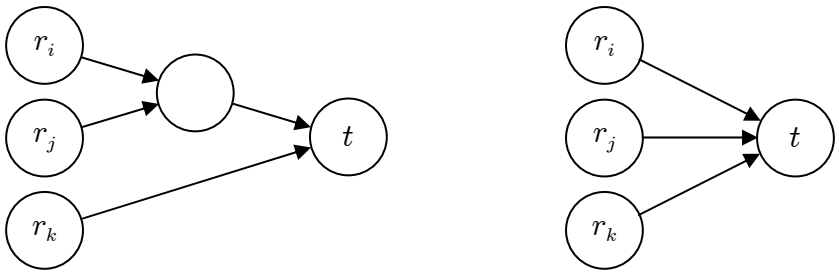
3.2.5 Additive Decompositions and Vertex Reduction

Recall the example in section 3.2.4 where $t \in \mathcal{A}(\mathcal{A}(R, R), R)$. We finished constructing t by adding 3 instances of some elements $r \in R$ with 2 adders. These 2 adders are an example of an additive decomposition. To exploit the symmetry in an additive decomposition, vertex reduction can be used in the corresponding DAG. Vertex reduction was introduced in [2] as a means to identify equivalent graph topologies (for algorithms that use the DAG framework), thereby avoiding redundant searching.

Before we discuss vertex reduction, let us first illustrate the symmetry that can be exploited in an additive decomposition. Recall that the adder-operation is symmetric. For any three elements $r_i, r_j, r_k \in R$, it was formally proven in [2] that $\mathcal{A}(\mathcal{A}(r_i, r_j), r_k)$ and $\mathcal{A}(\mathcal{A}(r_i, r_k), r_j)$ are identical sets. We will illustrate the basic idea. Each element in the set $\mathcal{A}(\mathcal{A}(r_i, r_j), r_k)$ has the form $|2^a|2^b r_i \pm 2^c r_j| \pm 2^d r_k|$. If we combine the 2^a shift with the 2^b and 2^c shifts, the form simplifies to $|\pm 2^m r_i \pm 2^n r_j \pm 2^d r_k|$. In this representation, it is apparent that we can freely interchange order of the operands r_i, r_j, and r_k since the set $\mathcal{A}(\mathcal{A}(r_i, r_j), r_k)$ contains *every* element of this form. Thus, it is more natural to represent this set as $\mathcal{A}(r_i, r_j, r_k)$. In order to exploit this symmetry, we extend the definition of the adder-operation to (3.4), in which an arbitrary number of operands are permitted. In (3.4), the operands are denoted by x_i and the shift of operand x_i is denoted by s_i, for $i = 1, \ldots, n$.

$$\mathcal{A}(x_1, \ldots, x_n) = \{z \mid z = |\pm 2^{s_1} x_1 \pm \ldots \pm 2^{s_n} x_n|, s_i \in \mathbb{Z}, z \in \mathbb{Z}, z/2 \notin \mathbb{Z}\} \qquad (3.4)$$

Each term created always has a corresponding node in the DAG. Previously, an adder-operation only had two operands, so each term had two dependents and was constructed by using one addition (or subtraction). Under these conditions, each node in the DAG represented one adder and two directed edges came to each node to represent the dependents. Now, a new term can be constructed using an adder-operation with n operands ($n \geq 2$). In this case, this new term has n dependents, so the node in the DAG that corresponds to this term must have n directed edges coming to it. It follows that a node with n directed edges coming to it represents a term that costs $n-1$ adders to construct. In Figure 3.6, we show the vertex reduction from $t \in \mathcal{A}(\mathcal{A}(r_i, r_j), r_k)$ to $t \in \mathcal{A}(r_i, r_j, r_k)$.



(a) $t \in \mathcal{A}(\mathcal{A}(r_i, r_j), r_k)$ (b) $t \in \mathcal{A}(r_i, r_j, r_k)$

Figure 3.6: Vertex reduction applied to an additive decomposition.

Assume t is not distance 1 (or else we would not bother testing if t could be constructed with 2 adders). Consider how we can establish whether or not $t \in \mathcal{A}(R, R, R)$, which is equivalent to $t \in \mathcal{A}(\mathcal{A}(R, R), R)$. We could simply compute $\mathcal{A}(R, R, R)$ but there is a more efficient method to check for membership in this set. If $t \in \mathcal{A}(\mathcal{A}(R, R), R)$, there must exist an element $x \in \mathcal{A}(R, R)$ such that $t \in \mathcal{A}(x, R)$. By Lemma 2 from section 3.2.1, this is satisfied iff $x \in \mathcal{A}(t, R)$ for some $x \in \mathcal{A}(R, R)$. In other words, there must be a common element between $\mathcal{A}(t, R)$ and $\mathcal{A}(R, R)$, or equivalently $\mathcal{A}(t, R) \cap \mathcal{A}(R, R) \neq \emptyset$. An element in $\mathcal{A}(R, R)$ could also be in R, but this would mean $t \in \mathcal{A}(\mathcal{A}(R, R), R) = \mathcal{A}(r, R)$ where $r \in R$ and thus t would be distance 1. Therefore we are only interested in elements in $\mathcal{A}(R, R)$ that are in S. Instead of testing $\mathcal{A}(t, R) \cap \mathcal{A}(R, R) \neq \emptyset$, we test $\mathcal{A}(t, R) \cap S \neq \emptyset$. Conclusively, t is distance 2 and $t \in \mathcal{A}(R, R, R)$ iff $\mathcal{A}(t, R) \cap S \neq \emptyset$.

Theorem 1 states that iff $t \in \mathcal{A}(r_i, r_j)$, then $r_i \in \mathcal{A}(t, r_j)$. By using vertex reduction, we can extend this to iff $t \in \mathcal{A}(r_i, r_j, r_k)$, then $r_i \in \mathcal{A}(t, r_j, r_k)$. This was illustrated in the above example. In fact, this extends to an arbitrary number of operands. This was not shown in [6]. Since we extended the definition of the adder-operation, we shall also extend its reflective property. The proof of Theorem 2 is analogous to the

proof of Theorem 1 and should be obvious from the $t \in \mathcal{A}(R, R, R)$ example above. Although not needed, Lemmas 1 and 2 could also be extended in a similar manner.

Theorem 2. *Assume z, x_1, x_2, ..., x_n are odd and positive integers. Iff $z \in \mathcal{A}(x_1, x_2, \ldots, x_n)$, then $x_1 \in \mathcal{A}(z, x_2, \ldots, x_n)$.*

Although vertex reduction was developed for DAGs, we have extended its application to the CSE framework. For example, consider $2451 \times A$, which has a CSD representation of $A0A0\overline{A}00A0A0\overline{A}$. If only one adder can be used to construct each new term, this is equivalent to enforcing that all new CSE patterns have only 2 nonzero digits. In this case, we could substitute $B = A0\overline{A}$ to get $A000B00A000B$, and then substitute $C = A000B$ to get $0000C000000C$. However, we ultimately substitute two occurrences of $A0A0\overline{A}$, so it does not matter whether we first build $A0A$, $A000\overline{A}$, or $A0\overline{A}$. As explained in section 3.2.3, if patterns only have 2 nonzero digits, a steepest-descent CSE algorithm substitutes the pattern that occurs the most. In this case, we have a three-way tie, so an algorithm could choose to substitute all three options (one at a time), in which case it would go to the same second substitution three times. To avoid this redundancy, a CSE algorithm could take advantage of the additive decomposition by immediately constructing $A0A0\overline{A}$ at a cost of two adders.

3.2.6 Computing the Exact Adder Distance

In section 3.2.3, we introduced the notion of adder distance and provided several examples to illustrate how and why it is useful. We will now present an exact method for *computing* the adder distance. Recall from Definition 8 that $\text{dist}(R, t)$ is the minimum number of adders needed to construct t starting from R. The problem can be simplified to determining whether or not the target t can be constructed with n

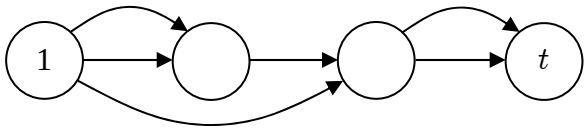
adders. By doing an exhaustive search for each n and in the order of increasing n, we are guaranteed to find the minimum number of adders needed to construct t.

An exhaustive search can only be done using directed acyclic graphs. As explained at the end of section 3.1.2, CSE has an inherent limitation. Since only one nonzero digit can occupy each location in the CSE representation of a constant, we lose the $m = n < 0$ class of shifts in the adder-operation (recall that shifts were expressed as multiplication by 2^m and 2^n).

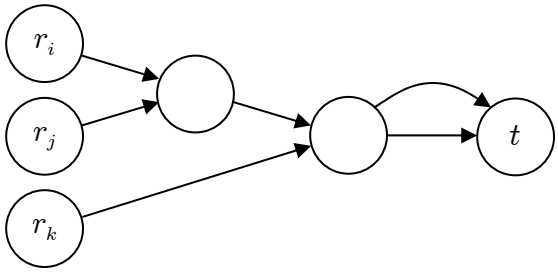
There are only so many ways in which we can arrange n adders, thus we can enumerate all of the possible corresponding DAG topologies. All of the possible vertex reduced graph topologies for up to 5 adders are shown in Figure 5 in [2]. These represent all of the possible ways of using n adders ($n \leq 5$) to construct a target t. In SCM, we are constrained to put $r_0 = 1$ at the source node of a DAG. However, when computing the adder distance, we are allowed to use any of the existing in R to construct t. It follows that by splitting the source node and asserting an element in R at each of the newly split source nodes, we obtain the equivalent topology for computing whether or not we can construct t using this topology starting with elements in R. An example is illustrated in Figure 3.7.

If t can be constructed with n adders, *at least* one graph topology with n adders can construct t. Thus, for *each* topology with n adders, we need to design a test to determine if t can be constructed with this topology. If *none* of these corresponding tests succeed, then t cannot be constructed with n adders.

The avid reader should realize that we have already done this for distance 1 and 2. For distance 1, we can only adder-operate two elements in R. As shown in section 3.2.2 (where we presented the successor set), for distance 1, we can either test if $t \in S$



(a) Graph topology for SCM.



(b) Graph topology for adder distance.

Figure 3.7: By splitting the source node of a SCM graph topology, we obtain the corresponding adder distance graph topology. Note that $r_i, r_j, r_k \in R$.

or if $\mathcal{A}(t, R) \cap R \neq \emptyset$. For distance 2, we must first construct one intermediate term before t can be constructed. As explained in section 3.2.4, there are only two ways in which this can be done: $t \in C_1 \cdot S$ or $t \in \mathcal{A}(R, S)$. These correspond to the two cost 2 graph topologies shown in Figure 4 of [1] or Figure 5 of [2]. In section 3.2.4, using a division-based test, we proved that iff $t/C_1 \cap S \neq \emptyset$, then $t \in C_1 \cdot S$. In section 3.2.5, we proved that iff $\mathcal{A}(t, R) \cap S \neq \emptyset$, then $t \in \mathcal{A}(R, S)$.

We have named the method used to derive the above tests as the inverse graph traversal (IGT) method. The target is used to find all of the useful intermediate terms, where said intermediate terms are one adder away from the target. In the general case, these useful intermediate terms are then used to find all of the useful intermediate terms that are two adders away from the target, and so on, until we reach S or R to check for a common element.

Alternatively, for distance 2, we could compute both $C_1 \cdot S$ and $\mathcal{A}(R, S)$. Let us define the 2^{nd} order successor set $S_2 = (C_1 \cdot S) \cup \mathcal{A}(R, S)$, then a target t is distance 2 if $t \in S_2$. This method could be called forward graph traversal since we build *up to* to the target. This approach is used in the MAG algorithm (sections 4.1.1 and 4.1.2).

The computational effort of these tests depends largely on the size of R, S, and T. If R is small but there are numerous targets, then creating S_2 may be a more efficient test because S_2 serves as a common checking criteria for all of the targets. Conversely, if R is large and there are only a few remaining targets, using IGT may be more efficient. In general, IGT is superior because S is typically much larger than R or T, thus much more computational effort is needed to construct $C_1 \cdot S$ than T/C_1, and more computational effort is needed to construct $\mathcal{A}(R, S)$ than $\mathcal{A}(R, T)$.

As one should expect, for distance 3, we need to enumerate all of the possible graph topologies with 3 adders and then design a constructability test for each topology. For example, the topology in Figure 3.7 corresponds to $t \in \mathcal{A}(R, S) \cdot C_1$. It can be shown that the corresponding test is $\mathcal{A}(\frac{t}{C_1}, R) \cap S \neq \emptyset$. Distance 3 tests are discussed in detail in section 6.1.3.1, where we also illustrate another property that can be exploited. Distance 4 tests are summarized in section 6.1.4 and some distance 5 tests are derived in section 5.2.2. Note that in practice, the adder distance can only be computed exactly for small n. The number of unique vertex reduced graph topologies grows very quickly with n. As computed in [2], there are 2, 5, 15, 54, 227 unique vertex reduced graph topologies at adder distance 2, 3, 4, 5, and 6, respectively. Also, the test for each topology requires more computation as the number of adders increase since we have to traverse through more nodes. To the best of our knowledge, we are the first to attempt distance 4 and 5 tests using IGT.

Chapter 4

Existing Algorithms

In this chapter, we will provide a survey of existing techniques for solving the SCM, MCM, and other closely related problems. Since the existing algorithms span over two decades, we will only cover the algorithms which are closely related to the contributions in this thesis and the algorithms which proposed the fundamental approaches and ideas that were frequently reused and/or improved by later algorithms.

We *strongly advise* the reader that much of the discussion in this chapter utilizes the concepts introduced in chapter 3. We will first provide an overview of what we will cover in this chapter. In sections 4.1 to 4.5, we will discuss exhaustive optimal algorithms, an overview of iterative heuristic algorithms, DAG-based algorithms which use a bottom-up approach, DAG-based algorithms which use a top-down approach, and CSE-based algorithms, respectively. Other existing approaches for solving the MCM problem are *very briefly* discussed in section 4.6. In section 4.7, we introduce problems related to SCM and MCM which have different objectives and/or metrics that the algorithms try to minimize. Finally, some theoretical lower and upper bounds on the SCM and MCM problems are presented in section 4.8.

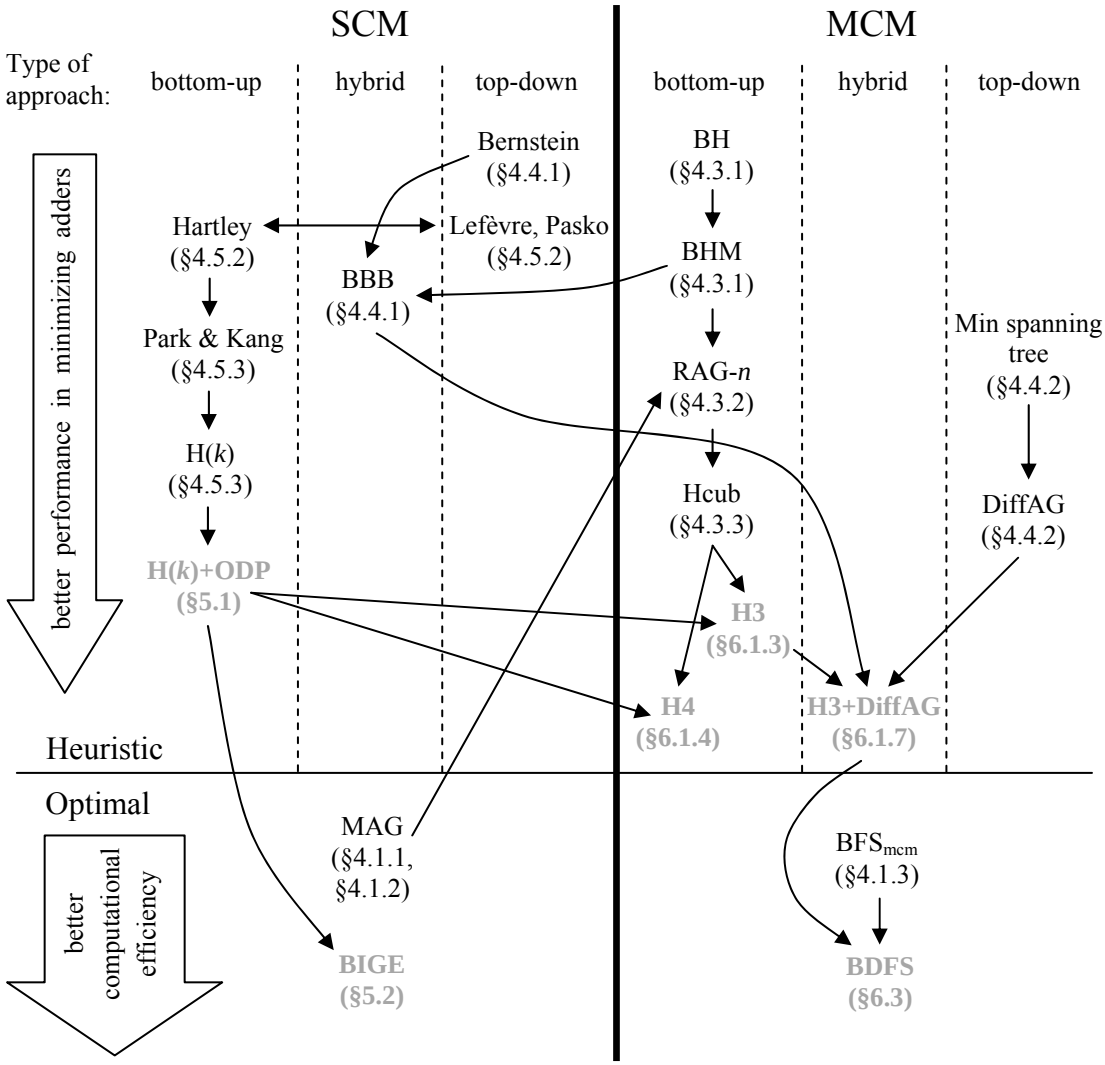


Figure 4.1: A summary of the algorithms discussed in this thesis. The arrows denote *some* relationship between the algorithms which is discussed in the corresponding sections. Our contributions are shown in gray (existing algorithms are in black). The performance of algorithms in terms of minimizing adders is only compared *within* SCM and *within* MCM (i.e. the relative vertical placement on the left side is *unrelated* to the right side). Optimal algorithms are not classified by the type of approach.

Figure 4.1 provides a summary of the algorithms we will discuss in this thesis. An arrow between algorithms denotes that the later algorithm reuses some approach or idea presented in the earlier algorithm. The exact nature of each dependency will be discussed in the corresponding sections. The existing algorithms are shown in black and the algorithms we propose in this these algorithms are shown in gray.

The performance of algorithms in terms of minimizing adders is only compared *within* SCM and *within* MCM (i.e. the vertical placement on the left side of Figure 4.1 is *unrelated* to the right side). We deliberately defined the SCM and MCM problems as two mutually exclusive problems since the best performing algorithm(s) for each problem are designed differently. Every SCM algorithm can solve the MCM problem by solving an SCM problem for each target in T and then collecting all intermediate terms used in any of the SCM instances into one set R. However, the inherent inability to share intermediate terms *between* targets suggests this is a poor approach. Every MCM algorithm can solve a SCM problem, as a SCM problem for a given target t is equivalent to a 2 constant MCM problem with $T = \{1, t\}$. SCM algorithms *only* share terms *within* one constant whereas MCM algorithms *also* share terms *between* targets, so a SCM algorithm can be fine-tuned to better solve just the SCM problem.

Note that a SCM algorithm can be used to solve a subproblem in the MCM problem (for example, from Figure 4.1, RAG-n uses the MAG algorithm). The BBB algorithm introduced the idea of using a hybrid algorithm composed of two subalgorithms, each with very different approaches. Better solutions are obtained by searching different areas of the solution space. We reuse this idea in our proposed H3+DiffAG hybrid algorithm. As shown in section 6.1, it is the best performing MCM heuristic (optimal exhaustive algorithms can only solve small problem sizes).

Within each subsection, the algorithms will be presented roughly in chronological order, as the older algorithms provided the fundamental ideas that were later reused and/or improved by later and more sophisticated algorithms. For example, a later algorithm could use an exact method to solve a subproblem that was previously solved by a heuristic. Also, the algorithms span more than two decades, over which time the amount of compute power has increased by orders of magnitude. The older algorithms are quite simple compared to the algorithms of today, but in fairness the algorithms of today would not have been practical one decade ago.

Many of the algorithms we will discuss predate the formalization of concepts like the successor set, adder distance, and the common sets like the ready set R, the successor set S, the target set T, and the complexity-n sets C_n. However, we will describe the algorithms with these unifying concepts in order to better emphasize the similarities and differences between the existing algorithms.

4.1 Exhaustive Search Methods

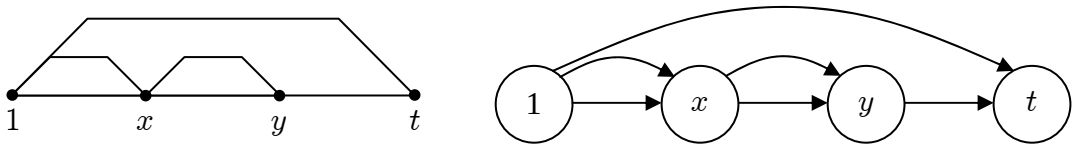
In this section, we will illustrate how an exhaustive search can be done to find an optimal SCM or MCM solution. This can only be done using directed acyclic graphs. As explained at the end of section 3.1.2, CSE has an inherent limitation. Since only one nonzero digit can occupy each location in the CSE representation of a constant, we lose the $m = n < 0$ class of shifts in the adder-operation (recall that shifts were expressed as multiplication by 2^m and 2^n).

4.1.1 The MAG Algorithm (Optimal SCM)

To the best of our knowledge, Dempster and Macleod [1] proposed the first optimal SCM algorithm in 1994, which was named the minimized adder graph (MAG) algorithm. It exhaustively checks whether or not a solution with 1 adder exists. If found, the algorithm stops, otherwise it exhaustively checks for 2 adders, then 3 adders, and so on. By exhaustively checking in the order of increasing adder cost, we are guaranteed to find the optimal solution (with the minimum number of adders).

With n adders, there are only so many possible ways of arranging add-subtract-shift units in an adder tree. We can therefore enumerate all of the possible corresponding graph topologies. In Figure 4 in [1], all of the possible graph topologies are shown for $n = 1, 2, 3, 4$. For each graph topology, we compute all of the possible values that could be constructed at the ending node in the DAG. We can then compare these values with the given SCM target t to determine if the graph topology can construct t. Since we search in the order of increasing adder cost, if we compute all possible values for all of the graph topologies with n adders and we still cannot construct t, this is sufficient proof that t requires *at least* $n + 1$ adders.

Due to the ‘‘first construct, then check against the target’’ nature of the MAG algorithm, it can be used to find the optimal SCM solution for several targets at the same time. In fact, Dempster and Macleod used the MAG algorithm to generate the optimal SCM solutions for all constants up to a bit width of 12. The algorithm creates a lookup table so that later when a SCM problem needs to be solved for a given target t, the solution can be accessed from the table. The search was limited to 4 adders due to the limited compute power in desktop computers at the time. In [1], it was discovered that all constants up to 12 bits can be constructed with 4 adders.



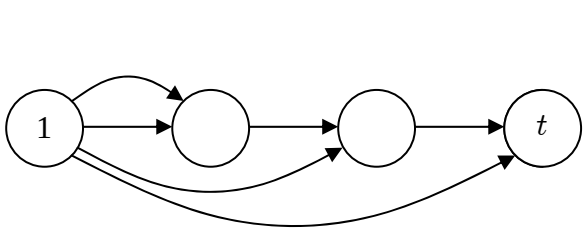
(a) Original graph (b) DAG

Figure 4.2: Graph number 4, cost 3, in [1] enables the construction of $\mathcal{A}(1, C_1 \cdot C_1)$.

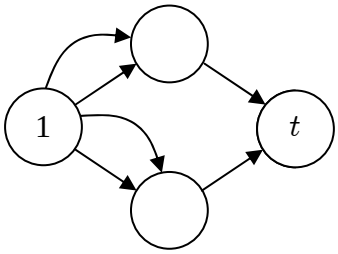
Let us illustrate how to generate all of the possible values for a given graph topology. Consider graph number 4, cost 3, in Figure 4 in [1]. We show the original graph in Figure 4.2(a) and we translate it to our style of DAGs in Figure 4.2(b). Note that [1] predates the notion of the complexity-n sets C_n, however we will use this to simply our explanations. From the labeling in Figure 4.2, clearly $x \in \mathcal{A}(1, 1) = C_1$, $y \in \mathcal{A}(x, x) = C_1 \cdot x$, and $t \in \mathcal{A}(1, y)$. Thus it follows that $t \in \mathcal{A}(1, C_1 \cdot C_1)$. It was proven in [1] that the shifts that we need to consider in each adder-operation are bounded above by $O(b)$, where b is the bit width of t. In other words, even if shifts larger than this bound are used, a better solution for t will not be found. The bound is $b + 2$ for this particular graph topology [1].

4.1.2 Extension of the MAG Algorithm

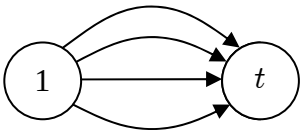
In [2], an analysis and classification of graphs was performed to gain insight on how the we can exploit the symmetry between different graph topologies of the same cost. As a result, vertex reduction was introduced for additive graphs (as discussed in section 3.2.5) and partitioning was introduced for multiplicative graphs (as discussed in section 3.2.4). The identification of unique vertex reduced DAGs facilitated a more efficient exhaustive search. In 2002, the MAG algorithm was extended to 5 adders



(a) Graph number 1 (cost 3) from [1]
$t \in \mathcal{A}(\mathcal{A}(\mathcal{A}(1,1),1),1)$



(b) Graph number 7 (cost 3) from [1]
$t \in \mathcal{A}(\mathcal{A}(1,1),\mathcal{A}(1,1))$



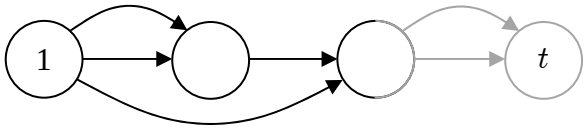
(c) Equivalent vertex reduced DAG
$t \in \mathcal{A}(1,1,1,1)$

Figure 4.3: The merging of equivalent graphs (a) and (b) via vertex reduction produces (c).

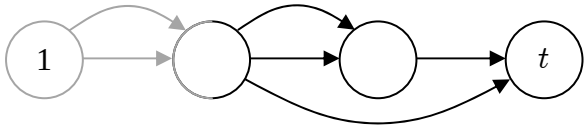
in [2], and it was discovered that all constants up to 19 bits have an optimal SCM solution with no more than 5 adders.

As an example of identifying redundant graphs via vertex reduction, graphs number 1 and 7 (cost 3) in Figure 4 in [1] are equivalent. Graph number 1 produces $\mathcal{A}(\mathcal{A}(\mathcal{A}(1,1),1),1)$ and graph number 7 produces $\mathcal{A}(\mathcal{A}(1,1),\mathcal{A}(1,1))$, both of these essentially add (or subtract) 4 items of the form $1 \ll n$, where n is some arbitrary shift. As shown in Figure 4.3, both of these graphs are equivalent to the vertex reduced graph corresponding to $\mathcal{A}(1,1,1,1)$.

As explained in [2], multiplicative graphs can be decomposed into two subgraphs such that the ending node of the first subgraph is the same node as the source node of the second subgraph. Assume the target t can be decomposed as $t = a \cdot b$, where a



(a) Graph number 3 (cost 3) from [1], $t \in \mathcal{A}(1, C_1) \cdot C_1$



(b) Graph number 5 (cost 3) from [1], $t \in C_1 \cdot \mathcal{A}(1, C_1)$

Figure 4.4: Transposing a multiplicative graph does not change set of possible outputs. The gray subgraph implements multiplication by C_1 and the black subgraph implements multiplication by $\mathcal{A}(1, C_1)$.

and b are integers. The first subgraph implements multiplication by a and the second subgraph by b. However, we will still construct t even if we interchange the position of the two subgraphs, i.e. the ending node of the second subgraph now becomes the same node as the source node of the first subgraph. It follows that graphs number 3 and 5 (cost 3) in Figure 4 in [1] are equivalent and thus only one of them needs to be searched in the MAG algorithm. As shown in Figure 4.4, graph number 3 produces $\mathcal{A}(1, C_1) \cdot C_1$ and graph number 5 produces $C_1 \cdot \mathcal{A}(1, C_1)$. Using the naming from [2], graph number 3 is the ‘‘transpose’’ of graph number 5, and vice versa.

The transpose is obtained by taking the left/right reflection of the vertex reduced graph. In SCM, a graph and its transpose produce the same set of possible values (since the source node is constrained to $r_0 = 1$). However, when computing the adder distance (as discussed in section 3.2.6), once the source node is split, we obtain different topologies. Thus for adder distance, we must consider both topologies.

4.1.3 An Optimal MCM Algorithm

To the best of our knowledge, the first optimal MCM algorithm BFS_{mcm} [15] was proposed in 2008. It is a DAG-based breadth-first exhaustive search which uses some pre-computed sets, but with *no pruning*. It was only proposed recently due to limits in the amount of compute power, which has significantly increased over the years. BFS_{mcm} can only solve small problem sizes, however [15] showed that the heuristic Hcub (section 4.3.3) is close to optimal. We depend on this fact when we propose a more compute-efficient optimal MCM algorithm *with pruning* in section 6.3.

Earlier attempts have been made at solving MCM optimally by using CSE and 0-1 integer linear programming (0-1 ILP) to find the maximal sharing of intermediate terms. Examples of such include [16--21]. However, the results of [15] clearly indicate that BFS_{mcm} provides better solutions than the *exact* CSE-based method in [20] which maximizes the sharing of intermediate terms. As explained in detail in section 3.1.2, the terms that can be shared depend on the starting CSE form. To the best of our knowledge, only the binary form or signed-digit (SD) forms with the minimum number of nonzero digits were considered by all of the existing CSE/ILP algorithms. As discussed in section 4.5.3.1, better solutions can be found by considering other SD forms, but as shown in section 5.1.4, *numerous* SD forms are needed. Given the present amount of compute power, it is infeasible to use an exact method for the large number of SD forms that are required to produce very good solutions.

While CSE is a great tool for *heuristics*, the results from [15] clearly indicate that graphs are more suitable than CSE for exhaustive searches. For this reason, all of the *exact* approaches that we will use in this thesis are graph based, for both optimal algorithms and subproblems (like computing the adder distance).

4.2 An Overview of Iterative Heuristics

All of the algorithms in sections 4.3, 4.4, and 4.5 are iterative heuristics. On each iteration, one or more terms are constructed (added to R) until $T \subseteq R$ at which point the algorithm ends. Let us denote the remaining targets as T', thus each algorithm initializes $T' = T$, when a target t is constructed then t is removed from T', and each algorithm ends when $T' = \emptyset$. We will therefore describe the behavior of these algorithms *for the current iteration*. Any sets associated with the algorithm (such as R, S, and T') must be updated *between* iterations.

When intermediate terms are needed (i.e. if no target is distance 1), algorithms have the choice to construct intermediate terms that are close to the targets (close in terms of adder distance) or intermediate terms that are close to the existing terms in R. It only makes sense to choose an intermediate term that is closer than each of the targets that it will help to construct. Thus in the first approach, over a few iterations we should expect the intermediate terms to get progressively closer to the existing terms. This is a top-down approach, as we build *down from* the targets. Conversely, if we choose intermediate terms that are close to the existing terms (for example, those in the successor set), over several iterations we will build *up to* the targets, thus this is a bottom-up approach. Algorithms typically use only one approach, however they can be combined to each solve different subproblems in a hybrid algorithm.

One interesting case is distance 2 targets. For a target t at distance of 2 from R, there must be at least one intermediate term which is distance 1 from R and that is also within one adder from t. If an algorithm can *detect* distance 2 targets as well as *find* the useful intermediate terms, it makes little difference whether a bottom-up or a top-down approach is used. We will revisit this idea in section 6.1.6.

4.3 Bottom-Up Graph-Based Algorithms

4.3.1 The BH and BHM Algorithms

In 1991, Bull and Horrocks [7] proposed the BH algorithm, a graph-based MCM heuristic that could be applied with: addition only, addition and subtraction, addition and shifts, and all three operators. We will describe all four variants of the BH algorithm in one generalized form, however note each will use a different definition of the adder-operation based on which operators are allowed. Also note that right shifts were not considered in [7]. The algorithm tries to minimize the number of additions or subtractions (if shifts are allowed, they incur no cost).

The BH algorithm constructs one target at a time in ascending order. With only addition, larger terms can be built from smaller terms but not the other way around. Let us consider how to construct the current target t. The BH algorithm pre-dates the notion of adder distance, so instead, it tries to minimize the *difference* between t and the closest element in R to t. The error ε is defined in (4.1) and it represents the smallest term that we would like to have so that t can be constructed with one more adder. The error will decrease after each iteration until it eventually becomes zero.

$$\varepsilon = \min_{r \in R} \quad t - (r \ll n) \quad \text{subject to } n \geq 0,\ n \in \mathbb{Z},\ \text{and } t > (r \ll n) \tag{4.1}$$

If shifts are not allowed, then simply constrain $n = 0$ in (4.1). The error ε is always positive, so intermediate terms must be less than t. Let r' denote the $(r \ll n)$ that minimized (4.1), i.e. $r' = t - \varepsilon$. If $\varepsilon \in R$, the algorithm constructs t with one more adder and we move on to the next target (this ends the current iteration). Otherwise, two terms will be constructed on this iteration according to (4.2) and (4.3).

$$s = \arg\min_{s \in S} \quad \varepsilon - s \quad \text{subject to } \varepsilon \geq s \tag{4.2}$$

$$w = r' + s \tag{4.3}$$

In (4.2) and (4.3), s gets as close as possible to the error ε, and then by adding it to r' (which is closest existing term to t), w now becomes the closest term to t and the new error will be smaller. For the SCM and MCM problems, we are only concerned with the variant of BH that uses addition, subtraction, and shifts.

In 1994, Dempster and Macleod [1] proposed the BHM (Bull Horrocks Modified) algorithm. It still uses the same fundamental error-minimization approach as BH, however some weakness in BH were remedied. In order to better utilize subtraction, the error (4.1) was now allowed to be negative and all terms were constrained to be no larger than $2 \cdot \max(T)$, where $\max(T)$ denotes the largest target. Since shifts incur no cost, we can constrain all terms to be odd integers without loss of generality, as explained in section 1.2.2, yet this was not exploited in the BH algorithm (instead, powers of 2 of each new term were also added to R). This issue is resolved in BHM, along with the ability to use right shifts. Finally, targets are now constructed in the order of increasing SCM cost, which could be evaluated using the MAG algorithm (optimal SCM, section 4.1.1) or could be estimated by, for example, the CSD cost. Although not stated in [1], our interpretation is that there are generally fewer ways to construct low cost terms than high cost terms simply since there are fewer ways to arrange operands and operators given fewer adders. With more ways of being constructed, the high cost terms are more likely to benefit from reusing existing terms (they are more likely to have a smaller remaining adder distance than the SCM cost).

In summary, for the current target t, BHM uses equations (4.4), (4.5), and (4.6) in place of (4.1), (4.2), and (4.3), respectively.

$$\varepsilon = \min_{r \in R} \quad |t - (r \ll n)| \quad \text{subject to } n \geq 0 \text{ and } n \in \mathbb{Z} \tag{4.4}$$

$$\{s, m\} = \underset{s \in S,\, m \in \mathbb{Z},\, m \geq 0}{\arg\min} \quad |\varepsilon - (s \ll m)| \tag{4.5}$$

$$w = r' \pm (s \ll m) \tag{4.6}$$

A significant improvement was obtained by fine-tuning the BHM heuristic to better match the SCM and MCM problems, yet BHM still has no notion of adder distance. Even so, many later algorithms (such as RAG-n and Hcub in sections 4.3.2 and 4.3.3, respectively) reuse this iterative graph construction methodology with the notion of reducing some kind of error in order to get closer to the target. RAG-n and Hcub use the adder distance as this error metric in order to obtain better solutions.

4.3.2 The RAG-n Algorithm

In 1995, Dempster and Macleod [14] proposed the n-dimensional reduced adder graph (RAG-n) algorithm. To the best of our knowledge, this was the first algorithm to introduce the idea of adder distance and the successor set, although these concepts were not formalized until 2007 by Voronenko and Püschel in [6]. Another extremely important contribution from [14] is the separation of the iterative graph construction process into an optimal part and a heuristic part. Because shifts are free, we can assume that all targets given in the MCM problem are not shifted versions of each other. Therefore, at least one adder is needed to construct each target. If any $t \in T'$ satisfies $t \in S$, then RAG-n immediately constructs t on this iteration (this ends the iteration). This is known as the *optimal part* of RAG-n since at least one adder for each target will be needed at some point anyways. The advantage of using one adder

now (as opposed to later) to construct t is that other terms can then be built off the newly constructed target. Unlike BHM, targets are not constructed in a pre-defined order in RAG-n. If there is no $t \in S$ (i.e. no target is distance 1), then the *heuristic part* of RAG-n is used on this iteration.

If R is a valid solution to the constant multiplication problem, then $T \subseteq R$, thus $|R| \geq |T|$. In other words, the best we can do is to construct all of the targets without any intermediate terms. This happens if RAG-n is able to construct all of the targets by only using its optimal part. Thus, in some cases, a heuristic algorithm can provide a solution that is known to be optimal. Because of this *global optimality*, almost all of the algorithms created after RAG-n reuse the optimal part of RAG-n, thus these algorithms can also be divided into an optimal and heuristic part. In the general case (*except* for the optimal part), there is no proof that taking the steepest descent on each iteration leads to global optimality (a solution with the absolute minimum number of adders). To the best of our knowledge, the only way to *guarantee* global optimality *in general* is to use an exhaustive search.

Let us now describe the heuristic part of RAG-n. First we check if any target t can be constructed with 2 adders. Note an exact adder distance 2 test is *not* used. Rather, two heuristic tests are performed:

1. for each target t and each $r \in R$, check if $|t - r| \in C_1$,
2. for each target t and each $s \in S$, check if $|t - s| \in C_0$ (i.e. $|t - s|$ has an adder cost of zero and thus is some integer power of 2).

As soon as one of these tests succeed, we will know which target t is distance 2 and which intermediate term will enable t to be constructed with 2 adders (this intermediate term will be an element of C_1 or S if test 1 or test 2 succeeds, respectively). As soon as

a target is found to be distance 2, it is constructed along with its useful intermediate term, which ends the current iteration. Otherwise, if none of the tests succeed for any target, then the target with the smallest optimal SCM cost is constructed (every term in the SCM solution is constructed). This ends the current iteration. The optimal SCM solutions are provided by the MAG algorithm [1].

BHM constructs one target at a time in the order of increasing SCM cost whereas RAG-n tries to construct the closest target in terms of adder distance. As R increases in size (as more terms get constructed), it is expected that the SCM cost becomes a less accurate estimate of the remaining adder distance because of the ability to build off of the existing terms (this is a weakness in BHM). However, one weakness that both of these algorithm share is the inability to choose intermediate terms that *jointly benefit* all of the remaining targets (when the optimal part of RAG-n is not used).

4.3.3 The Hcub Algorithm

Voronenko and Püschel proposed Hcub (short for Heuristic of Cumulative Benefit) [6] 12 years after RAG-n, in which time the amount of computation power in desktop computers had increased by orders of magnitudes. Thus it is feasible for Hcub to use more computation in order to achieve a more precise heuristic. With the optimal part of RAG-n already identified, the challenge is now to design good heuristics. When intermediate terms are needed, Hcub attempts to maximize the *joint benefit* that it provides to *all* of the remaining targets. For example, consider a MCM instance with $T = \{23, 81\}$. Both targets are distance 2 from $R = \{1\}$. RAG-n could choose to construct $23x$ as $23x = (3x \ll 3) - x$ where $3x = (x \ll 1) + x$. In this case, 81 is still distance 2 (from $R = \{1, 3, 23\}$). However, by considering

intermediate terms that are useful to both 23 and 81, we can instead construct $23x$ as $23x = (x \ll 5) - (9x)$ where $9x = (x \ll 3) + x$. Now only one more adder is needed to construct $81x = ((9x) \ll 3) + (9x)$.

On each iteration, Hcub spends one adder to construct one term, thus this term must be in the successor set S. Although Hcub reuses the optimal part of RAG-n, it is done a more computationally efficient manner by *incrementally* updating S (which was not done in RAG-n). Let r' denote the newly constructed term on this iteration, then $R_{new} = R_{old} \cup \{r'\}$. By definition, $S = \mathcal{A}(R, R) \backslash R$, thus we need to consider the adder-operation of all possible pairs of elements in R. If we add r' to R, the only *new* pairings will be r' with itself and r' with each element in R_{old}. Thus the new elements to add to the successor set are $S_{update} = \mathcal{A}(r', r') \cup \mathcal{A}(r', R_{old}) = (C_1 \cdot r') \cup \mathcal{A}(r', R_{old})$. To ensure R and S are mutually exclusive sets, it follows that $S_{new} = (S_{old} \cup S_{update}) \backslash R_{new}$.

Let us consider one iteration of the heuristic part of Hcub, which is used when $t \notin S$ for all t. As defined in [6], Hcub selects a successor s for construction according to (4.7). Also defined in [6] is the weighted benefit function, as shown below in (4.8). Note that we have changed the notation from [6]. The weighted benefit function in [6] was denoted with $\overline{\mathrm{B}}$, but we do not want to confuse this with our CSE notation.

$$\mathrm{H}_{\mathrm{cub}}(R, S, T) = \arg\max_{s \in S} \left(\sum_{t \in T'} \hat{\mathrm{B}}(R, s, t) \right) \tag{4.7}$$

$$\hat{\mathrm{B}}(R, s, t) = 10^{-\mathrm{dist}(R \cup \{s\}, t)} \left(\mathrm{dist}(R, t) - \mathrm{dist}(R \cup \{s\}, t) \right) \tag{4.8}$$

Recall from section 3.2.3 that $\mathrm{dist}(R, t)$ denotes the adder distance from the existing terms in R to the target t. For each target t and each successor s, (4.8) is a *quantitative measure* of how ‘‘useful’’ s is in terms of helping to construct t. Note that there is

more than one way to measure this ''usefulness'', (4.8) is just Hcub's chosen metric. Summing this usefulness over all $t \in T'$ allows us to measure how much *joint benefit* the successor s provides *over all of the remaining targets*. If s is useful, we would expect the remaining adder distance to decrease if s were constructed. In other words, if s is useful, $\text{dist}(R \cup \{s\}, t) < \text{dist}(R, t)$. Closer targets are given more benefit because we can use targets to build other terms (i.e. using the target(s) as operand(s) in an adder-operation) and thus it is generally more beneficial to construct targets sooner than later. This is facilitated by the exponent in (4.8).

The adder distance must be computed in order to evaluate (4.7) and (4.8). It is computationally expensive to compute the adder distance exactly for large distances, so Hcub computes the adder distance exactly only up to distance 3. Beyond this, the distance is estimated. Exact distance 2 tests (to establish whether or not a target is distance 2) were discussed in section 3.2.6. Exact distance 3 tests and the tests for distance estimation are discussed in sections 6.1.3.1 and 6.1.3.2, respectively.

4.4 Top-Down Graph-Based Algorithms

4.4.1 Bernstein's Software-Oriented SCM Algorithm and the BBB Algorithm

The constant multiplication problem first emerged in the 1970s for implementing constant multiplication *in software*. Many microprocessors at the time (such as Intel's 8008) did not have multipliers, so multiplication had to be done with additions, subtractions, and shifts. Even when the multiply instruction first became available in software, it would typically take more clock cycles to execute than an addition,

subtraction, or shift, thus solving the constant multiplication problem could lead to a reduction in execution time. Today, with high throughput pipelined multipliers and out of order execution, solving the constant multiplication problem provides no benefit in software. Algorithms for solving constant multiplication in software can be applied to the SCM and MCM problems (for custom hardware) by setting the cost of shifts to zero. Addition and subtraction typically have the same execution time in software and require approximately the same amount of logic resources in hardware.

In 1986, Bernstein [22] proposed a SCM algorithm. It can be described by the recursive formulas in (4.9). Note that every input argument x to the function $Cost(x)$ must be an odd integer. If the SCM target t is even, then we must incur an extra cost of $ShiftCost(w)$, where $t/2^w$ is an odd integer (and Bernstein's algorithm would be applied to $t/2^w$). In (4.9), a, b, c, and d are integers, $c \geq 1$, and $d \geq 1$.

$$
\begin{aligned}
&Cost(1) = 0 \\
&Cost(t) = 1 + \min \begin{cases}
Cost(\,(t-1)\,/\,2^a\,) + AddCost + ShiftCost(a) \\
Cost(\,(t+1)\,/\,2^b\,) + SubtractCost + ShiftCost(b) \\
Cost(\,t\,/\,(2^c-1)\,) + SubtractCost + ShiftCost(c) \\
Cost(\,t\,/\,(2^d+1)\,) + AddCost + ShiftCost(d)
\end{cases}
\end{aligned}
\tag{4.9}
$$

The cost of the shifts in (4.9) are a function of how many bits the operand was shifted. In microprocessors that only support single bit shifts, $ShiftCost(x)$ is proportional to x. For microprocessors that support shifting by an arbitrary number of bits, $ShiftCost(x)$ is a constant. In the SCM problem, $ShiftCost(x) = 0$. Although (4.9) is expressed in a depth-first manner, the search is computed breadth first since we are interested in finding the minimum cost of t.

Bernstein's algorithm is a branch and prune heuristic. The pruning arises from only allowing certain values of a, b, c and d in (4.9). Disregarding $Cost(1) = 0$, the first two branches of (4.9) are additive decompositions whereas the last two branches are multiplicative decompositions (note that Bernstein's algorithm predates these concepts). The first two branches realize $t \cdot x$ as $t \cdot x = ((u \cdot x) \ll a) + x$ and $t \cdot x = ((u \cdot x) \ll b) - x$, respectively. Since t is odd, $t \pm 1$ is even. Every input argument of $Cost(\cdot)$ must be an odd integer, so there are *unique* values for a and b such that $(t-1)/2^a$ and $(t+1)/2^b$) are odd integers. In the last two branches of (4.9), t is realized by multiplying some term by C_1 (although Bernstein's algorithm predates the notion of the C_n sets). Recall from section 3.2.4 that every element in C_1 has the form $2^n \pm 1$ where $n \geq 1$ and $n \in \mathbb{Z}$. The set $\frac{t}{C_1}$ may contain several elements, so several values of c or d could be used in the last two branches. Since only two specific additive decompositions are considered per recursion, Bernstein's algorithm typically produces solutions that are mostly multiplicative.

It was observed in [23] that Bernstein's algorithm favors multiplicative decompositions whereas BHM favors additive decompositions (the error minimization strategy in BHM is purely additive). In [23], the BBB algorithm (better of Bernstein or BHM) was proposed. For a SCM problem, it runs both of these algorithm independently and then simply selects the better result. BBB is an example of a hybrid algorithm. As shown in [23], usually one of the two methods will be better than the other for one constant, but for a different constant the other method may be better, thus much better solutions are obtained on average (over several problem instances). Our interpretation is that by choosing heuristics that can typically search the solution space with *little overlap*, much more of the *total* solution space can be explored.

4.4.2 Difference-Based Heuristics and the DiffAG Algorithm

Difference-based MCM algorithms typically try to build targets off of other targets, i.e. $t_i \in \mathcal{A}(t_j, t_k)$. When intermediate terms are needed, they are selected from some kind of difference set (which contain shifted differences of the targets, this will be explained in more detail below). In [24--26], a minimum spanning tree problem is solved on each iteration to determine which dependencies *between* targets are the most useful (i.e. if target t_i can be built off of t_m or t_n, we must decide which is the better option). These works introduced the idea of recursively searching for the best differential term between targets, hence the use of iterations. In [27], the difference problem is considered only once (no recursion). A weighted minimum set cover problem is solved to determine the best differences and then a CSE algorithm is used to implement the multiplication by the necessary differential terms.

In 2007, Gustafsson [28] proposed the DiffAG algorithm, which is short for the Difference-based Adder Graph heuristic. DiffAG combines the optimal part of RAG-n with the idea of recursively searching for the best differential term. When intermediate (non-output) terms are needed, DiffAG tries to create supporting intermediate terms *between* targets. The idea is to *first* construct intermediate terms that will *later* allow many of the targets to be constructed by the optimal part of the algorithm. However, the useful intermediate terms may be far away (in terms of adder distance from the existing terms in R). Instead of constructing a term at the expense of several adders, DiffAG *adds another target* with the value of the desired intermediate term. This facilitates a *recursive* search for useful intermediate terms. Like other top-down approaches, these terms will get progressively closer to R over several iterations, as explained in section 4.2.

In [28], Gustafsson explains that given two targets t_i and t_j, eventually both targets will be constructed (in order to satisfy the $T \subseteq R$ constraint in the SCM and MCM problems), so we should pick an intermediate term w such that once t_i is constructed, we can construct t_j using w and only one adder. In other words, t_j can be constructed optimally if $t_i \in R$ and $w \in R$. Formally, this means $t_j \in \mathcal{A}(t_i, w)$. From Theorem 1 in section 3.2.1, this is satisfied iff $t_i \in \mathcal{A}(t_j, w)$. Notice that if *either* target is constructed, the other remaining target can be constructed with w and one adder. It also follows that the set of useful intermediate terms between t_i and t_j is $W = \mathcal{A}(t_i, t_j)$. This is the underlying intuition of DiffAG, which favors the creation of terms that are *shifted differences between the targets*. Although not mentioned in [28], this will typically lead to decompositions that are mostly *additive* (also notice that DiffAG does *not* consider $t_i \in C_1 \cdot t_j$, perhaps due to the *unidirectional* relationship).

We will now summarize DiffAG. Assume the current iteration uses the heuristic part of DiffAG (i.e. $t \notin S$ for all t). All of the elements in R are placed in the node N_0. Each remaining target $t_i \in T'$ (indexing starts at $i = 1$) is placed in a node N_i (one target per node). Each node contains a set that is initialized with a cardinality of 1, except for N_0. Nodes have the property that once any element $n_i \in N_i$ is constructed, then all of the elements in N_i can be constructed. Obviously this property is satisfied if N_i contains one element. For two or more elements, we require that $n_i \in \mathcal{A}(R, n_i')$ for all pairs of elements $n_i, n_i' \in N_i$. In other words, there must exist an element $r \in R$ such that $n_i \in \mathcal{A}(r, n_i')$. From Theorem 1 in section 3.2.1, this is satisfied iff $r \in \mathcal{A}(n_i, n_i')$ or equivalently $\mathcal{A}(n_i, n_i') \cap R \neq \emptyset$. Let us define the difference set $D_{i,j} = \mathcal{A}(N_i, N_j)$ for a pair of nodes N_i, N_j where $i \neq j$. Because the adder-operation is symmetric, we can enforce $0 \leq i < j$ without loss of generality. It follows that if

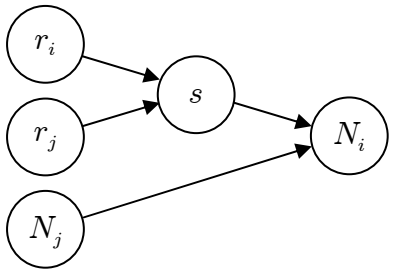


Figure 4.5: The topology corresponding to $D_{i,j} \cap S \neq \emptyset$, where $D_{i,j} = \mathcal{A}(N_i, N_j)$.

$D_{i,j} \cap R \neq \emptyset$, then we can merge nodes N_i and N_j because the ability to construct all targets in the node once any target is constructed will be maintained. If nodes are merged, we will need to update the indexing and the difference sets. The merging of nodes is done exhaustively. Let $\mathcal{D}$ denote the union of the $D_{i,j}$ over all indexes $0 \leq i < j$, then we have finished merging nodes when $\mathcal{D} \cap R = \emptyset$.

Now that the remaining targets have been classified into nodes, let us search for useful intermediate terms. $\mathcal{D}$ represents the set of all useful intermediate terms between any pair of nodes, thus the terms in $\mathcal{D}$ are close to the targets in adder distance. Among the terms in $\mathcal{D}$, we should choose one that is closest to R (to try to minimize the number of adders needed in the MCM problem). Since $\mathcal{D} \cap R = \emptyset$, the next best case is to find a $d \in \mathcal{D}$ such that $d \in S$. For each successor $s \in S$, we count how many times s occurs in each of the $D_{i,j}$ sets. If the largest count is not zero ($\mathcal{D} \cap S \neq \emptyset$), then DiffAG constructs the s with the largest count on this iteration (which ends the current iteration). Otherwise, none of the differential terms in $\mathcal{D}$ are distance 1. In this case, a new target will be created instead of constructing a successor. Let the set P be comprised of the elements in $\mathcal{D}$ that have the minimum CSD cost (the CSD cost is used to *estimate* the adder distance from R). For each $p \in P$, we count how many times p occurs in each of the $D_{i,j}$ sets. Assume p' has the

highest count. On this iteration, DiffAG will construct a new target p', i.e. p' is added to T' and no update is made to R or S. This ends the current iteration.

4.5 CSE-Based Algorithms

We will discuss both bottom-up and top-down CSE algorithms in this section, as both approaches have a similar underlying intuition.

4.5.1 An Introduction to CSE Algorithms

In 1991, Hartley [29] introduced common subexpression elimination (CSE) as means to exploit the redundancy in a set of constants. The CSE representation facilitates the use of a pattern searching algorithm in order to determine which patterns are ‘‘useful’’. As illustrated in detail in section 3.1.2, patterns are a collection of nonzero digits which represent how two or more existing terms have been added to create a new term. A pattern needs to appear in the CSE representation of a constant in order for it to be substituted. This can be used to help CSE-based heuristics prune the search space. For example, the CSD form of 45, which is $A0\overline{A}0\overline{A}0A$, does not contain $A00\overline{A} = 7 \times A$ (or its negative $\overline{A}00A$), so a heuristic could assume that $7 \times A$ is not a useful intermediate term. Furthermore, $A0\overline{A} = 3 \times A$ or its negative occurs twice whereas $A0A = 5 \times A$ or its negative occurs only once, thus the heuristic could assume $3 \times A$ is a *more useful* intermediate term than $5 \times A$.

Recall from section 3.2.3 that the number of nonzero digits can be used to estimate the adder distance. Assume the target t_i has n_i nonzero digits, then $n_i - 1$ adders are needed to construct t_i, so the total remaining adder distance is $\sum_i (n_i - 1)$. A pattern

can occur in multiple places in each target; we are interested in the *total* number of places that it occurs in all of the targets. A new pattern with m nonzero digits can be constructed with $m-1$ adders, where $m \geq 2$. Every time this new pattern is substituted somewhere in the CSE form of any target, m existing nonzero digits are replaced by one new nonzero digit. If this pattern can be substituted k times, the remaining adder distance will decrease by $k \cdot (m-1)$, but we have also used $m-1$ adders to construct the pattern, thus the *net savings* is $(k-1) \cdot (m-1)$ adders. Clearly, this should be maximized. Only $k \geq 2$ is beneficial (if $k = 1$, the pattern simply represents how the remaining terms are added together).

4.5.2 Top-Down Versus Bottom-Up Heuristics

CSE-based algorithms iteratively find and replace patterns in order to reduce the number of adders. Most of the CSE-based algorithms take a steepest descent approach on each iteration. However, there is no proof this leads to the global optimum. Most CSE algorithms use one of the two following strategies. On each iteration, one strategy is to find and substitute the pattern with 2 nonzero digits that occurs the most; the other strategy is to find and substitute the pattern that has the most nonzero digits subject to this pattern occurring at least twice in all of the targets.

The first strategy uses a bottom-up approach. With only 2 nonzero digits, each newly created pattern must be in the successor set, so over several iterations we will build up to the targets (we consider the creation and substitution of one pattern as one iteration). Hartley [30] presents an algorithm using this strategy. Since patterns have $m = 2$ nonzero digits, choosing the pattern that occurs the most (maximum k) maximizes the net savings in adders (we will save $(k-1) \cdot (m-1) = k-1$ adders).

The second strategy uses a top-down approach. Pasko [31] and Lefèvre [32] propose algorithms with this strategy. Once the pattern is constructed, some targets should be constructible with only a few adders. Several adders may be needed to construct the pattern, but we can search for patterns within this pattern to reduce the number of adders. Applied recursively, intermediate terms are first created close to the targets (in terms of adder distance) and over several iterations come closer to the existing terms in R, hence we build *down from* the targets.

Let us illustrate these two strategies with an example. Consider the CSD form of $2451 \times A$, which is $A0A0\overline{A}00A0A0\overline{A}$. Using a bottom-up approach, we would first substitute $B = A000000A = 129 \times A$ get $0000000B0B0\overline{B}$. Since no new pattern occurs at least twice, we can now collect the terms, thus $2451 \times A = (B \ll 4) + (B \ll 2) - B = 19 \times B$. Alternatively, using a top-down approach, we would first construct $C = A0A0\overline{A} = 19 \times A$, which costs 2 adders. This could be substituted in the CSD form to give $0000C000000C$, thus $2451 \times A = (C \ll 8) + C = 129 \times C$.

Hartley indicated in [29] that substituting a pattern with m nonzero digit n times is equivalent to substituting a pattern with n nonzero digits m times (for patterns *within one constant only*). Notice that we end up decomposing $2451 \times A$ as $129 \times 19 \times A$ regardless of whether we build the pattern B or we instead build the pattern corresponding to the collection of the B terms (which is actually the pattern $C = A0A0\overline{A}$, the B terms were collected as $B0B0\overline{B}$).

Our interpretation of this is that $2451 \times A$ can be decomposed in two ways since it is a *multiplicative* decomposition (as discussed in section 3.2.4). The construction of B corresponds to a subgraph that implements multiplication by an element in C_1 whereas C corresponds to a subgraph that implements multiplication by an element

in $\mathcal{A}(1, C_1)$. Subgraph C is an additive decomposition, so vertex reduction can be used, as shown in section 3.2.5. Two adders are needed to construct C, but notice that we did not define the intermediate term that would have been created after only one of the adders had been used.

As illustrated in section 3.2.5, an algorithm that allows patterns with more than 2 nonzero digits can take advantage of vertex reduction. To the best of our knowledge, none of the algorithms that use this strategy refer to notion of vertex reduction, the algorithms are only described as a top-down approach. In the MCM problem, more computational effort is required to find the pattern with the most nonzero digits compared to finding the pattern with 2 nonzero digits that occurs the most. Conclusively, there is a tradeoff between the two strategies in terms of the number of iterations versus the computational complexity of each iteration. In both strategies, typically an exhaustive search is used to find the best pattern on each iteration.

There is also the choice of which starting CSE representation should be used for the targets. The earliest CSE algorithms used the CSD form, but it was later found that other starting forms could produce better solutions, as discussed in section 4.5.3.1. There are corner cases which many algorithms do not discuss. For example, [30--32] do not specify which pattern should be substituted if there are several ‘‘best’’ patterns. These issues are discussed in relation to our proposed algorithm in section 5.1.5.

4.5.3 The H(k) Algorithm

4.5.3.1 Using SD Forms with Extra Nonzero Digits

Prior to our work, H(k) [33] was the best existing SCM heuristic. The optimal SCM algorithm in [2] can only find solutions with up to 5 adders, which limits it 19 bits

(there is a 20 bit constant that requires 6 adders). Thus for constants represented on larger bit widths, a heuristic is needed.

As explained in section 3.1.2, a pattern needs to exist in the CSE representation of the constant in order to be substituted. Recall the example from section 3.1.2 in which $15 \times A$ could be substituted in the CSD form of $45 \times A$ whereas $15 \times A$ could not be substituted in the binary form of $45 \times A$. This suggests that better solutions could be found if we consider different starting representations.

The remaining adder distance can be estimated by the number of nonzero digits in a CSE algorithm, as explained in section 3.2.3. CSE algorithms typically take a steepest descent approach on each iteration. If we consider the subproblem of choosing the starting CSE representation as one iteration, taking the steepest descent corresponds to choosing the representation with the minimum number of nonzero digits. This is why many CSE algorithms start with the CSD form of the constant.

Park and Kang [34] proposed an algorithm to find *all* of the representations with the minimum number of nonzero digits. These were named the minimum signed digit (MSD) representations, since ''CSD'' was already reserved for the representation with no adjacent nonzero digits. Park and Kang then applied a CSE algorithm to each MSD form. This produces better solutions compared to only considering the CSD form, as shown by the results of [34]. The H(k) algorithms takes this a step further. H(k) considers all SD forms of the constant with *up to* k extra nonzero digits (k more digits than the CSD form). The Hartley algorithm [30] is used to search and replace patterns in each of these SD forms (each SD form is considered independently from the other SD forms). The H(k) algorithms selects the best solution that was found by the Hartley algorithm among any of the SD forms considered.

H(0) is analogous to Park and Kang's algorithm. Park and Kang improved over existing methods by considering multiple ''best'' solutions at the stage before pattern searching begins (when choosing the starting representation). H(k) further improves the solutions by not being forced to take the steepest descent in this starting stage, although both Park and Kang and H(k) take the steepest descent on all subsequent iterations. In section 5.1.1, several detailed examples are provided to illustrate the benefits and the remaining limitations of considering more initial SD forms.

4.5.3.2 The Generation of SD Representations

In [35], the algorithm for creating all of the SD representations of the constant with up to k extra digits is presented. We will provide a summary, the reader is referred to [35] for details. An exhaustive branch and bound approach is used. We initialize a ''remainder'' r to the value of the target and the corresponding SD form is initially empty. Let $n(r)$ denote the degree of evenness of r (this is the largest integer n such that $2^{-n}r$ is an integer). From each current r and its associated partially constructed SD form, we create two new r (along with their new associated SD forms). One instance will have a new remainder of $r - n(r)$ and the new SD form will be the old SD form with the digit $1 \ll n(r)$ added to it. The other instance will have a new remainder of $r + n(r)$ and the new SD form will be the old SD form with the digit $\overline{1} \ll n(r)$ added to it. At any time, if we add the remainder and the number that the associated SD form represents, we must get the value of the target (every time $1 \ll n(r)$ is added to the SD form, we subtract $2^{n(r)}$ from the remainder). Clearly, whenever the remainder reaches zero, the associated SD form is a valid representation of the target (at this point we can launch the Hartley algorithm).

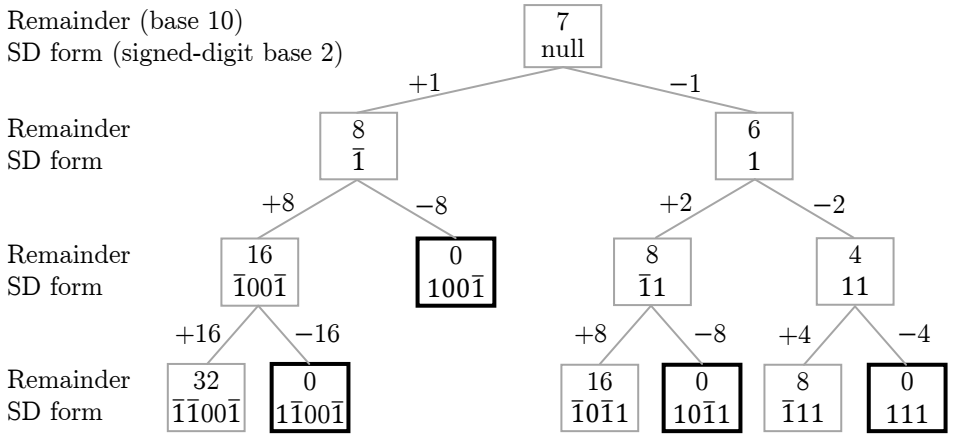


Figure 4.6: The signed-digit representation generating algorithm applied to the constant 7 with $k = 1$ extra nonzero digit. The branch values denote the evenness of the parent's remainder. We have highlighted all of the valid SD forms.

This process can be regarded as a construction of a tree, where the two new instances are the children. If a node is at depth d in the tree (the root of the tree is $d = 0$), then the SD form in this node will have d nonzero digits. Given that the CSD cost of the target is z, we must stop the tree construction at a depth of $z + k$ (k is the number of extra nonzero digits and is specified by the user). An example for the constant 7 and with $k = 1$ is shown in Figure 4.6.

4.6 Other MCM Algorithms

In this section, we will *briefly* discuss a few MCM algorithms that are somewhat related to this thesis. Note that we may use concepts that predate the algorithm we are describing (simply to keep the discussion concise).

Aksoy [36] proposed the idea of removing unnecessary intermediate terms *after* a valid solution to a SCM or MCM problem has been found. Basically, for each

intermediate term $r \in R$ where $r \notin T$, we temporarily remove r from R, then we check whether $R \backslash \{r\}$ is still a valid solution to the SCM or MCM problem. This may involve reordering the elements in $R \backslash \{r\}$ so that we can still satisfy the $r_k \in \mathcal{A}(r_i, r_j)$ constraints for each k, where $0 \leq i, j < k$ (from Definition 3 in section 2.3.2). We will not discuss the algorithm in [36] because its heuristic is incomplete. For targets distance 3 or higher, the heuristic in [36] cannot determine which successor(s) are useful. Thus if no targets are distance 2, the heuristic will make an arbitrary choice, effectively taking a shot in the dark. Even so, the post-removal of unnecessary intermediate terms can be applied to *any* algorithm, and we will use it to enhance some of the algorithms proposed in this thesis.

As discussed in sections 3.1.2, 4.5.3.1 and 5.1.1.1, the patterns that can be found and substituted by a CSE algorithm depend on the starting CSE representation of each target. In [37], a CSE-based MCM algorithm is proposed in which addition and subtraction is used to generate new patterns that otherwise would not be found by merging signed digits, as is done in CSE. Only the CSD form is considered in [37]. For example, $49 \times A = A0\overline{A}000A$ does not contain any instances of $A0A = 5 \times A$ or $A00A = 9 \times A$ or their negatives, yet we have $49x = ((5x) \ll 3) + (9x)$. However, [37] cannot apply this technique *within* a constant, it can only do it *between* constants. We will discuss the details of how CSE patterns go missing and how to recover the simplest cases in sections 5.1.1 and 5.1.2, respectively. There was no analysis of the underlying problem with CSE in [37], they only mention that using addition and subtraction can lead to better solutions. In sections 5.2 and 6.1, we will revisit the idea of combining the CSE and DAG frameworks.

4.7 Problems Related to SCM and MCM

We will *briefly* describe some of the related problems to SCM and MCM. These are variants of the SCM or MCM problem in which a secondary objective (or constraint) has been added and/or a slightly different minimization metric is used.

4.7.1 Depth Constraining

One of the common secondary problems is to enforce a depth constraint in the solution to a SCM or MCM problem. The adder depth is defined as the maximum number of adders that we pass through along any path from the input to any of the outputs in the constant multiplication logic circuit. The adder depth is an estimate of the longest path through the logic circuit, which is known as the *critical path*. Because of the physical construction, logic gates have a propagation delay, which is the length of time between when stable inputs are asserted to when all of the outputs become stable. As more logic gates are placed serially between the input and the output, the critical path becomes longer and the logic circuit must be clocked at a slower speed, which results in a lower computational throughput. The critical path is also a function of other things like the delay of each gate and the transit time along wires, however we will make abstraction of this in order to solve real-sized SCM and MCM problems within reasonable amounts of time. Most (if not all) of the work in this area of research uses the adder depth to estimate the length of the critical path.

Given the same problem instance, the number of adders increases as the depth constraint is made smaller. A solution may not exist if the depth is overly constrained. The depth constraint can also be used to prune the search space, as explained in detail in section 6.2.1.

Examples of MCM algorithms that minimize the number of adders subject to a depth constraint include [20, 21, 37, 38]. Said algorithms use techniques that have already been described in this thesis, but they must also keep track of the depth of each term. In section 6.2, we will propose a depth constrained MCM algorithm and we will show in detail how the depth constraint is managed in our algorithm.

4.7.2 Minimization of Single-Bit Adders

For small problem sizes, it is feasible to use a more accurate (less abstracted) metric to estimate the amount of logic resources required to implement constant coefficient multiplication. As justified in section 1.2.2, the number of additions or subtractions is used as the metric in much of the work in this area of research. To minimize the logic required, it is assumed that ripple-carry adders are used. A ripple-carry adder is a set of single-bit adders connected serially. Notice that when we compute addition *by hand*, we only need to compute one sum digit and one carry at a time (a single-bit adder computes this in binary). Each carry is passed from right to left, hence the name ripple-carry adder.

For the same SCM or MCM problem, an algorithm can find several ''best'' solutions in terms of additions or subtraction. In this case, it would be beneficial to break the tie by considering a more accurate metric, such as the number of single-bit adders. In [1], this was mentioned as a feature that could be added to the MAG algorithm. In an iterative algorithm that uses adder-operations as the metric, on each iteration, there could be several ''best'' intermediate terms. This tie can be resolved by considering single-bit adders, as done in [39, 40]. The algorithm in [41] directly minimizes the number of single-bit adders without considering the number of adder-operations.

In section 5.3, we will propose an optimal SCM algorithm which first minimizes the number of adder-operations. Among the equally good solutions, the one with the minimum number of single-bit adders is selected. The minimum adder depth is used to resolve any remaining tie, but beyond this, the choice is arbitrary.

4.8 Bounds on the SCM and MCM Problems

4.8.1 Theoretical Analysis

In this section, we provide some theoretical analysis on the upper and lower bounds in the SCM and MCM problem. Let n denote the number of constants and assume the largest constant is representable on b bits (b denotes the bit width).

We first consider the SCM problem ($n = 1$). As shown in section 2.1.2, a constant of bit width b has a maximum of $\lceil \frac{b+1}{2} \rceil$ nonzero digits in the CSD form, thus the worst case number of adders is $\lceil \frac{b+1}{2} \rceil = O(b)$. To the best of our knowledge, the CSD upper bound is the tightest asymptotic bound known. The CSD form has the minimum number of nonzero digits. If the CSD form has m nonzero digits, at least $\lceil \log_2 m \rceil$ adders are needed. This can be shown using CSE. Given m nonzero digits, a pattern with 2 nonzero digits can only be substituted up to $\lfloor \frac{m}{2} \rfloor$ times. At best we can halve the number of nonzero digits using 1 adder. Applying this recursively, it follows that the number of adders is bounded below by $\lceil \log_2 csd(t) \rceil$, where $csd(t)$ denotes the number of nonzero digits in the CSD form of the target t.

We can also consider the depth constraint in SCM. Obviously the depth is no larger than the number of adders, i.e. an adder-tree cannot have 4 serial additions if there are only 3 adders, hence the depth is bounded above by the CSD cost, which

is $O(b)$. Given that the CSD form of a number has m digits, the depth is bounded below by $\lceil \log_2 m \rceil$. Following a similar argument as with the number of adders, every time the depth increases by 1, at most half of the nonzero digits can be substituted. Assume depth 1 is the level closest to the input and depth d is the output of the adder tree. Only 2 terms can be present at depth $d-1$, as these are added to produce the final answer. At depth $d-2$, there can be up to 4 terms to produce the 2 terms on depth $d-1$. Applying this recursively, we obtain a bound of $d \geq \lceil \log_2 m \rceil$.

Now consider the MCM problem ($n \geq 2$). Note that we can solve a MCM problem by solving a SCM problem for each target and then combine all of the terms from all of these SCM solutions into one ready set R. Thus, given n targets of bit width b, each target requires up to $\lceil \frac{b-1}{2} \rceil$ adders, thus the total number of adders is upper bounded by $n \cdot \lceil \frac{b-1}{2} \rceil$. Clearly, this is $O(nb)$. By sharing intermediate terms between constants, we may be able to reduce the number of adders in comparison to solving SCM for each target. At least one adder is needed per target since targets are not shifted versions of each other. This is encompassed by the optimal part of RAG-n. However, if no target is an element of C_1, then clearly at least one intermediate term is needed to construct any of the targets and thus to solve the MCM problem. It follows that the best we can do is to optimally construct one target (we should choose the target with the smallest optimal SCM cost) and then use one adder per remaining target. We need at least $\min_{t \in T} \lceil \log_2 csd(t) \rceil$ adders to construct the first target (from the SCM lower bounds) and $n-1$ adders for the remaining targets, hence the total number of adders is bounded below by $\min_{t \in T} \lceil \log_2 csd(t) \rceil + n - 1$.

In the MCM problem, the depth still cannot be larger than the number of adders. The depth can be minimized by not sharing any terms between targets (for each target,

we can independently construct an adder tree of minimum depth). Thus the depth is bounded below by the worst case SCM depth of any target. By sharing terms between targets, we may save adders but we cannot reduce the depth. Likewise, in order to satisfy a depth-constraint, some terms may not be sharable. This is one of reasons why a tradeoff exists between minimizing the number of adders and minimizing the depth.

All of the above derivations were provided in [42] although we have independently derived the above bounds. In [42], many other bounds are also provided, but these are not closely related to this thesis. In [43], it was proved that the lower bound for both the average and the worst case number of adders in SCM is on the order of $\frac{b}{logb}$.

4.8.2 Justification for Not Providing the Asymptotic Run Time Analysis of the Algorithms

We will provide only a minimal amount of asymptotic run time analysis. The worst case run time of our proposed heuristic SCM algorithm H(k)+ODP in section 5.1 is exponential with respect to the bit width of the constant. Since H(k)+ODP is reused in *all* of our heuristics (including MCM), it follows that *all* of our algorithms are of exponential order (the optimal algorithms are exhaustive searches, which is clearly exponential). However, in digital signal processing, it is uncommon to represent numbers with extremely large bit widths (such as hundreds bits) or to require simultaneous multiplication by thousands of constants (although one may need to solve *numerous* SCM or MCM problem instances *independently* in order to realize a DSP system). Thus, extremely large problem sizes have little or no practical significance. Because of this, it is acceptable to use a *heuristic* with exponential run

time provided that the *absolute* run time is reasonable. Furthermore, our algorithms are generally *faster* than the best existing methods.

In [6], a thorough run time analysis of BHM, RAG-n, and Hcub were provided. Many algorithms are able to traverse the CSD solution to construct each target if nothing better is found. In this case, the solution to the SCM or MCM problem will have up to $n \cdot \lceil \frac{b-1}{2} \rceil$ adders, where n is the number of constants and b is the bit width. If the algorithm is iterative, the number of iterations is bounded by $O(nb)$ since at least one term is constructed per iteration. Knowing the maximum size of R, we pre-allocate memory for it, for example. Examples of iterative algorithms that produce solutions no worse than CSD include: RAG-n, Hcub, DiffAG, and our proposed algorithms H3 and H4 (sections 6.1.3 and 6.1.4, respectively).

In all of the prior work, the shifts used in the adder-operation are bounded by $O(b)$ (usually a bound of $b+1$ is used, but $b+2$ is also considered). Thus for elements x and y, the size of the set $\mathcal{A}(x, y)$ is $O(b)$. It follows that the size of $\mathcal{A}(X, Y)$ is $O(b \cdot |X| \cdot |Y|)$, recall $|X|$ denotes the cardinality of X. For example, the size of the successor set $S = \mathcal{A}(R, R) \backslash R$ is $O(b \cdot nb \cdot nb) = O(n^2 b^3)$. Each element in the C_n set (for a given n) is constructed with n adder-operations with respect to $R = \{1\}$, so the size of C_n is $O(b^n)$. By applying this analysis, one could calculate the size and thus how much computation is needed to construct a set using adder-operations and/or divisions by C_n. For example, this is used for computing the exact adder distance, as described in section 3.2.6. Also, S is sorted so intersecting with S adds a factor of $O(\log(n^2 b^3)) = O(\log nb)$ in the exact distance tests. However, for the same reasons that we have justified the use of a heuristic with exponential run time, we will *not* provide the asymptotic run time for each distance test since it has little relevance.

Chapter 5

New SCM Algorithms

In this chapter, we will propose several new algorithms for the single constant multiplication problem. We begin with a heuristic approach using CSE. To the best of our knowledge, the H(k) algorithm [33] was the best SCM heuristic (prior to our work). In section 5.1, we modify H(k) so that better solutions can be produced in significantly less run time. An insightful analysis of the CSE framework is provided to expose the properties that we exploit. In section 5.2, we propose an optimal SCM algorithm that uses both the CSE and DAG frameworks. Unlike all of the existing optimal SCM algorithms which create a lookup table for all constants up to 2^b, we solve the SCM problem *for the given constant*. Our algorithm is exhaustive but uses very sharp pruning. The average run time is less than 10 seconds at 32 bits whereas the MAG algorithm [1] is limited to 19 bits. In section 5.3, we propose an algorithm which considers the number single-bit adders. Compared to the algorithm in section 5.2, better solutions are obtained at the expense of more run time. Finally, some concluding remarks are provided in section 5.4.

5.1 Heuristic SCM

Our contributions in this section have been published in [44], but we will provide additional analysis and an experimental evaluation between more existing algorithms in this thesis. We will begin by examining some cases in which CSE algorithms cannot find the optimal solution. In section 5.1.1, we illustrate the *CSE digit clashing* problem in detail. The simplest cases of this problem can be resolved by using overlapping digit patterns (ODPs), which we propose in section 5.1.2. Also in section 5.1.2, we explain how ODPs are incorporated into our proposed algorithm H(k)+ODP. This is a variant of the H(k) algorithm [33], the best existing SCM heuristic. We will discuss the remaining limitations of H(k)+ODP in section 5.1.3. In section 5.1.4, we justify the use of ODPs primarily to reduce the run time of H(k) instead of focusing on reducing the number of adders. Implementation details are briefly discussed in section 5.1.5. Experimental results are provided in section 5.1.6.

5.1.1 Examples of Non-Optimal CSE Solutions

5.1.1.1 Problems Due to the Initial SD Form

As shown in section 2.1.2, the CSD transform is a simple method to reduce the number of adders in constant multiplication. In section 2.2, we demonstrated that signed-digit (SD) forms could be factored to further reduce the number of adders. This factoring is facilitated by finding and replacing patterns in the CSE form of the constant, as illustrated in section 3.1.2. CSE provides a means to collect common terms *once we have decided which signed powers of two will be used to construct the constant*. A CSE algorithm may not find an optimal solution if the initial SD form

of the constant does not have the appropriate signed powers of two. For example, $5 \times A = A0A = (A \ll 2) + A$ only requires one adder, but it is impossible to find a solution with 1 adder by using $A\overline{A}0A$, as 1 adder can only add 2 nonzero digits.

As stated in section 4.5.3.1, Park and Kang [34] found better solutions by applying CSE to all MSD forms of the constant (SD forms of the constant with the minimum number of nonzero digits). Even better solutions are obtained with H(k), which applies CSE to all SD forms of the constant with up to k more digits than the CSD form. For example, the CSD form of $105 \times A$, $A0\overline{A}0A00A$, has no patterns that occur at least twice. Without being able to factor common terms, 3 adders are needed to add the 4 terms. However, one MSD form of $105 \times A$ is $A00\overline{AA}00A$, in which the pattern $B = A00\overline{A}$ can be substituted twice to yield $000B00\overline{B}$. In this case, 2 adders are used (one to create B and one to add the remaining terms). The problem was due to the CSD form being constrained to have no adjacent nonzero digits. Notice the leftmost nonzero digit of $000\overline{A}000A$ is adjacent to the rightmost nonzero digit of $A000\overline{A}000$, which obviously cannot be represented by the CSD form.

This first case in which H(0) produces a non-optimal solution is $363 \times A$. There are only two MSD forms, $A0\overline{A}00\overline{A}0\overline{A}0\overline{A}$ and $AA00\overline{A}0\overline{A}0\overline{A}$, neither of which have a common pattern, thus H(0) requires 4 adders. However, H(1) can find the optimal solution by using the representation $A0AA0A0AA$. The pattern $B = A0000A$ is substituted to produce $00000B0BB$, thus the total cost is 3 adders (1 to create B, 2 to add the remaining terms). Recall from section 3.2.3 that the adder distance can be estimated by the number of nonzero digits. If a pattern with 2 nonzero digits is substituted n times, a total of $2n$ old digits will be replaced with n new digits. This results in a savings of $n-1$ adders (the adder distance estimate has decreased by n but

we have used one adder to construct the pattern). Since the MSD forms of $363 \times A$ have no common patterns, it is impossible to reduce the initial estimate of the adder distance. Although $A0AA0A0AA$ has a *larger* initial estimate of the adder distance, this is more than offset by the *three instances* of $B = A0000A$, which indicate that we can reduce the adder cost to less than that of the MSD forms.

5.1.1.2 The CSE Digit Clashing Problem

In the $363 \times A$ example above, several digits need to be placed adjacently in order to observe the three instances of $B = A0000A$, but when these digits are collapsed into a SD form with fewer nonzero digits, the patterns become obfuscated. By considering SD forms with extra nonzero digits, we can recover patterns that need adjacent digits, however this does not allow us to recover patterns in which the digits *collide*.

As an example, the first case in which H(k) produces a non-optimal solution *for any* k is $805 \times A$. The optimal solution requires 3 adders. As indicated in section 4.8.1, if we start with more than 8 nonzero digits, then more than 3 adders will be needed. The CSD form of $805 \times A$ has 5 nonzero digits, thus we only need to consider all SD forms with up to 3 extra nonzero digits. H(3) cannot find the optimal solution, and this is sufficient proof that H(k) is not an optimal algorithm *for any* k.

In order to find the optimal solution for $805 \times A$, we will need to consider an unusual case. Notice that $((A0A) \ll 2) + A0A = AA00A$. This translates to $2^2(5 \times A) + (5 \times A) = 25 \times A$. The left digit of $A0A$ aligns with the right digit of $((A0A) \ll 2)$ to produce a zero in this position and a carry one position to the left. Thus, if $B = A0A$, we can substitute B in $AA00A$ to get $00B0B$ even though $A0A$ does not appear at either location of where B was substituted. This is an example of a class 1 overlapping digit

pattern, as formally defined in section 5.1.2.3. Now consider representing $805 \times A$ as $AA00A00A0A$. We substitute $B = A0A$ to get $00B0B0000B$, thus only 3 adders are needed (1 to make B, 2 to add the remaining terms).

This digit alignment problem was recognized in [12] and was identified as *clashing*. We thus refer to this problem as the *CSE digit clashing problem*. However, [12] presented clashing as a motivation to limit the value of k in the H(k) algorithm, as H(k) is not likely to produce better solutions by further increasing k beyond a certain point. Assume we know that a solution with n adders exists (which could be found with CSD, for example). The upper bounds from section 4.8.1 indicate that the SD form of the constant cannot have more than 2^n digits or else we are *guaranteed* to need at least $n + 1$ adders. It is argued in [12] that k should be limited well below this theoretical bound. More importantly, [12] does not even suggest that one should attempt to solve the clashing problem. We will propose a solution to the simplest cases of clashing in the next section. Based on our results (section 5.1.6), most of the obfuscated patterns can be recovered by resolving only the simplest cases.

All of the above examples involve a *multiplicative* decomposition ($105 \times A$, $363 \times A$ and $805 \times A$ were decomposed as $7 \times 15 \times A$, $11 \times 33 \times A$ and $5 \times 161 \times A$, respectively). Adding a pattern with a shifted version of itself is equivalent to multiplying it by C_1. When the CSE digit clashing problem arises in SCM, we have observed that many of the optimal solutions can be obtained by resolving a multiplicative decomposition (where n instances of the *same existing* pattern are added, $n \geq 2$) rather than an additive decomposition (where one instance each of n *different existing* patterns are added). This is because the solution to a SCM problem typically contains *only a few adders* (results are in section 5.1.6). This observation does *not* apply to MCM.

5.1.2 The H(k)+ODP Algorithm and Overlapping Digit Patterns

5.1.2.1 Limiting the Scope of the Problem

In general, in order to substitute a pattern with m nonzero digits n times, we need to have $m \cdot n$ nonzero digits. By considering clashing, we can substitute this in *less than* $m \cdot n$ nonzero digits. However, as justified in section 5.1.4, our goal is to significantly reduce the run time of H(k) without increasing the average number of adders. Thus we will only consider $m = 2$ and $n = 2$, which are the simplest cases of clashing. Furthermore, we will only consider two instances of the *same pattern*. Under this constraint, only one pair of nonzero digits can align. For example, in $((A0A) \ll 2) + A0A = AA00A$, the left digit of $A0A$ aligns with the right digit of $((A0A) \ll 2)$. Given that patterns have two nonzero digits, we cannot align two pairs of nonzero digits. This is impossible because it would require having two patterns with no relative shift between them. As emphasized at the end of section 3.1.2, *only one* digit may occupy each location in the CSE form (the $A0\overline{A}$ type of representation).

If we relax any of the above constraints, it becomes possible to align *more than one* pair of nonzero digits. This makes the CSE digit clashing problem much more difficult to solve and thus it would require more computation to solve. Several pairs of nonzero digits could align and thereby produce multiple carries (i.e. a carry produced the middle digit in $AA00A$ in the above example). Each carry may also now propagate to produce a nonzero digit several positions away. For example, consider decomposing $75 \times A$ as $5 \times 15 \times A = ((15 \times A) \ll 2) + (15 \times A)$. If we express $15 \times A$ as $AAAA$, notice *two* pairs of digits align in the middle of $((AAAA) \ll 2) + AAAA = A00A0AA$. In this case, one carry propagated all the way to most significant digit. To substitute

a pattern with 4 nonzero digits twice, we expect 8 digits, however $A00A0AA$ only has 4 digits. In the general case, the obfuscated patterns may be represented with an *arbitrary* number of digits less than that expected (we expect $m \cdot n$ digits).

In this thesis, overlapping digit patterns (ODPs) strictly refer to the non-standard CSE patterns that we will use to identify clashing cases for *only two* instances of a the *same* pattern, where this pattern has *exactly two* nonzero digits. In the earlier example, $AA00A$ is an ODP. We will only enumerate the cases for $m = 2$ and $n = 2$, however the *strategies* used to derive ODPs can be applied to arbitrary m and n.

An ODP cannot contain any instances of the pattern that was used to construct it (for example, $AA00A$ does not contain any instances of $A0A$). If an ODP did contain such an instance, we would instead substitute the regular pattern (with 2 nonzero digits) and not bother with the ODP. ODPs are not applicable to graphs since DAGs have no restrictions due to their representation. ODPs are also not applicable to CSE algorithms that select the pattern with the most nonzero digits, such as [31, 32], however this is of little relevance. Hartley indicated in [29] that substituting a pattern with p nonzero digit q times is equivalent to substituting a pattern with q nonzero digits p times. Our interpretation is that we will not be able to use vertex reduction, but Hartley's algorithm and H(k) do not use this anyways.

5.1.2.2 Integrating ODPs Into H(k)

Searching for non-standard patterns is unrelated to the generation of SD representations. Our proposed algorithm, H(k)+ODP, uses the same initial SD forms as H(k) and we also search each SD form independently and then select the best solution found at the end. ODPs are integrated into Hartley's search and replace algorithm,

which H(k) applies to each SD form. Although we are actually modifying Hartley's algorithm, it is futile to be restricted to only the CSD form of the constant like Hartley, as better solutions can be found by considering alternate SD forms. Furthermore, there is more potential for clashing as the density of nonzero digits increases, thus ODPs are more beneficial when large values of k are used in H(k).

5.1.2.3 The Three General Classes of ODPs

Let P represent a pattern in the form $(S \ll i) \pm S$, where i is a positive nonzero integer and S in some existing term (for example, S can be the input which we earlier labeled A or it can be an intermediate term). Note that P may also represent the negative of the pattern (which is obtained by inverting the sign of each digit). Two instances of P will have the form $(P \ll n) \pm P$. Usually this will produce 4 digits of S, in which there is no clashing, thus a CSE algorithm does not need ODPs to be able to find and substitute this. However, $(P \ll n) \pm P$ may produce three digits of S. These three digits make an ODP. In general, there are two main strategies for aligning two instances of P to produce an ODP:

1. Align the left digit of P and the right digit of $P \ll n$ so that the two aligned digits of S produce a zero at this position and a nonzero digit one position to the left due to a carry.

2. Position the left digit of P with respect to the right digit of $P \ll n$ so that these two digits produce $S\overline{S}$ or $\overline{S}S$, which are actually $0S$ and $0\overline{S}$, respectively.

Class 1 ODPs use the first strategy. The earlier example of $((A0A) \ll 2) + A0A = AA00A$ was a demonstration of a class 1 ODP in which $S = A$ and $P = (S \ll 2) + S = A0A$. Thus, if we *find* $AA00A$, we can *substitute* $B = A0A$ to get $00B0B$.

Table 5.1: The formal definition of the first three general ODP Classes.

(a) Class 1 ODPs

If $P = (S \ll i) + S$	Search for:	$\pm[\ (S \ll (2i)) + (S \ll (i+1)) + S\ ]$
	Replace it with:	$\pm[\ (P \ll i) + P\ ]$
If $P = (S \ll i) - S$	Search for:	$\pm[\ (S \ll (2i)) - (S \ll (i+1)) + S\ ]$
	Replace it with:	$\pm[\ (P \ll i) - P\ ]$

(b) Class 2 ODPs

If $P = (S \ll i) + S$	Search for:	$\pm[\ (S \ll (2i+1)) + (S \ll i) - S\ ]$
	Replace it with:	$\pm[\ (P \ll (i+1)) - P\ ]$
If $P = (S \ll i) - S$	Search for:	$\pm[\ (S \ll (2i+1)) - (S \ll i) - S\ ]$
	Replace it with:	$\pm[\ (P \ll (i+1)) + P\ ]$

(c) Class 3 ODPs

If $P = (S \ll i) + S$	Search for:	$\pm[\ (S \ll (2i-1)) - (S \ll (i-1)) - S\ ]$
	Replace it with:	$\pm[\ (P \ll (i-1)) - P\ ]$
If $P = (S \ll i) - S$	Search for:	$\pm[\ (S \ll (2i-1)) + (S \ll (i-1)) - S\ ]$
	Replace it with:	$\pm[\ (P \ll (i-1)) + P\ ]$

Class 2 and class 3 ODPs use the second strategy above. An example of a class 2 ODP is $((A0A) \ll 3) - (A0A) = A0A\overline{A}0\overline{A} = A00A0\overline{A}$. Notice the middle two digits in $A0A\overline{A}0\overline{A}$ are positioned as specified in strategy 2, and then $A\overline{A} \rightarrow 0A$ as also specified in strategy 2. Thus, if we *find* $A00A0\overline{A}$, we can *substitute* $B = A0A$ to get $00B00\overline{B}$. The formal definition of these three general classes of ODPs is provided in Table 5.1 and several examples are provided in Tables 5.2 and 5.3.

Recall from section 5.1.1 that we discussed cases where patterns became obfuscated due to *placing digits adjacently* or due to *digits colliding*. The latter case is known as the CSE digit clashing problem and *cannot* be resolved by searching for patterns in SD forms with more nonzero digits. In general, class 1 ODPs will resolve the simplest

Table 5.2: Examples of the first three classes of ODPs with $B = A00A$.

	Class 1	Class 2	Class 3
Desired Substitution	$0\,0\,0\,B00B$	$0\,0\,0B\,0\,0\,0\overline{B}$	$0\,0\,0\,B0\overline{B}$
Regular pattern positions	$A00A$ $A0\,0\,A$	$\overline{A}00\overline{A}$ $A00A$	$\overline{A}\,0\,0\overline{A}$ $A0\,0\,A$
Intermediate result	$A0A\,0\,0\,0A$	$A00A\overline{A}00\overline{A}$	$A0\overline{A}A0\overline{A}$
Final result: the ODP to find	$A0A\,0\,0\,0A$	$A00\,0\,A00\overline{A}$	$A0\,0\,\overline{A}0\overline{A}$

Table 5.3: Examples of the first three classes of ODPs with $B = A00\overline{A}$.

	Class 1	Class 2	Class 3
Desired Substitution	$0\,0\,0\,B00\overline{B}$	$0\,0\,0B\,0\,0\,0B$	$0\,0\,0\,B0B$
Regular pattern positions	$\overline{A}00\overline{A}$ $A0\,0\,A$	$\_A00\overline{A}$ $A00\overline{A}$	$A\,0\,0\overline{A}$ $A0\,0\,\overline{A}$
Intermediate result	$A0\overline{A}\,0\,0\,0A$	$A00\overline{A}A00\overline{A}$	$A0A\overline{A}0\overline{A}$
Final result: the ODP to find	$A0\overline{A}\,0\,0\,0A$	$A00\,0\,\overline{A}00\overline{A}$	$A0\,0\,A0\overline{A}$

cases of clashing (there are likely more complex clashing cases that class 1 ODPs cannot resolve). We illustrated this with the $805 \times A$ example in section 5.1.1.2. In the case where digits are placed adjacently, we can use class 2 and class 3 ODPs to *directly* resolve the simplest cases *without* needing to search SD forms with more nonzero digits. This can significantly improve the run time. As shown in section 5.1.4, the number of SD forms grows very quickly with respect to the number of extra nonzero digits.

We will now examine the type of restrictions that are removed by considering ODPs. Let us try to substitute $C = (B \ll n) + B$ given that $B = (A \ll i) + A$ or substitute $C = (B \ll n) - B$ given that $B = (A \ll i) - A$. If $n = i$, then C is composed of *three* digits of A. We can only find and make these substitutions by considering class 1 ODPs. If $n \neq i$, then C has 4 digits of A, thus there is no clashing so ODPs are not needed. By considering class 2 and class 3 ODPs, a different set of restrictions

Table 5.4: The non-general class 4 ODPs.

Pattern definition	$P = S00\overline{S}$	$P = S0\overline{S}$
Desired Substitution	$0\,0\,0\,P0\,\overline{P}$	$0\,0\,P00\overline{P}$
Regular pattern positions	$\overline{S}\,0\,0\;S$ $S\,0\,0\,\overline{S}$	$\overline{S}0S$ $S\,0\,\overline{S}$
Intermediate result	$S0\overline{S}\overline{S}0\;S$	$S0\overline{S}\overline{S}0S$
Final result: the ODP to find	$S\,0\,S0\;S$	$S\,0\,S0S$

are removed. Let us try to substitute $C = (B \ll n) + B$ given that $B = (A \ll i) - A$ or substitute $C = (B \ll n) - B$ given that $B = (A \ll i) + A$. If $n = i + 1$ or $n = i - 1$, then C is composed of 3 digits of A. Class 2 ODPs enable us to use $n = i + 1$ and class 3 ODPs permit $n = i - 1$. The proof of said claims is trivial and can be immediately seen from Tables 5.1, 5.2 and 5.3. In the special case of $n = i$, C will have only two digits of A (these two digits form a pseudo-ODP, we will later discuss class 5 and class 6 pseudo-ODPs). All other relations between n and i result in C having 4 nonzero digits of A in which there is no clashing, so a CSE algorithm could find and substitute these patterns without considering ODPs. Notice that in all three classes, we recover *multiplicative* decompositions.

5.1.2.4 Non-General Class 4 ODPs

If we use a small value of i in $P = (S \ll i) \pm S$, there are alternate ways to align two instances of P such that three digits of S are produced. By placing all four nonzero digits close together, we can perform multiple digit recodings due to adjacent nonzero digits. Notice that $10\overline{1}\overline{1}01 = 32 - 8 - 4 + 1 = 21 = 10101$. There are only two specific cases of non-general class 4 ODPs, as shown in Table 5.4. For example, if we *find* $A0A0A$, we can substitute $B = A00\overline{A}$ to get $00B0\overline{B}$.

Table 5.5: The general class 5 pseudo-ODPs.

Search for:	$\pm[\ (S \ll (2i)) - S\ ]$
If $P = (S \ll i) + S$, replace it with:	$\pm[\ (P \ll i) - P\ ]$
If $P = (S \ll i) - S$, replace it with:	$\pm[\ (P \ll i) + P\ ]$

Recall from section 5.1.1.1 that H(0) produces a non-optimal solution for $363 \times A$. H(1) to finds the optimal solution but it has to examine many more SD forms compared to only the 2 SD forms searched by H(0). H(0)+ODP uses the same SD forms as H(0) but it can find the optimal solution by using class 4 ODPs. The CSD form of $363 \times A$ is $A0\overline{A}00\overline{A}0\overline{A}0\overline{A}$. We can substitute $B = A0\overline{A}$ to get $00B000\overline{B}00B$. This solution uses only 3 adders. Again, we have resolved a multiplicative decomposition, as we have decomposed $363 \times A$ as $3 \times 21 \times A$ in this example.

5.1.2.5 General Class 5 Pseudo-ODPs

With class 1 ODPs, we resolved one case of digit clashing. However, instead of adding digits to produce a carry, the aligned digits could cancel. For example, $((A0\overline{A}) \ll 2) + A0\overline{A} = A000\overline{A}$. However, notice that $((A0A) \ll 2) - A0A = A000\overline{A}$, therefore if we *find* $A000\overline{A}$, we could substitute either $B = A0\overline{A}$ to get $00B0B$ or $B = A0A$ to get $00B0\overline{B}$. In general, a class 5 pseudo-ODP has the form $(S \ll (2i)) - S$, and if we *find* this, we can substitute $P = (S \ll i) + S$ or substitute $P = (S \ll i) - S$. This is illustrated in Table 5.5.

ODPs have 3 nonzero digits whereas pseudo-ODPs have 2. When an ODP is substituted, 3 old digits are replaced by 2 new ones, thus each substitution decreases the remaining adder distance by 1. Each substitution of a regular pattern (with 2 nonzero digits) also has the same effect on the adder distance, thus one instance of an ODP is *functionally equivalent* to one instance of a regular pattern. Recall that

$((A0A) \ll 2) + A0A = AA00A$, thus we interpret that there are *two* ''occurrences'' of $A0A$ in $AA00A00A0A$. Substituting pseudo-ODPs does not reduce the adder distance because there is no net loss of nonzero digits, thus we do not count pseudo-ODPs as an ''occurrence'' of a pattern. For example, there is only one occurrence of $A0A$ in $A0A00A000\overline{A}$ even though we could substitute $B = A0A$ to get $00B00A000\overline{A}$ or to get $00B0000B0\overline{B}$.

However, by substituting a pseudo-ODP, we may affect the patterns that can be substituted in the next stage. Without class 5 pseudo-ODPs, the first case in which H(k)+ODP cannot find the optimal solution *for any* k is $4875 \times A$. Let us represent $4875 \times A$ as $A0A0\overline{A}000A0\overline{A}0\overline{A}$, then $B = A0A$ can be substituted to yield $00B000\overline{B}0B000\overline{B}$, and finally $C = B000\overline{B}$ is substituted to produce $000000C00000C$. This solution costs 3 adders. Without considering class 5 pseudo-ODPs, we could have substituted the middle $\overline{B}0B$, thus preventing the second substitution. This would produce a solution with 4 adders. Note it is also possible that substituting a pseudo-ODP could hinder further substitutions. In H(k)+ODP, when a pseudo-ODP can be substituted, we search for patterns in the next stage both with and without the substituted pseudo-ODP.

Pseudo-ODPs can only provide a benefit in *future* stages of substitution. Note we can emulate the process of making *more than one* substitution by considering patterns with *more than two nonzero digits*. Thus, pseudo-ODPs provide us with a partial ability to solve the clashing problem for patterns with more than two nonzero digits. In the above example, $4875 \times A = (C \ll 6) + C$, where $C = (B \ll 4) - B = (A \ll 6) + (A \ll 4) - (A \ll 2) - A$. Notice the rightmost A digit of $C \ll 6$ (which is actually $\overline{A}$) and the leftmost A digit of C are aligned and will cancel when added.

Table 5.6: The non-general class 6 pseudo-ODPs.

Pattern definition	$P = S0\overline{S}$	$P = S0\overline{S}$	$P = S0S$
Desired Substitution	$00PP$	$00P0\overline{P}$	$000PP$
Regular pattern positions	$S0\overline{S}$ $S0\overline{S}$	$\overline{S}0S$ $S0\overline{S}$	$S0S$ $S0S$
Intermediate result	$SS\overline{S}\overline{S}$	$S\overline{S}00S$	$SSSS$
Final result: the ODP to find	$S00S$	$S00S$	$S000\overline{S}$

5.1.2.6 Non-General Class 6 Pseudo-ODPs

Like the class 4 ODPs, when small patterns are used, all four nonzero digits can be used to create special cases. There are only three specific cases of class 6 pseudo-ODPs, as defined in Table 5.6. Without class 6 pseudo-ODPs, the first case where H(1)+ODP produces a non-optimal solution is $2325 \times A$. One optimal solution can be obtained by substituting $B = A0A$ in $A0A\overline{A}000A0A0A$ to get $00B000\overline{BB}000B$, and then by substituting $C = B000\overline{B}$ to get $000000C0000\overline{C}$. This costs 3 adders, but without considering class 6 pseudo-ODPs, we would not be able to substitute the middle $\overline{BB}$. This prevents the second substitution and thus we would need 4 adders.

5.1.3 The Remaining Limitations of H(k)+ODP

We will present the smallest coefficients in which H(k)+ODP produces a non-optimal solution (for different k) and an analysis of why this happens. The smallest coefficient that H(0)+ODP produces a non-optimal solution for is 1829. H(1) can find the optimal solution. If we represent $1829 \times A$ as $A000\overline{AAA}00A0A$, we can substitute $B = A0000\overline{A}$ to get $00000B000\overline{B}0\overline{B}$. Without an extra nonzero digit in the initial SD form, $0\overline{AAA}$ must necessarily be represented as $\overline{A}00A$. Thus, H(0)+ODP has only one SD form to search, but $A00\overline{A}00A\overline{A}00A0A$ does not have two occurrences of any

pattern, even if we consider each ODP as one occurrence of the pattern that it was built from. In order for H(0)+ODP to find the optimal solution, we would need to resolve clashing for 3 occurrences of the same pattern (or if we allow patterns to have 3 nonzero digits, then we could consider 2 occurrences of this type of pattern).

The first non-optimal case for H(1)+ODP is $3255 \times A$. H(2) can find the optimal solution. Starting with $A00\overline{AAA}0AAA00\overline{A}$, substitute $B = A00\overline{A}$ to get $000B000\overline{BB}000B$, then substitute $C = B000\overline{B}$ to obtain $0000000C0000\overline{C}$. In order to find the optimal solution with only up to one extra nonzero digit in the SD form, we would need to resolve clashing for 4 occurrences of the same pattern. The problem is similar to the case in which H(0)+ODP produces the first non-optimal solution. Without 2 (or more) extra nonzero digits in the initial SD form, either $0\overline{AAA}$ collapses into $\overline{A}00A$ or $0AAA$ collapses into $A00\overline{A}$, however we need *both of these* to be represented in the form with more digits in order to find and substitute the patterns as shown above.

H(2)+ODP faces a similar problem. The first non-optimal solution is produced for $5049 \times A$. The optimal solution is as follows: starting with $A00AAA0AAA00A$, substitute $B = A00A$ to get $000B000BB000B$, then substitute $C = B000B$ to get $0000000C0000C$. This optimal solution was found by H(3).

For any value of k, the first non-optimal case for H(k)+ODP is 21403 (thus H(k)+ODP produces solutions which happen to be as good as optimal for all constants up to 14 bits). The problem in this case arises from selecting the *wrong instance* of the pattern. The optimal solution can be expressed within the ODP framework. Starting with $A0A0A000\overline{AA}00\overline{A}0\overline{A}$, substitute $B = A0A$ to get $00B0A00000\overline{B}0\overline{B}0\overline{A}$, then substitute $C = B0A$ to get $0000C00000\overline{B}000\overline{C}$. Notice that in $\overline{AA}00\overline{A}0\overline{A}$ (the

rightmost digits of the starting SD form), the rightmost digit of the ODP $\overline{A\overline{A}}00\overline{A}$ competes with the leftmost digit of $\overline{A}0\overline{A}$. In this case, we could substitute $B = A0A$ to get $00\overline{B}0\overline{B}0\overline{A}$ or $\overline{A\overline{A}}0000\overline{B}$. Unfortunately, H($k$)+ODP selects the latter case, which does not lead to an optimal solution.

Better solutions can be obtained by resolving this digit contention problem, however this requires extra computation. As justified in section 5.1.4, since our primary goal is to reduce the run time, we do *not* solve the digit contention problem in H(k)+ODP. In section 5.1.5, we will discuss how substitutions are selected when there is more than one best option.

5.1.4 Run Time Versus Minimizing Adders

To the best of our knowledge, H(k) is the best existing SCM heuristic. As shown in [33], H(2) is very close to optimal. For all constants up to 19 bits, it produces solutions that require on average only 1.0% more adders than the optimal. Thus, there is a *strict limitation* on how much improvement can be made. This applies to *any* new algorithm. Any two algorithms that are nearly optimal will inherently have a similar performance in terms of minimizing adders. Whether a heuristic produces solutions within 1% or 1.5% of the optimum usually has little impact in practice. Furthermore, the results are *not guaranteed* to be optimal. Instead of trying to improve the performance to within 0.5% of the optimal, we believe that it is more useful to make an algorithm that is at least as good as H(k) in terms of minimizing adders but can run in significantly less time. As shown in our results (section 5.1.6), H(k)+ODP sometimes achieves over one order of magnitude in run time reduction compared to H(k) while still marginally improving the number of adders.

Table 5.7: An estimate of the average number of SD forms used by H(k) at each bit width and each k.

	$k=0$	$k=1$	$k=2$	$k=3$
20 bits	5	40	164	465
24 bits	7	68	328	1075
28 bits	10	111	602	2216
32 bits	13	161	976	4018
40 bits	20	293	2197	11060
48 bits	44	760	6588	38511

As shown in section 5.1.2.3, class 2 and class 3 ODPs do not resolve clashing but rather enable us to find patterns and make substitutions that would otherwise require SD forms with more nonzero digits. Recall that H(0) cannot find the optimal solution for $363 \times A$ whereas both H(1) and H(0)+ODP can. H(0)+ODP does a little more searching within each SD form while H(1) searches many more SD forms. In Table 5.7, we show that each increment in k causes the average number of SD forms to increase extremely fast. These values are *estimates*, as we experimentally obtained them using uniformly distributed random constants at each bit width and each k (1000 constants up to 32 bits, 100 constants at 40 and 48 bits). Good solutions (in terms of minimizing adders) require a large k, however for constants of large bit width, numerous SD forms must be searched which translates to long run times. ODPs facilitate a more efficient search and substitution of patterns in each SD form, so it may be possible to search fewer SD forms (in order to reduce the run time) and still produce solutions that are at least as good (compared to using more SD forms and not using ODPs). Conclusively, we are interested if H(k)+ODP can produce better solutions in less run time than H($k+n$) for $n \geq 1$. Note that H(k)+ODP requires more run time than H(k) because the same SD forms are searched, but H(k)+ODP must also search for ODPs, thus we are *not* interested in a comparison between them.

5.1.5 Implementation Details

We implemented H(k) with *optional* ODPs in C. We will first describe the common components of H(k) and H(k)+ODP. Both algorithms use the same initial SD forms (given the same k). Using a recursive function, we implemented a depth-first version of the SD generation algorithm in [35], which is described in detail in section 4.5.3.2. The Hartley algorithm [30] is used to search and substitute patterns in each SD form. However, neither Hartley nor H(k) specified which pattern should be substituted if several occur the maximum number of times. Furthermore, which pattern instance should be substituted is not specified. For example, given $A0A0A0\overline{A}0\overline{A}$, we could substitute $B = A0A$ to get $00B0A000\overline{B}$ or $A000B000\overline{B}$, or substitute $B = A000\overline{A}$ to get $A00000B0B$, or substitute $B = A00000\overline{A}$ to get $0000A0B0B$. The choice made now may affect which patterns can be substituted on the next iteration. However, our objective is to improve the run time, so we will not consider every possible best option. At each iteration, for each pattern that has the maximum occurrence, we will substitute *all* occurrences with a right bias (for example, substitute $B = A0A$ in $A0A0A0A0A$ to get $A000B000B$) and *all* occurrences with a left bias (to get $00B000B0A$). At each iteration, each substitution is done *one at a time*. Notice that we have missed $00B0A000B$ in this example.

Our version of the Hartley algorithm is a *branch and prune* algorithm. More than one substitution *at each stage of substitution* causes branching, and only considering maximally occurring patterns is a form of pruning. Thus, the worst case run time of H(k)+ODP is of exponential order with respect to the bit width of the SCM constant, however the *absolute* run time is typically reasonable for the problem sizes of most importance in practice. The justification for using an exponential heuristic

was provided in section 4.8.2. We implemented our version of the Hartley algorithm in a depth-first recursive manner to minimize the memory usage.

If we consider ODPs in the Hartley algorithm, we must also decide whether an ODP or a regular pattern (with 2 nonzero digits) should take priority if both compete for the same digit. As illustrated in section 5.1.3, the rightmost digit of the ODP $\overline{AA}00\overline{A}$ competes with the leftmost digit of $\overline{A}0\overline{A}$ in $\overline{AA}00\overline{A}0\overline{A}$. Since each CSE representation has a limited number of digits, we prefer to substitute regular patterns since these only consume 2 nonzero digits (compared to 3 digits for an ODP), thus leaving more digits that could be substituted. If we want all substitutions to have a left bias, we search from left to right. Once a pattern is found, all digits in the pattern instance are labeled as ''used'' (a different marker is used for each type of pattern). This prevents multiple instances of the same pattern from competing for the same digit. Thus, *first* we search and label regular patterns, *then* we do this for ODPs. However, we experimentally discovered that better solutions are obtained if ODPs that contain the leftmost digit in the CSE representation are given first priority (then the priority goes to regular patterns, and finally to ODPs that do not contain the leftmost digit). An analogous set of priority rules are used when searching from right to left (for right biased substitutions).

5.1.6 Experimental Results

All of the experimental results in this thesis were benchmarked on a set of identical 3.06 GHz Pentium 4 Xeon workstations running Linux. All algorithms were implemented in C or C++ and were compiled with gcc 3.2.3. In this section, we will compare H(k) and H(k)+ODP with several existing algorithms. All algorithms were

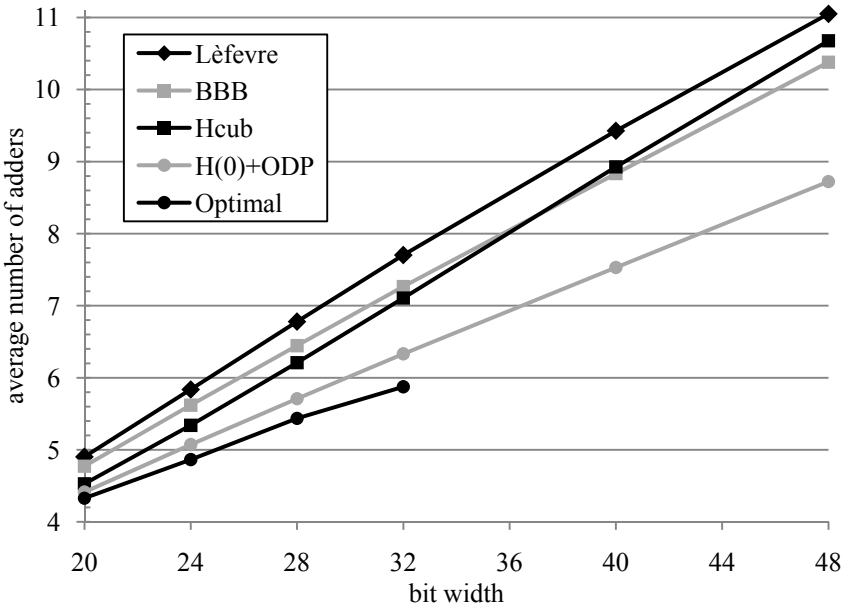


Figure 5.1: The average SCM cost versus the bit width.

tested with the same constants at each bit width. We used all constants up to 20 bits. At larger bit widths, uniformly distributed random constants were used. We used 100000 constants at 24, 28, and 32 bits, 1000 constants at 40 bits, and 100 constants at 48 bits.

Lefèvre [32] presents a CSE algorithm that searches for the pattern with the most nonzero digits using only the CSD form. We used the C implementation written by Rigo from Lefèvre's website [45]. We expect this algorithm to perform similarly to Hartley's algorithm [30] since substituting a pattern with m nonzero digits n times is equivalent to substituting a pattern with n nonzero digits m times, as shown in [30]. Prior to H(k), the best existing SCM algorithm was BBB (better of Bernstein or BHM, section 4.4.1). We implemented BBB in C. We believe our implementation is no less efficient than any of the other algorithms that we implemented. However, as expected, H(0)+ODP outperforms both Lefèvre and BBB, as shown in Figure 5.1.

The best existing MCM algorithm in terms of its SCM performance is Hcub [6]. We used the C++ implementation from the website of the authors [46]. From Figure 5.1, H(0)+ODP on average outperforms Hcub. In fairness, the heuristic in Hcub is catered towards a slightly different problem. In MCM, one may share intermediate terms *within* a constant or *between* constants. MCM heuristics generally try to do both whereas SCM heuristics can focus on maximizing the sharing of terms only *within* a constant. Our results confirm that MCM algorithms do not perform SCM as well as algorithms designed *solely for SCM*.

Although not shown, the average run times of both Lefèvre and BBB are faster than H(0) and the average run time of Hcub is typically between that of H(0) and H(1). The run times for H(k) are provided in Table 5.9.

In order to produce good solutions, a large value of k is needed. In Figure 5.1, we compare H(0)+ODP and our optimal algorithm from section 5.2. Clearly, a non-trivial improvement in the average number of adders is possible by increasing the value of k. We tested H(k) with $k = 0, 1, 2, 3$ and H(k)+ODP with $k = 0, 1, 2$. The average number of adders is shown in Table 5.8. When we check if H(k)+ODP produces better solutions than H($k + n$) for $n \geq 1$, the comparisons can be quite close. The average run times are provided in Table 5.9. For bit widths up to 32 bits, the average percent more adders than the optimal is shown in Figure 5.2. At 20 bits, H(2)+ODP is within 0.2% of the optimum (compared to 0.9% for H(2)). Thus, we are able to recover most of the obfuscated patterns by resolving only the simplest cases of clashing. However, the solutions become increasingly farther from the optimal as the bit width increases. With more adders, more complex cases of clashing can occur (i.e. more occurrences of a pattern with more nonzero digits).

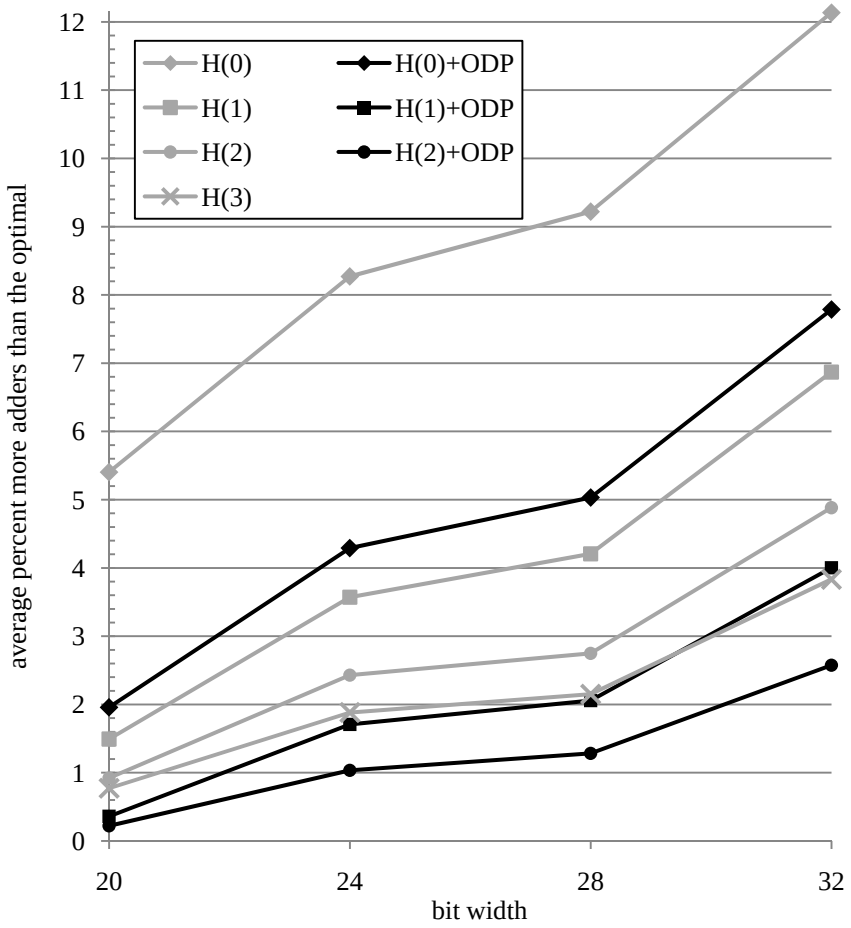


Figure 5.2: The average percent more adders than optimal versus bit width.

Table 5.8: The average number of adders versus bit width.

	20 bits	24 bits	28 bits	32 bits	40 bits	48 bits
H(0)	4.563	5.267	5.938	6.588	7.843	9.079
H(0)+ODP	4.414	5.073	5.710	6.332	7.531	8.723
H(1)	4.394	5.038	5.665	6.278	7.451	8.605
H(1)+ODP	4.345	4.947	5.548	6.110	7.228	8.348
H(2)	4.369	4.983	5.586	6.162	7.290	8.400
H(2)+ODP	4.338	4.915	5.506	6.026	7.106	8.173
H(3)	4.362	4.956	5.553	6.100	7.205	8.285
Optimal	4.329	4.864	5.437	5.875	---	---

Table 5.9: The average run time (seconds) versus bit width.

	20 bits	24 bits	28 bits	32 bits	40 bits	48 bits
H(0)	0.001	0.001	0.003	0.006	0.044	0.234
H(0)+ODP	0.001	0.001	0.003	0.008	0.056	0.324
H(1)	0.004	0.012	0.035	0.096	0.871	5.338
H(1)+ODP	0.006	0.016	0.045	0.125	1.152	7.717
H(2)	0.022	0.076	0.241	0.831	9.066	63.878
H(2)+ODP	0.029	0.099	0.317	1.081	12.125	87.093
H(3)	0.075	0.313	1.228	4.738	55.589	218.857
Optimal	0.006	0.046	0.325	8.851	---	---

As expected, both H($k+1$) and H(k)+ODP perform better than H(k) at the expense of run time. However, compared to H(k), H($k+1$) typically requires several times the amount of run time due to the number of initial SD forms (as explained in section 5.1.4). Conversely, only about 30% more run time is typically needed for H(k)+ODP to also search for ODPs (in addition to regular patterns that both it and H(k) search for). By increasing k, we expand the search space by providing more combinations of signed powers of two. As explained in section 5.1.1.1, CSE provides a means to collect common terms once we have decided which signed powers of two will be used to construct the constant (i.e. which digits are in the initial SD form). Conversely, by using ODPs and pseudo-ODPs, we are able to more efficiently search and replace patterns in each SD form. Conclusively, increasing k increases the size of the search space whereas using ODPs improves the ability to search *within the search space provided* by the initial SD forms. Our results show that much less computation is needed to search for ODPs compared to increasing k. Also, improving the efficiency of the search may lead to better solutions than simply searching more items.

We are interested if H(k)+ODP can produce better solutions in less run time than H($k+n$) where $n \geq 1$. A summary of when this happens and the improvement in

Table 5.10: The average improvement in run time when H(k)+ODP can produce better average solutions than H($k+n$) for $n \geq 1$. The improvement is specified as a ratio (i.e. how many times faster).

Bit width	H(1)+ODP		H(2)+ODP	
	Outperforms	Improvement in the run time	Outperforms	Improvement in the run time
20	H(3)	13.2x	H(3)	2.6x
24	H(3)	19.7x	H(3)	3.2x
28	H(3)	27.3x	H(3)	3.9x
32	H(2)	6.6x	H(3)	4.4x
40	H(2)	7.9x	H(3)	4.6x
48	H(2)	8.3x	H(3)	2.5x

run time is provided in Table 5.10. Although not shown in Table 5.10, H(0)+ODP outperforms H(3) for constants up to 13 bits and outperforms H(1) up to 14 bits. ODPs become increasing beneficial as the amount of clashing increases (as well as other pattern obfuscation problems, such as those caused by placing digits adjacently). The likeliness of clashing is related to the *density* of nonzero digits (as opposed to the *number* of nonzero digits). As the bit width increases, each increment in k increases the density of nonzero digits by a lesser extent. Consequently, H(1)+ODP outperforms H(3) only up to 28 bits, although H(1)+ODP outperforms H(2) up to at least 48 bits (we did not test beyond 48 bits, as such large bit widths are less relevant in practice). H(2)+ODP outperforms H(3) up to at least 48 bits.

The improvement in the number of adders decreases as k increases (i.e. the improvement from H(2) to H(3) is less than from H(1) to H(2)). It becomes increasingly difficult to improve the solutions as we approach the optimum. It therefore seems counter-intuitive that the run time of an optimal algorithm can sometimes be *faster* than H(k) and H(k)+ODP, as shown in Table 5.8. This will be discussed in the next section, where we propose our optimal SCM algorithm.

5.2 Optimal SCM

In this section, we will propose the bounded inverse graph enumeration (BIGE) algorithm. It is an exhaustive optimal SCM algorithm which uses aggressive pruning. An outline is provided in section 5.2.1 and we will discuss the exhaustive search methodology in section 5.2.2. Details regarding the adder-operation are illustrated in section 5.2.3. Finally, the experimental results are presented in section 5.2.4 along with many statistical distributions that were previously unknown. We benchmark up to 32 bits (compared to 19 bits in the MAG algorithm [1]), as we believe this caters to the problem sizes of the most practical importance in digital signal processing.

5.2.1 Outline of the BIGE Algorithm

5.2.1.1 The Bounding, Inverse Graph, and Enumeration Components

In the BIGE algorithm, we introduce the idea of a *bounding* function. Although H(k)+ODP is a *heuristic*, it can be reused within our optimal algorithm. If H(k)+ODP finds a solution with n adders, we use an exhaustive search to either find a solution with up to $n-1$ adders or prove that no solution with up to $n-1$ adders exists (in which case we have *exhaustively verified* that the solution found by H(k)+ODP is optimal). Not needing to exhaustively search the last adder results in a significant decrease in the size of the search space, thus we can search larger constant coefficients within a reasonable amount of time. However, a tight bounding heuristic is required in order to take advantage of this approach. If the bounding heuristic typically produces solutions that are far from optimal, it provides little or no benefit compared to using only an exhaustive search.

The BIGE algorithm uses both the CSE and DAG frameworks. The CSE-based heuristic H(k)+ODP finds the upper bound (number of adders) while the DAG-based exhaustive search finds the lower bound. We progressively tighten both bounds until they meet, at which point we have an optimal solution.

Unlike the MAG algorithm, we do not create a lookup table for all constants up to 2^b. Instead, we solve the SCM problem *for the given constant*. This avoids the need for a table which grows exponentially in size with respect to the bit width. Furthermore, it is unlikely that large parts of this table will be used in practice. We conjecture that most designs will not contain several thousands or millions of SCM instances, but at 32 bits, there are more than 2 billion odd and positive integers.

Since we solve the problem for *the given target* t, we can use t to find all of the useful intermediate terms that are one adder away from t. Using these terms, we find all of the useful terms that are two adders away from t, and so on. This is exactly like the *inverse graph traversal* (IGT) technique that was used to compute the adder distance in section 3.2.6. Once we project backwards far enough, we check for a common element between the backwards projection and the successor set S (or the ready set R). For a single target, this requires much less computation than blindly constructing all of the possible terms with n adders and then checking whether or not the target was constructed. Blind construction is better for finding the SCM solutions of *numerous* targets simultaneously, as is done in the MAG algorithm.

An exhaustive search for adder distance n requires one to *enumerate* and test each of the possible graph topologies that have n adders. By doing this in the order of increasing n, we are guaranteed to find the minimum number of adders. As explained in section 4.1.3, graphs are more suitable than CSE for an exhaustive search.

Input : the SCM target t
Output: the optimal SCM solution for t

if ($t \in C_1$) { **return** solution of c_1, where $t = c_1$ and $c_1 \in C_1$ }
if ($t \in C_2$) { **return** solution of c_2, where $t = c_2$ and $c_2 \in C_2$ }
if ($t \in C_3$) { **return** solution of c_3, where $t = c_3$ and $c_3 \in C_3$ }
if ($t \in C_4$) { **return** solution of c_4, where $t = c_4$ and $c_4 \in C_4$ }

comment: at this point, no solution with up to 4 adders exists, so a solution with 5 adders is optimal even if found by a heuristic

[hk_odp_solution, hk_odp_cost] = get_solution(H(1)+ODP)
if (hk_odp_cost==5) { **return** hk_odp_solution }
for $i = 1$ to 10 {
    if (solution found by exact_cost_5_test(i)) { **return** solution found }
}

comment: at this point, no solution with up to 5 adders exists

if (hk_odp_cost==6) { **return** hk_odp_solution }

comment: try tightening the heuristic bound

[hk_odp_solution, hk_odp_cost] = get_solution(H(2)+ODP)
if (hk_odp_cost==6) { **return** hk_odp_solution }

comment: intrinsic cost 6 tests

for $i = 1$ to 3 {
    if (solution found by exact_cost_6_test(i)) { **return** solution found }
}

comment: non-intrinsic cost 6 tests, create all possible R sets with 2 elements (which means 1 adder was spent) and use exact distance 5 tests

for each $c_1 \in C_1$ {
    $R = \{1, c_1\}$
    for $i = 1$ to 17 {
        if (solution found by exact_distance_5_test(i) with the current R) {
            return solution found
        }
    }
}

comment: at this point, no solution with up to 6 adders exists

if (hk_odp_cost==7) { **return** hk_odp_solution }
else { **print** ‘‘no solution found’’, **return** null }

Algorithm 1: The bounded inverse graph enumeration (BIGE) algorithm. Note it is limited to 7 adders and there are multiple exit points (but every path is covered).

5.2.1.2 The Strategy and the Choice of the Bounding Heuristic

The BIGE algorithm is described in Algorithm 1. We use a lookup table to check whether the target has a solution with up to 4 adders. This is expressed in lines 1-4 of Algorithm 1. We precompute the sets C_1, C_2, C_3, and C_4, as these do not depend on t. These C_n sets are sorted and a binary search is used to check if $t \in C_n$. One solution is stored for each element in C_n. At 32 bits, C_4 contains less than 6 million elements, so all of the elements and solutions in C_4 only require a few tens of megabytes to store. We will discuss the creation of these C_n sets in section 5.2.2.2.

Based on the increase in size of C_n as we increment n and other statistical projections, we *estimate* that C_5 at 32 bits will contain around 300 million elements. Precomputing C_5 would require a lot of computation and a few gigabytes to store, which limits its practicality. Thus, we switch to a new strategy at 5 adders.

Before we created the BIGE algorithm, we were already aware that H(1)+ODP produced solutions with about 0.4% more adders than the optimal on average at 19 bits (using the MAG algorithm as the reference). Since the number of adders is an integer, this means that H(1)+ODP produces a non-optimal solution roughly every 1 in 250 cases (assuming we are very infrequently off by more than 1 adder). H(2)+ODP averages about 0.2% more adders than optimal, but at a substantial increase in run time (refer to Table 5.9 in section 5.1.6). If we could confirm that H(1)+ODP happened to produce an optimal solution, we would not need to use H(2)+ODP, thereby saving run time. This is the *fundamental reason* why the BIGE algorithm can sometimes produce optimal SCM solutions *in less run time than H(k)+ODP*, especially if k is large. We could apply this same argument to H(0)+ODP, however the *absolute* run time of H(1)+ODP at 32 bits is on average 0.1 seconds. We decided

to use H(1)+ODP directly because the improvement over H(0)+ODP is considerable (at 19 bits, about 2% more than optimal compared to 0.4% more, we later found that at 32 bits this grows to 7.8% versus 4.0%).

H(k)+ODP is nearly optimal and we will show in section 5.2.4 that the final solution returned by the BIGE algorithm is very often found by this bounding heuristic. We will discuss the exhaustive search parts of the BIGE algorithm in section 5.2.2. After we have exhaustively confirmed that no solution with 5 adders exists, if the original solution found by H(1)+ODP has 6 adders, then the solution is optimal (line 9 in Algorithm 1). If not, we could now use an exhaustive search immediately or we could first attempt to tighten the bound. Although H(2)+ODP produces generally solutions that are marginally better, at 32 bits its absolute run time is about 1 second whereas typically a couple minutes are needed to do an exhaustive search at 6 adders. We tighten the bounding heuristic using H(2)+ODP because increasing the run time of many cases by 1 second and decreasing the run time of a few cases by a couple minutes still results in an overall decrease in the average run time. Although not shown in this thesis, the improvement from H(2)+ODP to H(3)+ODP is very limited and requires roughly $5x$ more run time. We do not use H(3)+ODP in the BIGE algorithm.

Note that when we use H(2)+ODP, we only search and substitute patterns in SD forms with *exactly* 2 extra non-zeros digits, to avoid repeating the search already done by H(1)+ODP. Recall from section 4.5.3.2 that when we generate SD forms, if we are at depth d in the tree, the SD form will have d nonzero digits. Let z denote the CSD cost of t. We would normally apply the Hartley algorithm to any valid SD form found at a depth of *up to* $z + 2$, now we only do so at a depth of *exactly* $z + 2$.

5.2.2 Exhaustive Searching in the BIGE Algorithm

5.2.2.1 A Generalized Approach for Exhaustively Searching SCM

Consider the problem of determining if t has a SCM solution with m adders (by progressively increasing m, we will find the minimum SCM cost). We can use distance k tests to determine if t can be constructed with k *more* adders. If we enumerate every possible way to use $m-k$ adders to construct some terms and then for each of these cases we apply the distance k tests, we will have an exhaustive search for m adders. Thus, we must construct every possible set R with $m-k+1$ elements. Given a set of already constructed terms R, in order to test for distance k (not *cost* k), we must test each graph topology with k adders (the inverse graph traversal method was illustrated in section 3.2.6 and we will provide more examples in this section).

In the BIGE algorithm, adder costs 1-4 are tested in the same way as the MAG algorithm (we use precomputed lookup tables). At cost 5 ($m=5$), we use distance $k=5$ tests, thus we only need to consider all possible sets of R with $m-k+1=1$ element. There is only $R=\{1\}$. At cost 6, we cover some topologies with $k=6$ and $R=\{1\}$ (the intrinsic tests on line 13 of Algorithm 1) and the remaining topologies are covered with $k=5$ and all of the possible sets of R with 2 elements (the non-intrinsic tests on line 17 Algorithm 1).

Although the BIGE algorithm as described in Algorithm 1 is limited to 7 adders, this can easily be extended. For example, to exhaustively search at 7 adders, we could create all possible sets of R with 3 elements (so 2 adders have been used) and then to each of these, we would apply the distance 5 tests. This would enable us to confirm a heuristic solution with 8 adders is optimal. We could extend this to all possible sets of R with 4 elements, and so on.

For a fixed m, as we increase k, more pruning can be done because we will traverse further backwards in the adder distance tests. This also means less adders are used to construct terms in all of the possible ways, which obviously has no pruning. However, using adder distance tests with larger k requires precomputed C_n sets with larger n. In general (for an arbitrary R), in order to test for distance k, one of the tests that is always be needed is $t/C_{k-1} \cap S \neq \emptyset$. Although C_{k-1} is precomputed once, we will need to divide t by every element in C_{k-1} for every SCM problem instance. For example, one must consider whether constructing C_5 and computing t/C_5 is worthwhile so that general distance 6 tests can be used (versus using distance 5 tests and one more adder in R). However, if R only has 1 or 2 elements, we can exploit some properties to reduce the computation. Due to this, at cost 6 (not *distance* 6), using distance 5 tests requires less computation. At higher costs, this may no longer be the case.

We have not investigated this tradeoff between pruning and precomputing, as it has little practical importance. In most DSP applications, constants are typically represented on up to 32 bits. In our benchmark experiments, all of the 100000 random 32 bit constants each required only up to 7 adders. Even if there is a 32 bit constant that requires 8 adders, it is so unlikely that it is of negligible practical importance.

5.2.2.2 Construction of the C_1 to C_4 Sets

This is essentially the original MAG algorithm [1], but as shown in [2], vertex reduction and multiplicative partitioning can be used to identify equivalent graphs, thus eliminating redundancy in the exhaustive search. This was explained in detail in section 4.1.2. All of the possible vertex reduced graph topologies are shown in Figure 5 in [2]. Using these topologies, it follows that C_1 to C_4 are constructed as follows.

$$C_1 = \mathcal{A}(1,1) \tag{5.1}$$

$$C_2 = \mathcal{A}(C_1,1) \ \cup \ (C_1 \cdot C_1) \tag{5.2}$$

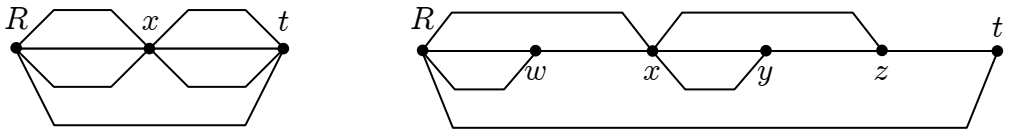
$$C_3 = \mathcal{A}(C_2,1) \ \cup \ (C_2 \cdot C_1) \tag{5.3}$$

$$C_4 = \mathcal{A}(C_3,1) \ \cup \ (C_3 \cdot C_1) \ \cup \ (C_2 \cdot C_2) \ \cup \ \mathcal{A}(z, C_1 \cdot \mathcal{A}(z,1)) \text{ for each } z \in C_1 \tag{5.4}$$

5.2.2.3 The Cost 5 Tests

In this section, we will show how line 8 of Algorithm 1 is evaluated. Note that *cost 5* tests are *not* the same as the *distance* 5 tests (on line 17). The distance 5 tests are used to test for *adder distance*, i.e. they establish whether or not t can be constructed with 5 *more* adders. The cost 5 tests are functionally equivalent to using the distance 5 tests when $R = \{1\}$. However, when $R = \{1\}$, we can exploit some properties to reduce the computation.

We will check whether each graph topology permits the construction of t by using the inverse graph traversal technique (the IGT method was illustrated in detail for distance 2 in section 3.2.6). Let us illustrate how we apply the same technique to a graph topology with 5 adders. Consider graph number 11 (cost 5) from Figure 5 in [2], which we show in Figure 5.3(a). First we apply inverse vertex reduction so that each node represents one adder and thus has two edges coming to the node. This is shown in Figure 5.3(b). There are generally multiple solutions, so we pick the one that yields the least amount of computation when it is used to perform an inverse graph traversal test. As explained in section 3.2.6, we obtain the topology for the distance 5 test by splitting the input node, which is shown in Figure 5.4.



(a) Vertex reduced form of graph number 11.

(b) Non-vertex reduced form of graph number 11.

Figure 5.3: Cost 5, graph topology number 11 from Figure 5 in [2].

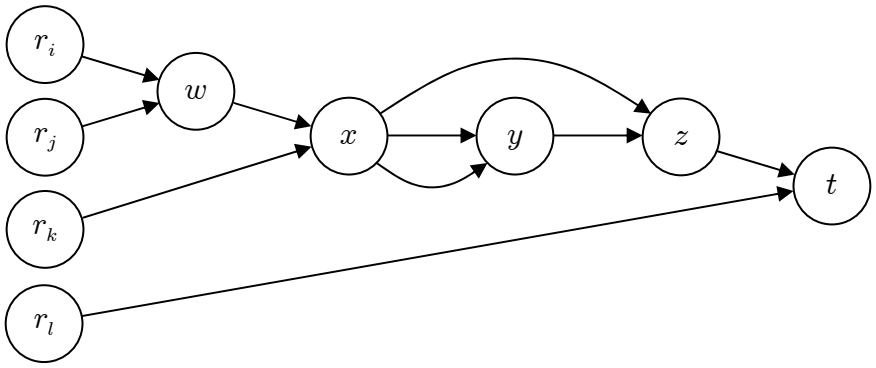


Figure 5.4: Cost 5, graph topology number 11 for testing adder distance. Note that $r_i, r_j, r_k, r_l \in R$ and thus $w \in S$.

Now let us illustrate how to design an inverse graph traversal test for graph number 11. Based on the labeling in Figure 5.4, $t \in \mathcal{A}(z, R)$. From Lemma 2 in section 3.2.1, it follows that $z \in \mathcal{A}(t, R)$. If we are to construct t using this topology, the only allowed values of z are $z \in \mathcal{A}(t, R)$. Now let us traverse further backwards. Clearly, $z \in \mathcal{A}(x, y)$ and $y \in \mathcal{A}(x, x) = C_1 \cdot x$. Thus $z \in \mathcal{A}(x, C_1 \cdot x) = (\mathcal{A}(1, C_1)) \cdot x \subseteq C_2 \cdot x$. In other words, the subgraph formed by nodes x, y, and z implements multiplication by some of the elements in C_2. Knowing all of the possible values of z that will permit the construction of t with this topology, it follows that all of the possible values allowed for x are $x \in \frac{z}{C_2}$. This type of division test was explained in detail in section 3.2.4. Because some elements in C_2 are constructed as $C_1 \cdot C_1$, by enforcing

$x \in \frac{z}{C_2}$, we will actually test another graph topology at the same time (this other graph topology is identical to Figure 5.4 except that $z \in \mathcal{A}(y, y)$).

So far, we have established that all of the permitted values of x are $x \in \frac{z}{C_2} = \frac{\mathcal{A}(t,R)}{C_2}$. Continuing the backwards traversal, $x \in \mathcal{A}(w, R)$, thus the only permitted values of w are $w \in \mathcal{A}(x, R) = \mathcal{A}(\frac{\mathcal{A}(t,R)}{C_2}, R)$. Since $w \in \mathcal{A}(R, R)$, it is possible that $w \in R$, but this would mean t is distance 4 (if $w \in R$, we would only need 1 adder to construct x, 2 for y, 3 for z, and 4 for t). Since we search in the order of increasing adder cost in the BIGE algorithm, we know t is not distance 4. Thus, we are only interested in $w \in S$. We have established that $w \in \mathcal{A}(x, R) = \mathcal{A}(\frac{\mathcal{A}(t,R)}{C_2}, R)$, so we should check if there is a common element between S and $\mathcal{A}(\frac{\mathcal{A}(t,R)}{C_2}, R)$. Conclusively, t is constructible with this topology iff $\mathcal{A}(\frac{\mathcal{A}(t,R)}{C_2}, R) \cap S \neq \emptyset$.

In graph topology number 11, if $R = \{1\}$, notice that z has a SCM cost of 4 adders, thus $z \in C_4$. Recall that C_4 is precomputed so that we can use a lookup table approach to check if $t \in C_4$ in the BIGE algorithm. As stated above, $z \in \mathcal{A}(t, R)$. When $R = \{1\}$, it follows that t can be constructed with this graph topology iff there is a common element between C_4 and $\mathcal{A}(t, R)$, or equivalently, iff $\mathcal{A}(t, R) \cap C_4 \neq \emptyset$. This requires much less computation than the generalized test $\mathcal{A}(\frac{\mathcal{A}(t,R)}{C_2}, R) \cap S \neq \emptyset$ (C_4 is only ever computed *once* and is reused for every SCM problem instance).

In Table 5.11, we provide a summary of all of the cost 5 tests. Each test may simultaneously test for multiple topologies. This arises when we divide by C_n for $n \geq 2$, as there are several topologies that are identical except for the subgraph that implements multiplication by C_n. This was illustrated in the above example. In Table 5.11, * denotes the transpose of the graph from [2]. The transpose is the left/right reflection of the topology. For SCM, a graph and its transpose produce the same set

Table 5.11: A summary of the cost 5 tests.

Case	Test	Graphs covered in [2]
1	$\frac{t}{C_1} \cap C_4 \neq \emptyset$	13-26
2	$\frac{t}{C_2} \cap C_3 \neq \emptyset$	27-29
3	$\mathcal{A}(t,1) \cap C_4 \neq \emptyset$	1-11
4	$\mathcal{A}(t,C_2) \cap C_2 \neq \emptyset$	12
5	for each $y \in C_2 : \mathcal{A}\left(\frac{\mathcal{A}(t,y)}{C_1}, y\right) \cap \{1\} \neq \emptyset$	31, 32*
6	for each $x \in C_1 : \mathcal{A}\left(\frac{\mathcal{A}(t,x)}{C_2}, x\right) \cap \{1\} \neq \emptyset$	31*, 32
7	for each $x \in C_1 : \mathcal{A}\left(\frac{\mathcal{A}(t,x)}{C_1}, x\right) \cap C_1 \neq \emptyset$	33
8	for each $x \in C_1 : \frac{\mathcal{A}\left(\frac{\mathcal{A}(t,x)}{C_1}, 1\right)}{x} \cap C_1 \neq \emptyset$	30
9	for each $x \in C_1 : \mathcal{A}\left(\frac{\mathcal{A}(t,C_1 \cdot x)}{C_1}, x\right) \cap \{1\} \neq \emptyset$	33*
10	for each $x \in C_1$ and for each $y \in \mathcal{A}(x,1)$: $\mathcal{A}\left(\frac{\mathcal{A}(t,y)}{C_1}, y\right) \cap \{x\} \neq \emptyset$	34

of values (due to multiplicative partitioning, as discussed in section 4.1.2), however when testing for adder distance, different topologies are obtained after we split the input node, thus we must test both topologies. We share common sets between different tests whenever possible. For example, cases 4 and 5 from Table 5.11 both first compute $\mathcal{A}(t, C_2)$ before performing different operations. These tests are done together to avoid redundancy or storing any intermediates. Aside from this grouping, the tests are sorted roughly by run time since we stop after the first solution is found.

Table 5.12: A summary of the intrinsic cost 6 tests.

Case	Test	Graphs covered in [2]
1	$\frac{t}{C_2} \cap C_4 \neq \emptyset$	13*, 14*, 15*, 16*, 17, 18, 19*, 20*, 21, 22, 22*, 23, 23*, 24*, 25*, 26, 26*, 27, 28
2	$\frac{t}{C_3} \cap C_3 \neq \emptyset$	24, 25, 27*, 28*, 29
3	$\frac{t}{C_1} \cap C_5 \neq \emptyset$	13, 14, 15, 16, 19, 20

5.2.2.4 The Intrinsic Cost 6 Tests

As mentioned earlier, we can search for cost 6 by applying distance $k = 5$ tests to each possible R with 2 elements. To test for distance 5, we do a test for each graph topology with 5 adders, but *some* tests (for some of these topologies) can be simplified when R has 2 elements. These simplified tests (which are not applicable if R is arbitrary) are known as the intrinsic cost 6 tests (line 13 of Algorithm 1) since we can directly solve for cost 6 without guessing what is in R. There are three cases summarized in Table 5.12. Note in case 3, we do not construct C_5, but rather the cost 5 tests (summarized in Table 5.11) are performed for each element in $\frac{t}{C_1}$.

5.2.2.5 The Non-Intrinsic Cost 6 Tests Using Distance 5 Tests

Not all of the graph topologies are covered by the cost 6 intrinsic tests. For the remaining topologies, we will use distance 5 tests for each possible R with 2 elements (line 15 in Algorithm 1). Thus R has form $R = \{1, c_1\}$ where $c_1 \in C_1$. Note that the distance 5 tests in this section can be applied to arbitrary R. It is possible that several intermediate terms can form a closed loop, in which case it is difficult to use the IGT method to design tests. This happens when a graph is a ‘‘leapfrog’’ graph (this naming comes from the classification of graphs done in [2]).

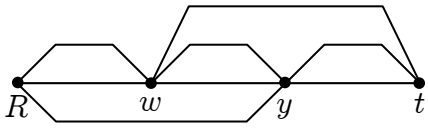


Figure 5.5: Cost 5, graph topology number 30.

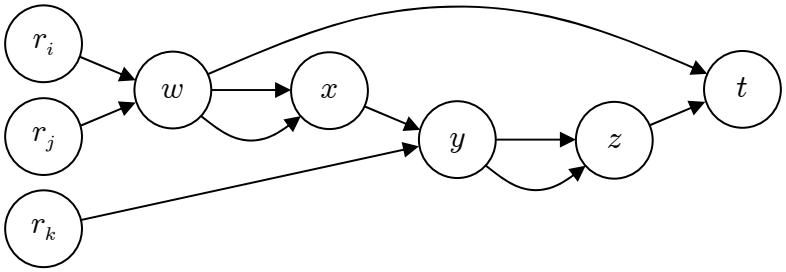


Figure 5.6: Cost 5, graph topology number 30 for testing adder distance.

Consider graph number 30 from Figure 5 in [2]. We show the original graph in Figure 5.5 and the corresponding adder distance DAG is shown in Figure 5.6. Based on the labeling in Figure 5.6, $t \in \mathcal{A}(w, z)$. It follows that $w \in \mathcal{A}(t, z)$ and $z \in \mathcal{A}(t, w)$, but we do not know either w or z. When we illustrated the IGT method for graph number 11, $t \in \mathcal{A}(z, R)$. Since t was constructed with *only one unknown* term z, so the other known term R could be used to deduce all possible values of the unknown term (recall $z \in \mathcal{A}(t, R)$). Unfortunately, we cannot do this in graph number 30.

Leapfrog graphs contain subgraphs that ''leap'' over each other and have a cyclic inter-dependency among the intermediate terms. Notice that w, x, y, and z form a closed loop in Figure 5.6. Leapfrog graphs need at least 4 adders [2]. Although IGT was first introduced in [6], they only considered up to distance 3, thus we are the first to illustrate how to handle cyclic inter-dependencies between the intermediate terms.

In order to use IGT in a leapfrog graph, we have to guess the value of one of the intermediate terms in the loop, use IGT to traverse all the way around the loop, and

then check whether our initial guess was correct. Since $w \in \mathcal{A}(R, R)$, we will do this for each $w \in S$ (each w is tested independently). Note that an element in $\mathcal{A}(R, R)$ may also be in R, but if $w \in R$, then this topology turns into a distance 4 test. We could assert any one of w, x, y, or z, but we choose w because the closer the term is to R, the less possible values it can take.

Knowing t and assuming a value for w, then all of the possible values for z are $z \in \mathcal{A}(t, w)$. Since $z \in C_1 \cdot y$, then $y \in \frac{z}{C_1} = \frac{\mathcal{A}(t,w)}{C_1}$. Clearly, $y \in \mathcal{A}(x, R)$, thus all the possible values of x are $x \in \mathcal{A}(y, R) = \mathcal{A}(\frac{\mathcal{A}(t,w)}{C_1}, R)$. Since $x \in C_1 \cdot w$, it follows that $w \in x/C_1 = \mathcal{A}(\frac{\mathcal{A}(t,w)}{C_1}, R)$ / C_1. Conclusively, t is constructible with this topology iff there exists a $w \in S$ such that $w \in \mathcal{A}(\frac{\mathcal{A}(t,w)}{C_1}, R)$ / C_1. Note that testing for a common element between S and $\mathcal{A}(\frac{\mathcal{A}(t,w)}{C_1}, R)$ / C_1 is *meaningless* due to the inter-dependencies.

Since we have assumed a value for w and we have the relation that $x \in C_1 \cdot w$, instead of checking if $w \in \frac{x}{C_1}$, we could check if $\frac{x}{w} \in C_1$. This is more efficient since we will divide all possible values of x by a single element w instead of by a set of values C_1. Intersection with C_1 is fast because C_1 is sorted (recall that C_1 is precomputed) thus facilitating the use of a binary search. Conclusively, t is constructible with topology number 30 if

$$\frac{\mathcal{A}\left(\dfrac{\mathcal{A}(t,w)}{C_1}, R\right)}{w} \cap C_1 \neq \emptyset \tag{5.5}$$

is satisfied for some $w \in S$, and t is not constructible if no $w \in S$ satisfies (5.5).

Alternatively, we could have traversed around the loop in the opposite direction (assuming a value for w, find all of the possible values for x, then y, then z, and then finally check if $w \in \mathcal{A}(t, z)$). However, traversing in this direction involves multiplication by C_1, which does not prune the search space like division (since we require that elements divide with zero remainder).

Like the cost 5 tests, common sets are shared whenever possible to reduce the computation. In topology number 30, it is possible that multiple values of w could produce the same value of z. We can prevent a redundant searching of terms by only considering unique z. If z can be constructed by multiple w, when we get to the division by w part of the test in (5.5), we divide by *every* w that permitted the construction of z. Let us define the set $Ats = \mathcal{A}(t, S)$. As we construct every possible value of z (which is the set AtS), each z is tagged with the w used. Once AtS is constructed, for each unique z, we can identify all of the supporting w terms. For a given $u \in AtS$, let us define $Ats_d(u) = \{z \mid u \in \mathcal{A}(t, z), z \in S\}$. In other words, for a given $u \in AtS$, $Ats_d(u)$ denotes the set of *dependents* in S that enable u to be constructed. Thus, topology number 30 can construct t if

$$\frac{\mathcal{A}\left(\frac{u}{C_1}, R\right)}{AtS_d(u)} \cap C_1 \neq \emptyset \tag{5.6}$$

is satisfied for some $u \in AtS$, and t is not constructible if no $u \in AtS$ satisfies (5.6).

Note that AtS_d is also used for graph topology number 33. Other commonly used sets include: $AtR = \mathcal{A}(t, R)$, $C1S = C_1 \cdot S$, and $ARS = \mathcal{A}(R, S)$. For a given $u \in C1S$ or a $v \in ARS$, $C1S_d(u)$ and $ARS_d(v)$ denote the set of all elements in S that enable u or v to be constructed, respectively. Like AtS_d, these are sets of *dependents*. In order to remove duplicates and/or facilitate a quick intersection using binary search, we always keep the following sets sorted: R, S, AtS, each AtS_d, $C1S$, each $C1S_d$, ARS, and each ARS_d. In Table 5.13, we provide a summary of the distance 5 tests which search the graph topologies that were not covered by the intrinsic cost 6 tests. Again, each IGT test may search more than one topology, common sets are shared

Table 5.13: A summary of the distance 5 tests for cost 6 which cover the topologies missed by the intrinsic cost 6 tests. Note $AtR = \mathcal{A}(t, R)$, $AtS = \mathcal{A}(t, S)$, $C1S = C_1 \cdot S$, and $ARS = \mathcal{A}(R, S)$. For each $u \in AtS$, $AtS_d(u)$ denotes the set of *dependents* in S that enable u to be constructed ($C1S_d(u)$ and $ARS_d(u)$ have an analogous definition).

Case	For each	Test	Graphs covered in [2]
1	$u \in AtR$	$\frac{u}{C_3} \cap S \neq \emptyset$	2, 3*, 5, 8*, 10*
2		$\frac{u}{C_2} \cap ARS \neq \emptyset$	10, 11
3		$\mathcal{A}\left(\frac{u}{C_1}, R\right) \cap C1S \neq \emptyset$	3
4		$\mathcal{A}\left(\frac{u}{C_1}, R\right) \cap ARS \neq \emptyset$	8
5	$u \in AtS$	$\frac{u}{C_2} \cap S \neq \emptyset$	7*, 9
6		for each $v \in AtS_d(u) : \frac{u}{C_2} \cap \mathcal{A}(v, R) \neq \emptyset$	31*, 32
7		$\frac{u}{C_1} \cap ARS \neq \emptyset$	7
8		$\mathcal{A}\left(\frac{u}{C_1}, AtS_d(u)\right) \cap S \neq \emptyset$	33
9		$\frac{\mathcal{A}\left(\frac{u}{C_1}, R\right)}{AtS_d(u)} \cap C_1 \neq \emptyset$	30
10	$u \in C1S$	$\mathcal{A}(t, u) \cap C1S \neq \emptyset$	12
11		$\mathcal{A}(t, u) \cap ARS \neq \emptyset$	6
12		for each $v \in C1S_d(u) : \frac{\mathcal{A}(t,u)}{C_1} \cap \mathcal{A}(v, R) \neq \emptyset$	33*
13		$\mathcal{A}\left(\frac{\mathcal{A}(t,u)}{C_1}, u\right) \cap R \neq \emptyset$	32*
14	$u \in ARS$	$\mathcal{A}(t, u) \cap ARS \neq \emptyset$	1
15		for each $v \in ARS_d(u) : \frac{\mathcal{A}(t,u)}{C_1} \cap \mathcal{A}(v, R) \neq \emptyset$	4
16		$\mathcal{A}\left(\frac{\mathcal{A}(t,u)}{C_1}, u\right) \cap R \neq \emptyset$	31
17		$\mathcal{A}\left(\frac{\mathcal{A}(t,u)}{C_1}, u\right) \cap ARS_d(u) \neq \emptyset$	34

whenever possible (the dependent sets were deliberately created to facilitate sharing), tests with common sets are done together, and tests are sorted roughly by run time.

Note that some tests by themselves are not efficient, but take advantage of terms that other tests need anyways. For example, case number 2 in Table 5.13 intersects $\frac{\mathcal{A}(t,R)}{C_2}$ with $\mathcal{A}(R,S)$, but it is much more expensive to construct $\mathcal{A}(R,S)$ than to construct $\mathcal{A}(\frac{\mathcal{A}(t,R)}{C_2},R)$ and then intersect it with S. However, case 16 in Table 5.13 is a leapfrog case in which a closed loop is formed between elements that are at least 2 adders away from R. As explained earlier, we assume a value for closest node, use IGT to go around the loop, and then check if the assumption was valid. Since we must test all possible assumed values (one at a time), we must construct $\mathcal{A}(R,S)$ for case 16 anyways, so case 2 takes advantage of this.

5.2.3 Details Specific to the Adder-Operation

Unless the shifts m and n in the adder-operation are bounded, $\mathcal{A}(x,y)\{z \mid z = |2^m x \pm 2^n y|\}$ contains an infinite number of elements. It was proved in [1] that the shift bounds are finite (allowing larger shifts will not decrease the number of adders in the solution). Upper bounds were provided in [1], i.e. optimality is guaranteed if we consider all shifts up to this bound. Nothing was proved on the *lower* bounds of the shifts that are needed to guarantee optimality. Also, [1] predates vertex reduction and [2] did not discuss how the bounds change if we only consider vertex reduced graph topologies. For example, graphs a and b may be identical under vertex reduction, but if we only search graph a, the shift bound to cover both graphs is at least as large as that to cover graph a but it is otherwise unknown. More research is needed to establish the *minimum* bounds on the shifts that are needed to *guarantee* optimality.

The exact adder distance tests in both RAG-n [14] and Hcub [6] use an adder-operation that is constrained to $\mathcal{A}(x, y) \leq 2^{b+1}$, where b is the bit width of the largest constant. The values of x and y implicitly place constraints on the shifts. Recall from section 2.3.3 that there are three cases of shifts to consider: $m > 0$ *and* $n = 0$, $m = 0$ *and* $n > 0$, and $m = n < 0$. In the first case, we must have $2^m x \leq |2^{b+1} \pm y|$, which imposes a limit on m. By symmetry, we will not consider the second case. The third case corresponds to adding x and y and then dividing the result by 2 until it becomes odd. Thus, if both $x \leq 2^{b+1}$ and $y \leq 2^{b+1}$, then $\mathcal{A}(x, y) \leq 2^{b+1}$. Since we initialize $R = \{1\}$, by induction, all terms can be enforced to be no larger than 2^{b+1}.

One may be tempted to enforce $\mathcal{A}(x, y) \leq 2^{b+1}$ and claim that an exhaustive search based on the tests in section 5.2.2 is optimal *under the constraint that all intermediate terms are no larger than* 2^{b+1}. However, this is not true, and a specific example is provided in section 6.1.3.5. When vertex reduction is used, we can miss some solutions that would otherwise be valid by combining terms *in the wrong order*.

Initially we arbitrarily chose to enforce $\mathcal{A}(x, y) \leq 2^{b+2}$ in the BIGE algorithm. We later experimented with bounds of 2^{b+1} and 2^{b+3}. Our results indicate that changing the bound has an extremely negligible impact on the number of adders in the solutions (details are provided in section 5.2.4). At 32 bits, the average adder cost decreases (increases) on order of 10^{-4} by incrementing (decrementing) the bounding exponent by 1. While the BIGE algorithm is not truly optimal *from a theoretical perspective*, we argue it is close enough to optimal for all practical purposes. Furthermore, each increment in the bounding exponent causes about 20% increase in the run time. Recall from section 5.2.1.2 that part of the motivation for creating the BIGE algorithm was to further reduce the value of k used in the H(k)+ODP algorithm. The BIGE

Input : odd and positive integers x and y, the bounding value 2^k
Output: $\mathcal{A}(x, y)$

1 $tmp = x + y$
2 **do** { $tmp = tmp/2$ } **while** (tmp is even)
3 add element tmp to the set $\mathcal{A}(x, y)$
4 $tmp = |x - y|$
5 **do** { $tmp = tmp/2$ } **while** (tmp is even)
6 add element tmp to the set $\mathcal{A}(x, y)$
7 $i = 1$
8 **while** ($((x \ll i) + y) \leq 2^k$) {
9 add element $(x \ll i) + y$ to the set $\mathcal{A}(x, y)$
10 add element $|(x \ll i) - y|$ to the set $\mathcal{A}(x, y)$
11 $i = i + 1$
}
12 **while** ($|(x \ll i) - y| \leq 2^k$) {
13 add element $|(x \ll i) - y|$ to the set $\mathcal{A}(x, y)$
14 $i = i + 1$
}
15 $i = 1$
16 **while** ($((y \ll i) + x) \leq 2^k$) {
17 add element $(y \ll i) + x$ to the set $\mathcal{A}(x, y)$
18 add element $|(y \ll i) - x|$ to the set $\mathcal{A}(x, y)$
19 $i = i + 1$
}
20 **while** ($|(y \ll i) - x| \leq 2^k$) {
21 add element $|(y \ll i) - x|$ to the set $\mathcal{A}(x, y)$
22 $i = i + 1$
}
23 return $\mathcal{A}(x, y)$

Algorithm 2: A computationally efficient method for computing the adder-operation subject to $\mathcal{A}(x, y) \leq 2^k$.

algorithm is sometimes *faster* than H(k)+ODP. We believe 20% run time is more important than producing a worse solution on the order of once in every 10^4 cases.

In Algorithm 2, we provide the pseudo-code used by all of our algorithms any time the adder-operation is computed (the BIGE algorithm uses $k = b + 2$). Note that values are not always stored. For example, as we generate each element, we may check for membership in a set and then discard the element. To minimize storage, we typically compute the tests in Tables 5.11, 5.12, and 5.13 in a depth-first manner.

One special case is to test whether $\mathcal{A}(x, y) \cap \{1\} \neq \emptyset$. Let $odd(n) = odd(n/2)$ if n is even and $odd(n) = n$ if n is odd. Given odd and positive integers x and y, $\mathcal{A}(x, y) \cap \{1\} \neq \emptyset$ is satisfied iff at least one of the following are satisfied: $odd(x+1) = y$, $odd(x - 1) = y$, $odd(y + 1) = x$, $odd(y - 1) = x$, $odd(x + y) = 1$, or $odd(|x - y|) = 1$. We exploit this property in cases 5, 6, and 9 of the cost 5 tests (from Table 5.11).

5.2.4 Experimental Results

As stated in section 5.2.3, we can tradeoff run time for quality of solution by changing the bound on the largest intermediate term to consider. The results of our benchmark were obtained by enforcing $\mathcal{A}(x, y) \leq 2^{b+2}$. We later experimented with different bounds, but since the results changed by a negligible amount, repeating the entire benchmark is pointless (also, our results have experimental error due to the use of random constants). For example, the average number of adders and average run times (in seconds) at 32 bits are respectively: 5.87494 and 7.457 with a bound of 2^{b+1}, 5.87471 and 8.851 with a bound of 2^{b+2}, and 5.87442 and 10.052 with a bound of 2^{b+3}. For each increment in the bounding exponent, the run time increases roughly 20% and the number of adders decreases by about $3 \cdot 10^{-4}$. Although not shown, the

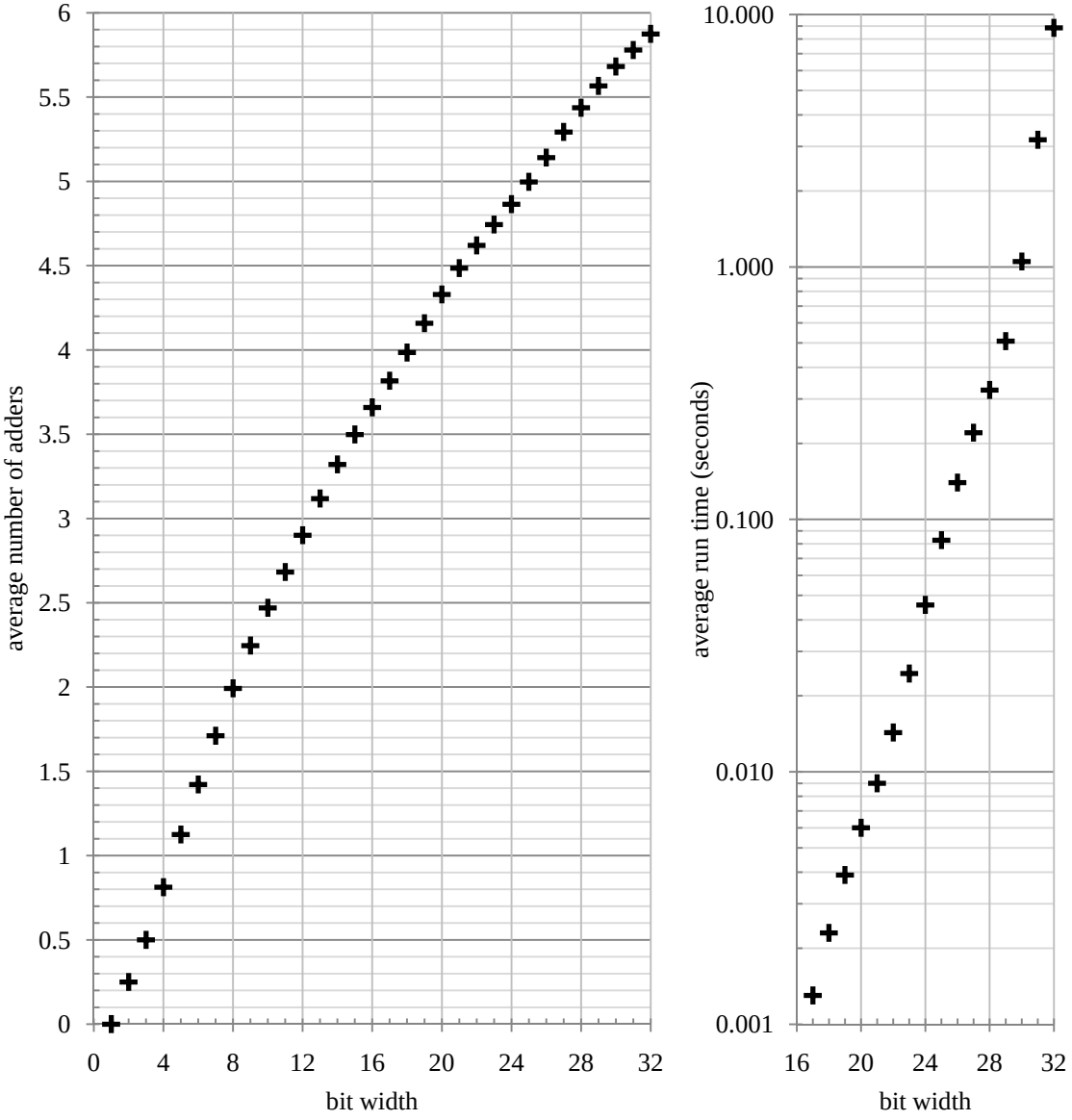


Figure 5.7: The average number of adders and average run time of the BIGE algorithm. The average run time is less than 1 millisecond for bit widths up to 16.

Table 5.14: Bit width versus the sizes of the C_n sets and the total run time (in seconds) required to compute all 4 sets. The run time is an average of 10 trials. Given a SCM constant of bit width b, the BIGE algorithm uses C_n sets that contain all elements up to 2^{b+1}.

Bit width of the SCM constant	C_1 size	C_2 size	C_3 size	C_4 size	Total run time
12	24	279	2083	1710	0.02
16	32	559	6172	24718	0.16
20	40	935	20617	333041	0.73
24	48	1407	41431	1133053	2.37
28	56	1975	72789	2798831	5.64
32	64	2639	116819	5769889	13.02

number of adders changes by even less as the bit width decreases. Since the number of adders is an integer, this means that we will find a worse solution roughly 3 times in every 10^4 cases for each increment. The minimum bound to guarantee optimality is *at least* 2^{b+3} and more research is needed to establish it. However, for practical purposes, using a smaller bound leads to good solutions in less run time.

We implemented the BIGE algorithm in C. As mentioned in section 5.1.6, all experiments were run on 3.06 GHz Pentium 4 Xeon workstations running Linux. We implemented the MAG algorithm in C to precompute the C_n sets for $n = 1, 2, 3, 4$. For constants up to 20 bits, we used *all* constants. At each bit width at or above 21 bits, we used 100000 uniformly distributed random constants (the same constants were used in the comparison with the heuristics in section 5.1.6 for bit widths 20, 24, 28, and 32). The average number of adders and the average run time are shown in Figure 5.7. Due to the exhaustive nature of the BIGE algorithm, the average run time appears to grow exponentially. Our results suggest that the average number of adders increases in a sublinear manner with respect to the bit width. The authors of the MAG algorithm [1] were the first to observe this trend. They explain that there

Table 5.15: Smallest numbers that require a specific part of the BIGE algorithm.

Adders / Method	Number	Bit Width
1 / lookup table	3	2
2 / lookup table	11	4
3 / lookup table	43	6
4 / lookup table	683	10
5 / H(1)+ODP	14709	14
5 / graphs	349725	19
6 / H(1)+ODP	699829	20
6 / H(2)+ODP	23242013	25
6 / intrinsic graphs	43667339	26
6 / non-intrinsic graphs	79269941	27
7 / H(2)+ODP	224227085	28

are many more ways to share intermediate terms as the number of adders increase, so it becomes increasingly likely that a solution better than the CSD cost exists. The average CSD cost grows linearly with respect to the bit width [4], and as computed in [2], there are 2, 5, 15, 54, 227 unique vertex reduced graph topologies for SCM solutions with 2, 3, 4, 5, and 6 adders, respectively.

The sizes of the C_n sets are shown in Table 5.14, along with the average run time (over 10 trials) needed to compute the sets. In Table 5.15, we show the smallest constant that uses each part of the BIGE algorithm. For example, since we search in the order of increasing adder cost, the first time that H(1)+ODP is used is for the smallest number that requires 5 adders, which is 14709. We try a heuristic before an exhaustive search, thus from Table 5.15, the first case in which H(1)+ODP cannot find a solution with 5 adders is 349725. Since random constants are used for bit widths of 21 and above, some of the values in Table 5.15 are estimates.

Finally, in Table 5.16, we examine what percent of the solutions are found by each part of the BIGE algorithm as well as the average run time for each part. As shown

Table 5.16: Detailed distributions for each part of the BIGE algorithm.

(a) What percent of the solutions were produced by each part of the BIGE algorithm.

Adders / Method	16 bits	20 bits	24 bits	28 bits	32 bits
0 / lookup table	0.026	0.002			
1 / lookup table	0.343	0.034	0.004		
2 / lookup table	3.456	0.480	0.040	0.013	
3 / lookup table	28.238	6.464	1.022	0.161	0.010
4 / lookup table	65.768	52.592	17.381	3.525	0.579
5 / H(1)+ODP	2.168	40.358	67.860	38.881	11.727
5 / graphs		0.068	7.692	9.880	3.563
6 / H(1)+ODP		0.0001	6.001	46.370	60.292
6 / H(2)+ODP				0.912	7.254
6 / intrinsic graphs				0.096	2.602
6 / non-intrinsic graphs				0.161	10.024
7 / H(2)+ODP				0.001	3.949

(b) The average run time (in seconds) of each part of the BIGE algorithm.

Adders / Method	16 bits	20 bits	24 bits	28 bits	32 bits
0 / lookup table	<0.001	<0.001			
1 / lookup table	<0.001	<0.001	<0.001		
2 / lookup table	<0.001	<0.001	<0.001	<0.001	
3 / lookup table	<0.001	<0.001	<0.001	<0.001	<0.001
4 / lookup table	0.001	0.002	0.005	0.012	0.023
5 / H(1)+ODP	0.007	0.012	0.022	0.036	0.055
5 / graphs		0.034	0.114	0.190	0.284
6 / H(1)+ODP		0.190	0.353	0.585	0.921
6 / H(2)+ODP				1.437	2.343
6 / intrinsic graphs				1.918	3.527
6 / non-intrinsic graphs				3.152	14.924
7 / H(2)+ODP				86.290	165.140

in Table 5.16(a), the bounding heuristic typically produces most of the solutions returned by the BIGE algorithm. This was expected since we knew that H(k)+ODP is nearly optimal *before* we created the BIGE algorithm. Thus in most cases, we only need to exhaustively search up to $n-1$ adders in order to show a solution with n adders is optimal. From Table 5.16(b), exhaustively searching 6 adders requires about 3 minutes at 32 bits whereas more than 60% of the solutions at 32 bits have 6 adders and are found in less than one second on average. The exhaustive solution space explodes with each extra adder, thus not having to search the last adder is very beneficial. Note we exhaustively search 6 adders to prove that a 7 adder solution is optimal. When solving the SCM problem *for the given target*, the BIGE algorithm is much more computationally efficient than the MAG algorithm. The MAG algorithm must exhaustively search up to the last adder and it does not use the target to prune the search for useful intermediate terms.

5.3 Minimization of Single-Bit Adders

In this section, we will use single-bit adders to estimate of the amount of logic needed to implement a constant coefficient multiplier. This is a more accurate metric than the number of additions or subtractions (less abstraction), thus we expect better solutions. We will begin by examining how shifts affect the number of single-bit adders. In section 5.3.2, we propose an optimal algorithm that first minimizes the number of adder-operations. Among the equally good solutions, it keeps the one(s) with the minimum number of single-bit adders. The minimum adder depth is used to break any remaining tie (beyond this, the choice is arbitrary). Experimental results and a discussion of the results are presented in section 5.3.3.

5.3.1 Single-Bit Adders

We will assume ripple-carry adders are used since our objective is to minimize the amount of logic resources. As a case study, we will only consider single-bit full adders in FPGAs. In FPGAs that use 4-input look-up tables (LUTs) as the programmable logic elements, each single-bit adder requires 1 LUT (in most FPGAs, the carry chain is passed through adjacent LUTs in a different manner than how logic elements are typically used). The derivations that we will provide can easily be modified for single-bit half adders in ASICs, for example.

Assume x is representable on b bits, then $c \cdot x$ can be represented on $\lceil \log_2 |c| \rceil + b$ bits, where c is an odd integer constant. If $c \cdot x$ has the form $c \cdot x = (d \cdot x) \pm ((e \cdot x) \ll n)$ where d and e are odd integer constants, notice that the n least significant bits of $d \cdot x$ are added with the zeros that serve as placeholders in $(e \cdot x) \ll n$. Thus, the least significant n bits of $c \cdot x$ incur no cost, so we only need $\lceil \log_2 |c| \rceil + b - n$ single-bit adders. However, if we subtract the *non-shifted* term (i.e. $c \cdot x = ((e \cdot x) \ll n) - (d \cdot x)$), then all $\lceil \log_2 |c| \rceil + b$ single-bit adders are needed, as negating the least significant n bits of $d \cdot x$ does not come for free. The only other case to consider is $c \cdot x = ((d \cdot x) \pm (e \cdot x)) \gg n$. Since $(c \cdot x) \ll n$ requires $\lceil \log_2 |c| \rceil + b + n$ bits to be represented, $(d \cdot x) \pm (e \cdot x)$ requires $\lceil \log_2 |c| \rceil + b + n$ single-bit adders. The n least significant bits of $(d \cdot x) \pm (e \cdot x)$ are necessarily zero, but we still need $\lceil \log_2 |c| \rceil + b + n$ LUTs in an FPGA because the n least significant bits may produce a carry which affects the final answer $c \cdot x$.

It is favorable to add or subtract the left-shifted term and unfavorable to use right shifts. If all terms are positive and the subtracted term is left-shifted, we will likely produce a negative number (the left-shifted term is typically larger than the non-shifted term *after considering shifts*). By allowing negative numbers in the search,

we facilitate the use of subtraction in such a way that less single-bit adders are needed, so we can obtain better solutions. For example, with only positive numbers, there is only one solution for $105x$ that uses the minimum number of adder-operations: $105x = ((7x) \ll 4) - (7x)$, where $7x = (x \ll 3) - x$. It follows that $7x$ requires $b + 3$ single-bit adders and $105x$ requires another $b + 7$ single-bit adders. Notice that we can propagate the subtraction through the solution: $105x = (-7x) - ((-7x) \ll 4)$, where $-7x = x - (x \ll 3)$. In this case, $-7x$ costs $b + 3 - 3 = b$ single-bit adders and $105x$ requires another $b + 7 - 4 = b + 3$ adders, thus 7 single-bit adders were saved.

We will consider negative terms and thus we will find better solutions at the expense of more run time (obviously the solution space is larger with negative terms). Clearly, a different definition of the adder-operation is needed (we use (2.2) from section 2.3.3) and the C_n sets must be adjusted accordingly (their definitions in terms of adder-operations were provided in section 5.2.2.2). Note that $5 \in C_1$ whereas $-5 \notin C_1$ since we are not allowed to use *two* negations in one adder-operation (i.e. $-5x = -(x \ll 2) - x$ costs 2 adder-operations). As another example, $-3 \in C_1$.

5.3.2 An Exhaustive Search

5.3.2.1 Justification for First Minimizing Adder-Operations

We name the exhaustive algorithm in this section the single-bit adder cost (SBAC) algorithm. It first minimizes the number of adder-operations, *then* the number of single-bit full adders, and *then* the adder depth (beyond this, the choice is arbitrary).

The SBAC algorithm is not optimal in terms of minimizing single-bit adders, however we conjecture that it is unlikely that a solution with more adder-operations will have fewer single-bit adders than the *best* solution among all of the ones that have

the minimum number of adder-operations. If we directly tried to minimize the number of single-bit adders, we would need to search for solutions with more adder-operations. In addition to not knowing where to stop, the solution space grows extremely fast with respect to the number of adder-operations.

To maintain practical amounts of computation, the SBAC algorithm searches in the order of increasing adder-operation cost, so the first solution found determines the maximum number of adder-operations that we will consider. Thus, *every* solution found by SBAC will have the same minimum number of adder-operations, so the single-bit adder cost is actually the first metric that we *compute*.

5.3.2.2 Less Pruning than the BIGE Algorithm

If the best solution requires n adder-operations, the SBAC algorithm must search the *entire* solution space of up to n adder-operations. Unlike the BIGE algorithm from section 5.2, SBAC cannot stop after the first solution is found because there could be another solution with same number of adder-operations but with fewer single-bit adders. A heuristic bound is also useless. Even if a heuristic happened to find a solution with the minimum number of adder-operations, the SBAC algorithm still has to search the *entire* solution space of up to n adders, thus the exhaustive search will eventually find the minimum number of adder-operations. The BIGE algorithm only finds *one* solution with the minimum number of adder-operations and this is often found by the bounding heuristic (as shown in Table 5.16). In this case, the exhaustive search was only performed up to $n - 1$ adder-operations.

Minimizing the adder depth is only considered when the current solution found has exactly the same number of adder-operations *and* single-bit adders as the best

solution found so far (we keep track of the best solution as the search is done). We illustrate how to compute the adder depth in section 6.2.1. Since the depth is not a *constraint* in SBAC, the depth cannot be used prune the search space (this type of pruning is used in the algorithm in section 6.2).

Clearly, SBAC will require more run time than the BIGE algorithm. However, the number of single-bit adders is a more accurate estimate of the amount of logic resources compared to the number of additions and subtractions. If we were to measure the amount of silicon used in the logic circuit (which is the absolute metric), we would expect SBAC to produce better results than the BIGE algorithm. Hence, we are trading run time for better solutions. In order to maintain reasonable run times, this can only be done for small problem sizes. We will consider up to 24 bit constants in our benchmark.

5.3.2.3 Following the Solution Towards Construction

Like the BIGE algorithm, in order to establish whether the SCM target needs m adder-operations, we apply distance k tests to all possible sets of R with $m-k+1$ elements. In the SBAC algorithm, up to 6 adders are considered, k is limited to 4 and we will *vary* k, and the C_n sets are needed for $n = 1, 2, 3$. Distance 2, 3, and 4 tests are discussed in sections 3.2.6, 6.1.3.1, and 6.1.4, respectively. Some tests involve division by C_n, but each element in C_n could be constructed in more than one way. Also, since we use vertex reduction, it may be possible to combine terms in a different order so that the number of single-bit adders is reduced. Instead of enumerating every variant of the solution found by a distance k test, we *follow the solution towards construction.*

When we apply all of the distance k tests, we either prove that t cannot be constructed with k more adders or we find *all* of the useful successors that lead to the construction of t in k more adders. For $k \leq 4$, each distance k test either performs an intersection with the successor set S or uses guess and check where we must select each element in S one at a time (the latter case is for leapfrog graphs). Assume a solution with k more adders exists. For each useful s (we will do the following *one useful s at a time*), we construct the useful s (add it to R) and then apply distance $k-1$ tests to find all of the useful successors at distance $k-1$. We are guaranteed to find a solution at distance $k-1$ since we already found one of the solutions at distance k. This can be applied recursively until we get to distance 1. We call this process *following the solution towards construction*. Because the distance k tests find *all* of the useful successors, this process can be used to find *all* of the solutions. To minimize memory, we compute this process in a depth-first manner (i.e. we traverse all the way to distance 1 first instead of processing each useful s at distance k first).

5.3.2.4 Reordering Solutions Before Evaluating the Single-Bit Adder Cost

Now that we can generate all of the possible solutions with the minimum number of adder-operations, we must evaluate the single-bit adder cost of each solution. When we generate all of the possible sets of R with a certain number of elements, we avoid redundant R (for example, we will construct one of $R = \{1, 3, 5\}$ or $R = \{1, 5, 3\}$, not both). A method for generating R without redundancy is shown in detail in section 6.3.3.3. For a given solution, it is therefore possible that by constructing the terms in the solution in a different order, we may be able to use fewer single-bit adders. Even if redundant R were allowed, we would still end up processing all of the same

solutions with the terms in a different order (due to the *completeness* of the search), thus we may as well prevent redundant R to reduce the run time.

For each solution P that we find, we try to construct the elements in P in every possible order. In other words, for each $p \in P$, we check if p can be constructed using the terms constructed so far (the element 1 is always initially constructed for free). If p can be constructed, we remove p from P (i.e $P_{new} = P \backslash \{p\}$) and then repeat the process for P_{new}. This recursive process is done in a depth-first manner and will produce *all* of the valid solutions that can be obtained by permuting the elements in the original P. For each valid solution, we can evaluate the single-bit adder cost using the approach illustrated in section 5.3.1. In order to determine if p can be constructed with the terms constructed so far (let R' denote the terms constructed so far), we test if $\mathcal{A}(p, R') \cap R' \neq \emptyset$. This is a functionally equivalent but more computationally efficient test to determine if $p \in \mathcal{A}(R', R')$ (this was explained in section 3.2.2).

5.3.3 Experimental Results

We implemented the SBAC algorithm in C. All experiments were run on a 3.06 GHz Pentium 4 Xeon workstation. Only up to 6 adders were considered, but solutions were found for all of the tested cases. We only tested up to 24 bits, which is sufficient to implement multiplication by a single-precision floating point number (the mantissa is 24 bits, as 23 bits are stored and 1 bit is implied).

We compared the SBAC and BIGE algorithms using the same constants. The average number of single-bit adders is shown in Figure 5.8 and all of the results are shown in Table 5.17. Both algorithms enforce $-2^{b+2} \leq \mathcal{A}(x, y) \leq 2^{b+2}$, as discussed in section 5.2.3 (SBAC needs a lower bound since it accepts negative targets and

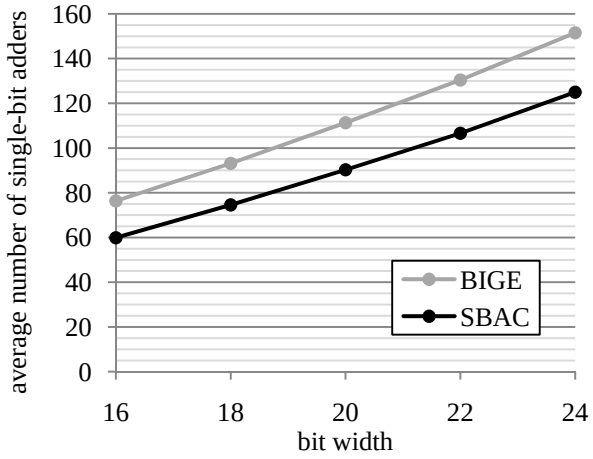


Figure 5.8: The average number of single-bit adders versus bit width.

may use negative integers as intermediate terms regardless of the sign of the target). At each bit width, 10000 uniformly distributed random constants were used. Only unsigned constants were used in the benchmark (as required by the BIGE algorithm). Although not shown, the SBAC algorithm produces similar average results for both positive and negative constants of the same *signed* bit width, i.e. a 16 bit unsigned constant requires 17 bits in signed binary (2's complement). The *absolute* number of single-bit adders depends on the bit width of the multiplicand (the value of b in section 5.3.1). We assume the SCM constant and the multiplicand have the same bit width, thereby providing absolute numbers in Figure 5.8 and Table 5.17. Given that all solutions have n adder-operations, there will be a common factor of $n \cdot b$ single-bit adders, so we really only needed to compare the *relative* number of single-bit adders.

As expected, both algorithms find solutions with the same number of adder-operations and SBAC finds better single-bit adder solutions at the expense of more run time. From Table 5.17, the number of single-bit adders typically decreased by about 25% whereas the run time increased by 2-3 orders of magnitude. The worst case run time of SBAC drastically increases at 22 bits since we must now search at cost 6.

Table 5.17: The experimental results of the SBAC and BIGE algorithms.

		16 bits	18 bits	20 bits	22 bits	24 bits
Adder-operations (average)		3.66	3.99	4.33	4.61	4.87
Adder-operations (worst case)		5	5	5	6	6
Single-bit adders (average)	SBAC	59.85	74.56	90.28	106.56	124.96
	BIGE	76.32	93.14	111.30	130.43	151.52
Single-bit adders (worst case)	SBAC	108	120	136	162	192
	BIGE	133	156	178	207	254
Adder depth (average)	SBAC	3.63	3.95	4.26	4.53	4.77
	BIGE	3.64	3.84	3.95	4.04	4.19
Run time (seconds) (average)	SBAC	0.08	0.46	1.25	6.75	111.74
	BIGE	<0.01	<0.01	<0.01	0.01	0.05
Run time (seconds) (worst case)	SBAC	4.24	5.19	6.23	1305.43	2309.29
	BIGE	0.03	0.09	0.25	0.63	1.10

From Table 5.17, the SBAC algorithm typically produces solutions with a larger adder depth than the BIGE algorithm. In the SBAC algorithm, the adder depth is only considered when there are multiple best solutions in terms of adder-operations and single-bit adders, thus we expect it to have little impact. Our results illustrate the inherent tradeoff between minimizing the amount of logic resources and optimizing the logic circuit for speed. These objectives can be estimated by the number of single-bit adders and the adder depth, respectively. For SCM, the adder depth has little practical importance (clearly, it can be no larger than the number of adder-operations). For high-throughput circuits, registers can be placed between each addition or subtraction (by reducing the register-to-register delay, the circuit can be clocked faster). In this case, minimizing single-bit adders will also minimize the number of registers. If the throughput of the constant coefficient multiplier is not on the critical path of a system-wide task, it makes little difference whether 4 or 5 adders are placed serially. Conversely, MCM solutions may have a much larger adder depth since we can build targets off of other targets (we discuss the adder depth for MCM in section 6.2).

5.4 Concluding Remarks on SCM

In section 5.1, we analyzed and proposed solutions to simplest cases of the CSE digit clashing problem. Compared to H(k), the best existing SCM heuristic, our proposed algorithm H(k)+ODP on average produces better solutions (by considering ODPs) in significantly less run time (by using smaller k, the search space is much smaller). In section 5.2, we used an exhaustive search to either find the optimal solution or verify the optimality of a heuristic solution. We obtain optimal solutions, and sometimes in less run time than H(k)+ODP (increasing k to get a better solution is futile if we know the existing solution is optimal). Finally, in section 5.3, we used single-bit adders to obtain a more accurate minimization of the amount of logic resources, but at the expense of a significant increase in the run time.

Constants are typically represented with no more than 32 bits in the problems of the most practical importance in digital signal processing. We can optimally solve the SCM problem for constants up to 32 bits in less than 10 seconds on average. For all practical purposes, we have essentially closed the SCM problem. For constants up to 24 bits, we can provide even better solutions by considering the single-bit adder cost (with an average run time of less than 2 minutes). We have also mentioned the idea of using a parallel search to improve the run time and/or perform larger searches, however the details of such are beyond the scope of the thesis and are left as future work. In other applications that involve multiplication by very large constants (such as cryptography), exhaustive optimal approaches are infeasible, but it may be possible to reuse the heuristic strategies developed for H(k)+ODP. Better solutions can be obtained by partially resolving the CSE digit clashing problem.

Chapter 6

New MCM Algorithms

We will propose several new algorithms for solving the multiple constant multiplication problem in this chapter. When designing a MCM heuristic, one can exploit the redundancy *within* each constant as well as the redundancy *between* constants. In section 6.1, we will analyze the implications and the tradeoffs due to favoring one type of redundancy over the other. Within Hcub, one of the best existing MCM heuristics, we will identify several aspects that can be modified to better exploit the redundancy *within* each constant. Also in section 6.1, we propose our heuristics H3 and H4. Compared to Hcub, H3 finds better solutions in less run time and H4 finds significantly better solutions but at the expense of much more run time (although the *absolute* run time is tolerable for moderate problem sizes). In section 6.2, we introduce a depth-constrained version of H3 and discuss the implications of depth constraining. In section 6.3, we propose a depth-first optimal MCM algorithm. Unlike all of the existing exhaustive searches, we are able to use pruning even though there are *multiple* constants. Finally, concluding remarks are provided in section 6.4.

6.1 Heuristic MCM

We will begin this section by analyzing the strengths and weaknesses of the DAG and CSE frameworks in terms of their ability to find redundancy *within* each constant and redundancy *between* constants. In section 6.1.1, we will examine how the underlying framework of an algorithm can determine its ability to find certain types of redundancy. We will use H(k)+ODP, Hcub, and DiffAG as case studies. In section 6.1.2, we will identify the components within Hcub that can be easily modified to better exploit the redundancy *within* each constant. We propose the H3 algorithm in section 6.1.3, along with a very detailed analysis of the modifications from Hcub. In section 6.1.4, we propose H4, a variant of H3 that produces better solutions at the expense of more run time (even so, the *absolute* run time is still tolerable for the problem sizes in which we designed H4 to perform well). Experimental results are provided in section 6.1.5. In order to analyze the results, in section 6.1.6, we will introduce the idea of *differential* adder distance and illustrate that DiffAG is a special case of this more generalized notion. Finally, in section 6.1.7, we propose the H3+DiffAG hybrid algorithm, which is the best performing generalized MCM heuristic (i.e. it performs well over the *entire* spectrum of MCM problem sizes).

6.1.1 An Analysis of Redundancy Within Constants Versus Redundancy Between Constants

6.1.1.1 An Introduction

Redundancy refers to *how much* a term can be reused within a solution to the constant multiplication problem. For example, we can use two instances of $3x$ by sharing it

between $11x = (3x) + (x \ll 3)$ and $35x = (3x) + (x \ll 5)$ (for a MCM problem with $T = \{11, 35\}$) or by sharing it *within* $45x = ((3x) \ll 4) - (3x)$ (for a SCM problem with $T = \{45\}$). As the number of constants increases (assuming the bit width is fixed), there will generally be more cases in which a target can be built off of other targets, thus there is more redundancy *between* constants which can be exploited. For a fixed number of constants, generally more adders are needed as the bit width increases, thus there is more redundancy *within* each constant that we can exploit.

MCM algorithms search for *both* types of redundancy, however, as verified by our results in section 6.1.5, better solutions are obtained if the heuristic is fine-tuned to the characteristics of the given MCM problem instance. For example, given a MCM problem with many constants represented on a small bit width, we are likely to obtain a better solution by using a heuristic that favors redundancy between constants more than it favors redundancy within constants. If the reverse conditions arise, we should favor the other type of redundancy.

In the following subsections, we will analyze how well the CSE and DAG framework can search for each type of redundancy in a *qualitative* manner.

6.1.1.2 Analysis of the CSE Framework

As stated in section 5.1.1.1, CSE provides a simple means to collect common terms once we have decided which signed powers of two will be used to construct the constant (the initial SD form). H(k)+ODP is nearly optimal, which suggests that CSE is highly efficient at finding redundancy *within* a constant. In order to determine how useful a new term is (which is represented by a pattern), we simply need to count how many times the pattern occurs in the CSE form of the constant. A higher count means that

	$A\,0\,A\,0\,0\,0\,A\,0\,A$	$325 \times A$
$+$	$A\,0\,0\,A\,0\,0\,\overline{A}\,0\,0\,\overline{A}$	$567 \times A$
	$A\,A\,A\,0\,0\,0\,\overline{A}\,A\,0\,0$	$(223 \times A) \ll 2$
	$A\,0\,0\,\overline{A}\,0\,0\,0\,0\,\overline{A}\,0\,0$	$(223 \times A) \ll 2$ recoded to CSD

Figure 6.1: An illustration of CSE-related problems.

the term is highly redundant and therefore is more beneficial in terms of minimizing adders. However the occurrence count may be hindered (made lower) by the CSE digit clashing problem or by selecting the wrong initial SD form.

In SCM, the solution typically has only a few adders. With only a few iterations of substitution, the patterns are composed of only a few original CSE digits. For example, let A denote an original CSE digit (in the initial SD form), let $C = B000B$ and $B = A0\overline{A}$, then C represents *only four* digits of A. Due to the limited problem size in SCM, we generally do not encounter these problems associated with CSE (or they are easily resolvable with ODPs and/or SD forms with more nonzero digits).

The solution to a MCM problem typically contains many more adders than in SCM, thus we can encounter CSE-related problems that are much more difficult to resolve (for example, *several* digits from two patterns could align). For instance, in a MCM problem, assume we have already constructed the terms $325 \times A$ and $567 \times A$ and we need to construct $223 \times A$. This can be done as $223 \times A = ((325 \times A) + (567 \times A)) \gg 2$. If we use the CSD form to represent the constants as shown in Figure 6.1, notice that we have digits canceling, digits adding to produce a carry one position to the left, and a recoding of digits due to digits being place adjacently. Furthermore, there is *zero relative shift* between $325 \times A$ and $567 \times A$, which cannot be represented by a CSE substitution since only one digit can occupy each location in the CSE representation (this was explained in detail in section 3.1.2).

As mentioned in section 4.6, the CSE-based algorithm in [37] uses addition and subtraction to *partially* resolve this problem. The only case considered in [37] was the construction of a target by adding or subtracting two existing terms. However, we may also need to create a useful intermediate term this way and this term may not appear in the CSE form of the constant. From the previous example, $325 \times A$ was a useful intermediate term for $223 \times A$, but $325 \times A$ did not appear in $223 \times A$ and $325 \times A$ may need to be constructed by adding two existing terms in a way that would cause problems in CSE. Applying this argument recursively, the workaround (using addition and subtraction) essentially uses the DAG framework.

In conclusion, CSE is good for finding redundancy *within* a constant (as demonstrated by H(k)+ODP), but relatively poor at finding redundancy *between* constants due to clashing and similar problems. In order to facilitate the sharing of terms *between* constants, it is more practical to use a representation that inherently does not have these problems (such as DAGs).

6.1.1.3 Analysis of the DAG Framework

Graphs have the advantage of not having any restrictions, such as allowing a zero relative shift between terms. Also, no obfuscation of terms arises due to the representation. Thus, one can use graphs to perform an exhaustive search, as done in sections 4.1 and 5.2.2. For small adder distances, computing the exact adder distance is feasible since we only need to test a few small graph topologies. However, it is impractical to use exact distance tests for large distances because the number of graph topologies increases very quickly with respect to the number of adders. Also, each test would require more computation since we must traverse through more nodes.

In a MCM problem, the upper bound on the number of adders needed to construct each target t is its SCM cost. By taking advantage of the redundancy *between* targets, we can reuse terms that were needed for other targets anyways, thus it may be possible to construct t with *less* adders than its SCM cost. In most cases, targets are typically much closer than their SCM cost in terms of adder distance. For example, the average optimal SCM cost at 16 bits is about 3.65 adders (see section 5.2.4) whereas for MCM problems with 16 constants on 16 bits, Hcub produces solutions with an average of about 27 adders (see section 6.1.5). The heuristic in Hcub is largely determined by the closest targets due to the $10^{-\text{dist}(R \cup \{s\}, t)}$ term in the weighted benefit function (see (4.8) in section 4.3.3). With an average of 1.7 adders per target, it is very feasible to use graphs to determine how a target can be built off of other targets and/or off of intermediate terms that are needed by other targets anyways. Conclusively, graphs are generally good for exploiting the redundancy *between* targets.

When we search for redundancy *within* a constant, we are looking for *several instances of the same term*. This is done easily in CSE, as we simply count how many times each pattern occurs. For example, in $A0\overline{A}00\overline{A}0A0A00A0\overline{A}$, we can substitute $B = A0\overline{A}$ three times to get $00B00\overline{B}0A0000B$. The equivalent graph topology is shown in Figure 6.2 and requires 4 adders. It follows that the test for this topology is $\mathcal{A}(t, 1) \cap C_3 \neq \emptyset$. Let c_3 denote the common element between $\mathcal{A}(t, 1)$ and C_3, then c_3 in the DAG framework corresponds to $00B00\overline{B}000000B$ in the CSE framework (notice this is missing the A since we adder-operated t with 1). The term B in the CSE framework corresponds to the *supporting term* which enabled c_3 to be constructed in the DAG framework. This supporting term is in C_1. We try to remove one nonzero digit in the CSE form by adder-operating t with 1 in the DAG test. However the

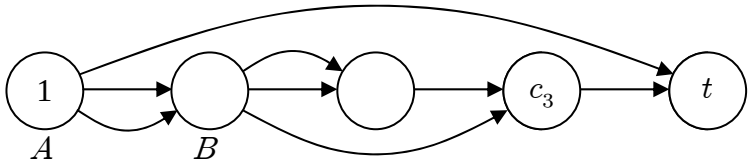


Figure 6.2: The equivalent graph topology for $00B00\overline{B}0A0000B$. The target t is composed of 3 instances of B and 1 instance of A. The test is $\mathcal{A}(t,1) \cap C_3 \neq \emptyset$.

DAG test tries *every possible placement* of the nonzero digit. In CSE, we are *able to see* what terms are useful whereas we must guess and check for useful terms in the DAG framework. In general, in order to find *many* instances of a term with DAGs, we would need to search *several large* graph topologies, which would require a significant amount of computation. Searching for CSE patterns is much easier.

In conclusion, graphs are good for exploiting the redundancy *between* constants. Constants are typically quite close together in terms of adder distance, so an exhaustive search is feasible (for small adder distances) and has the advantage of not missing any useful terms. In order to exploit the redundancy *within* a constant, graphs can still be used for small adder costs, however it becomes more computationally efficient to use CSE as the number of adders increases.

6.1.1.4 Analysis of the DiffAG and Hcub Algorithms

We are only concerned with the *heuristic* part of the algorithms (both DiffAG and Hcub reuse the optimal part of RAG-n). On each iteration that the heuristic part of DiffAG is used, one term in one of the difference sets $D_{i,j}$ is selected. If this term is in S, we construct a new term, otherwise we add a new target to T'. Once any term in $D_{i,j}$ is constructed, then constructing *any* term in node N_i *or* node N_j will allow *every* term in both nodes N_i *and* N_j to be constructed *off of the other terms in*

these nodes as well as off of the existing terms. Clearly, DiffAG favors redundancy *between* targets. For targets that are far away in terms of adder distance, DiffAG will create new targets so that it can exploit the redundancy between all of these targets (instead of exploiting the redundancy within each of the original targets). In general, difference-based algorithms, such as [24--27], favor the redundancy between targets. We only consider DiffAG since it outperforms all of [24--27].

Hcub is relatively balanced in terms of using redundancy within and between constants. Recall the heuristic part of Hcub selects one successor according to (6.1).

$$\mathrm{H}_{\mathrm{cub}}(R,S,T) = \arg\max_{s \in S} \left(\sum_{t \in T'} \hat{\mathrm{B}}(R,s,t) \right) \tag{6.1}$$

$$\hat{\mathrm{B}}(R,s,t) = 10^{-\mathrm{dist}(R \cup \{s\},t)} \left(\mathrm{dist}(R,t) - \mathrm{dist}(R \cup \{s\},t) \right) \tag{6.2}$$

The adder distance must be computed in order to evaluate the weighted benefit function (6.2). Notice that (6.2) is a function of a *single* target, not T'. Therefore, (6.2) measures the redundancy *within* each constant. By adding the benefit over all $t \in T'$, we can quantify how much *joint benefit* each successor s provides for *all* of the targets. Thus, redundancy *between* constants is strictly handled by (6.1).

DiffAG needs 3 adders for the SCM of 45. Hcub can find the optimal solution, which requires a *multiplicative* decomposition: $45x = ((3x) \ll 4) - (3x)$ where $3x = (x \ll 1) + x$. In DiffAG, one node corresponds to the existing terms in R, thus if a term in the difference set between R and t is also in the successor set S, then t is distance 2. The difference sets enables DiffAG to find $t \in \mathcal{A}(R,S)$, but DiffAG cannot find $t \in C_1 \cdot S$. Unlike the *additive* difference sets, *multiplicative* decompositions are *unidirectional*, which limits the ability to share terms between constants.

However, sometimes it is better to reduce the adder distance *between* the targets first (before reducing the distance from R to the targets). For example, consider a MCM problem with $T = \{41, 103\}$. DiffAG will realize that $103x = ((9x) \ll 4) - 41x$, thus 9 is in the difference set between 41 and 103. Also, $41x = (x \ll 5) + (9x)$ where $9x = (x \ll 3) + x$. DiffAG finds a solution with 3 adders whereas Hcub needs 4. Although Hcub could also find $41x = (x \ll 5) + (9x)$, there are multiple useful supporting terms for $41x$ and Hcub arbitrarily selected the wrong term in this case. Since Hcub does not consider the difference sets used in DiffAG, it has no reason to construct $9x$ in particular. It arbitrarily chooses to construct the supporting term $5x = (x \ll 2) + x$ so that $41x = ((5x) \ll 3) + x$. With $R = \{1, 5, 41\}$, 103 is not constructible with one more adder. In the remaining solution found by Hcub, $9x$ and $103x$ are constructed the same way as illustrated above.

6.1.2 Enhancing the Use of Redundancy Within Constants

6.1.2.1 Justification

By designing a heuristic which is better able to exploit the redundancy within constants, we expect that it will perform better than the existing algorithms on MCM problems with only a few constants but on large bit widths. However, it may also perform worse on problems with many constants of small bit width. As shown in section 6.1.5, better results are obtained when the heuristic is *catered towards the characteristics* of the MCM problem instances.

Our proposed algorithms H3 and H4 place more emphasis on redundancy within constants. However, overly favoring the redundancy within constants can produce poor solutions. For example, given a few large constants, RAG-n will typically operate

mostly by using the optimal SCM lookup table, thereby sharing few or no terms between constants. As shown in [6], Hcub consistently outperforms RAG-n (except for SCM) which suggests that even for MCM problems with a few large constants, the redundancy between constants is not trivial.

In section 6.3.4.4, we will show that the size of the solution space in the MCM problem grows faster with respect to the bit width than with respect to the number of constants. As the compute power continues to increase in the future, exhaustive searches will be able to solve larger problem sizes within a reasonable amount of time. However, the problem sizes will grow by more in the number of constants than in the bit width. By design, our heuristics H3 and H4 achieve more improvement over other heuristics as the bit width increases, thus H3 and H4 will likely be the last of the heuristic algorithms to be replaced by an exhaustive search in the future.

6.1.2.2 Modification of the Components in Hcub

As explained in section 6.1.1.4, redundancy between constants is strictly handled by summing the benefit $\hat{\mathrm{B}}$ over all $t \in T'$. Since our focus is on redundancy within constants, there is little motivation to modify (6.1). We are interested in the weighted benefit function (6.2), which measures the redundancy within each constant. The accuracy of this measurement depends on the accuracy of the measurement of the adder distance. Hcub computes the exact adder distance up to distance 3, but beyond this an estimate is used since the amount of computation increases very quickly with respect to increasing the adder distance. In section 6.1.3, we will propose more computationally efficient exact adder distance tests and we will significantly improve Hcub's distance estimators.

6.1.3 The H3 Algorithm

Our proposed algorithm H3 reuses equations (6.1) and (6.2) from Hcub. Our improvement comes from how the adder distance is computed. As shown in section 6.1.5, H3 is both faster and produces better solutions than Hcub. In each subproblem, we adjust the quality of solutions versus the amount of computation so that overall we gain in both. H3 uses exact distance 2 and 3 tests; we estimate distance 4 and higher.

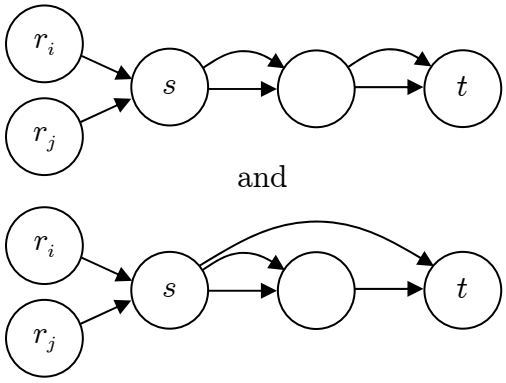
6.1.3.1 Exact Adder Distance 3 Tests

For the exact distance tests, our objective is to reduce the computation. Hcub already caches the sets t/C_1 and t/C_2 for each t, the successor set S is updated incrementally, and S is kept sorted to facilitate a fast intersection with it. As shown in section 3.2.6, for distance 2 we need to test $t/C_1 \cap S \neq \emptyset$ and $\mathcal{A}(t, R) \cap S \neq \emptyset$. These tests are already computationally efficient, so no modification is needed.

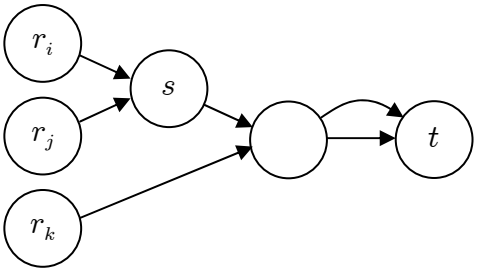
The four distance 3 inverse graph traversal tests are shown in Figure 6.3. Note that the test in Figure 6.3(a) covers two graph topologies. Before we do distance 3 tests, we must use distance 2 tests to confirm that the target is *more than* a distance of 2, so $\mathcal{A}(t, R)$ from the distance 2 tests can be reused in the test in Figure 6.3(c).

Every time we traverse one adder in the graph, we can divide by C_1 or adder-operate with R. We can traverse 2 adders by dividing by C_2 or adder-operating with S ($\mathcal{A}(t, s)$ and constructing a $s \in S$ each need one adder). In the distance 3 tests, we back-project 2 adders and forward-project one adder (by intersecting with S).

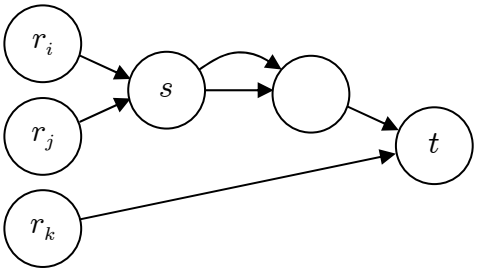
As shown in [6], the $\mathcal{A}(t, S) \cap S \neq \emptyset$ test requires much more computation than all of the other tests. We will provide some insight to complement the mathematical explanation given in [6]. Note that $\mathcal{A}(t, S) \cap S \neq \emptyset$ and $\mathcal{A}(\mathcal{A}(t, R), R) \cap S \neq \emptyset$ are



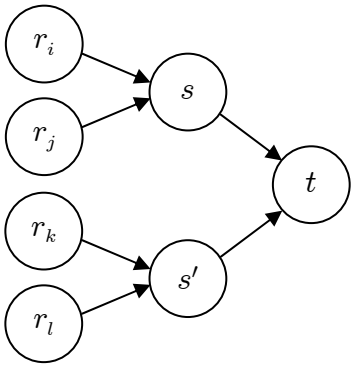
(a) $t \in \mathcal{A}(r_i, r_j) \cdot C_2 = S \cdot C_2$
IGT test: $t/C_2 \cap S \neq \emptyset$



(b) $t \in \mathcal{A}(r_i, r_j, r_k) \cdot C_1 = \mathcal{A}(R, S) \cdot C_1$
IGT test: $\mathcal{A}(t/C_1, R) \cap S \neq \emptyset$



(c) $t \in \mathcal{A}(\mathcal{A}(r_i, r_j) \cdot C_1, r_k) = \mathcal{A}(S \cdot C_1, R)$
IGT test: $\mathcal{A}(t, R)/C_1 \cap S \neq \emptyset$



(d) $t \in \mathcal{A}(r_i, r_j, r_k, r_l) = \mathcal{A}(S, S)$
IGT test: $\mathcal{A}(t, S) \cap S \neq \emptyset$

Figure 6.3: A summary of the distance 3 tests. The topology is first described and then the corresponding inverse graph traversal (IGT) test is provided.

equivalent due to vertex reduction. All of the other tests involve division, so let us compare how much computation is needed to construct t/C_1 versus $\mathcal{A}(t, R)$. C_1 contains a few tens of elements and often only a few will divide t with zero remainder (which means that few or no terms go on the next step of the back-projection). Conversely, for $\mathcal{A}(t, R)$, we must add, subtract, and shift *every possible combination* of t and r *for each* element $r \in R$. R can get quite large, and we will consider problem sizes with a few hundred adders in the solution (R must get this large when the algorithm ends). In general, adder-operating is more expensive than dividing.

Most of the computation in the $\mathcal{A}(t, S) \cap S \neq \emptyset$ test comes from constructing $\mathcal{A}(t, S)$ (as each element in $\mathcal{A}(t, S)$ is constructed, a binary search is used to check for membership in S since S is sorted). Unlike Hcub, we will cache solutions as they are found. Thus, we are only concerned with finding *new* solutions that did not exist on the previous iteration. Any newly created solution of the form $t \in \mathcal{A}(S, S)$ must use a newly created successor as one or both of the supporting terms for t. Newly created successors are created during the update of S between the previous and current iteration, let us denote this with S_{update}. As explained in section 4.3.3, $S_{update} = (C_1 \cdot r') \cup \mathcal{A}(r', R_{old})$ where r' is the newly created element in R (i.e. $R_{new} = R_{old} \cup \{r'\}$). Since $S_{update} \subseteq S$, then $\mathcal{A}(S_{update}, S_{update}) \subseteq \mathcal{A}(S_{update}, S)$, thus we need to test for $\mathcal{A}(t, S_{update}) \cap S \neq \emptyset$ to find *new* solutions. Clearly, this requires much less computation, as S_{update} contains a small fraction of the elements in S.

6.1.3.2 Estimating the Adder Distance

Exact distance tests at distance 4 and higher can be used, however this requires a significant amount of computation, so Hcub uses an estimate for distance 4 and higher.

In the exact adder distance tests, when we confirm that $\text{dist}(R,t) = n$, we also find which successors s are useful (i.e. which s have $\text{dist}(R \cup \{s\}, t) = n - 1$). This is due to the intersection with S. Conversely, the distance estimators in Hcub guess and check how useful each $s \in S$ is in order to compute $\text{dist}(R \cup \{s\}, t)$.

In Hcub, $\text{dist}(R,t)$ takes the value of $\min_{s \in S} \text{dist}(R \cup \{s\}, t)$ from the *previous* iteration (on the first iteration, $\text{dist}(R,t)$ takes an arbitrary value of 1000). However, in our proposed algorithms H3 and H4, we define $\text{dist}(R,t) = \min_{s \in S} \text{dist}(R \cup \{s\}, t) + 1$ (we do *not* use values from previous iterations). This enforces that $\text{dist}(R,t) - \text{dist}(R \cup \{s\}, t) \in \{0, 1\}$. At least one successor must be useful towards the construction of t or else t is impossible to make since S represents the first step along any path of construction. Hcub does not enforce $\text{dist}(R,t) - \text{dist}(R \cup \{s\}, t) \in \{0, 1\}$, so the benefit function (6.2) may assign a successor more benefit than it deserves. We have observed that this distorted benefit function can misguide the heuristic (6.1), thereby producing poor solutions, especially when Hcub is used with large bit widths. We impose $\text{dist}(R,t) - \text{dist}(R \cup \{s\}, t) \in \{0, 1\}$ to reduce this distortion.

Now consider how $\text{dist}(R \cup \{s\}, t)$ is computed. For each successor s and each target t, we would like to figure out what terms would enable the construction of t with one adder (assuming s is also constructed). A term z is useful if $t \in \mathcal{A}(s, z)$. By Theorem 1, this is satisfied iff $z \in \mathcal{A}(t, s)$. It follows that the set of useful terms is $Z = \mathcal{A}(t, s)$. Let $csd(Z)$ denote the minimum CSD cost of any element in Z, then $\text{dist}(R \cup \{s\}, t) \leq csd(Z) + 1$. At least one element $z \in Z$ can be constructed with $csd(Z)$ adders and then one more adder is needed to adder-operate z and s to make t. It is possible that t could be constructed with fewer adders since we have not examined all possible graph topologies, hence the use of the inequality.

Assuming s is constructed, we could also consider terms that would enable t to be constructed with two more adders. By enumerating all of the graph topologies in which two adders can construct t starting from s and z (and R), it follows that z is useful term if $t \in \mathcal{A}(z, s) \cdot C_1$, $t \in \mathcal{A}(z, C_1 \cdot s)$, or $t \in \mathcal{A}(z, s, R)$. Using a similar argument as before, the set of useful terms is $Z = \mathcal{A}(t/C_1, s)$, $Z = \mathcal{A}(t, C_1 \cdot s)$, and $Z = \mathcal{A}(t, s, R)$, respectively. In each of these cases, we have $\text{dist}(R \cup \{s\}, t) \leq csd(Z) + 2$. Each test produces an upper bound on the number of adders needed to construct t. We are interested in finding the minimum number of adders to construct t, thus we can take the smallest upper bound. In conclusion, the distance is estimated in Hcub as shown in (6.3).

$$\text{dist}(R \cup \{s\}, t) \approx \min \begin{cases} csd(\ \mathcal{A}(t, s)\ ) + 1 \\ csd(\ \mathcal{A}(t/C_1, s)\ ) + 2 \\ csd(\ \mathcal{A}(t, C_1 \cdot s)\ ) + 2 \end{cases} \tag{6.3}$$

For each successor-target pair (each s and each t), the distance estimation in (6.3) is independent of R, so it is computed *only once in the entire algorithm*, not every iteration. For this reason, Hcub does not consider the $Z = \mathcal{A}(t, s, R)$ case.

In our interpretation, using a partial graph topology forces t to be constructed with s, thus the distance estimators measure how much redundancy within t is encompassed by s. The remaining cost $csd(Z)$ represents the *remaining* redundancy within t that s could not provide, so a smaller value of $csd(Z)$ means that s is more useful in constructing t.

While this seems like a reasonable approach, there are two fundamental problems. Firstly, the partial graph topologies only consider one or two instances of s. In order to find large amounts of redundancy *within* a constant, we need to consider *many*

instances of s. Secondly, the remaining redundancy within t (leftover from s) is poorly captured by the CSD cost, as the CSD method does not share intermediate terms, it only minimizes the number of terms (recall this is often done *before* CSE even starts). For example, at 32 bits, we have observed that the first substitution made by H(k)+ODP typically happens in 4 places in the CSE form of the constant. When $R = \{1\}$, the average distance estimate that Hcub produces is about 9 adders whereas the optimal SCM cost is less than 6 adders. The *initial estimate* in Hcub is very misguided due to the CSD cost and only considering up to 2 instances of s.

In H3 and H4, we divide the distance estimation problem into two classes. In one case, the target t may not be too far from R but far enough that the exact distance tests cannot construct t. When only a few adders are needed to construct t, we do not need to consider numerous instances of s (as this requires several adders). In the second case, t may be so far away from R that its cost can be approximated with a SCM solution (or a partially traversed SCM solution once the appropriate supporting terms for t have been constructed).

6.1.3.3 Special Distance 4 Estimators

As proved in [6], Hcub may not terminate if the optimal SCM cost of Z is used. The partial graph tests search only part of the solution space, so on the next iteration, they are not guaranteed to find a successor s with a lower dist$(R \cup \{s\}, t)$ than the existing dist(R, t). Thus there would appear to be no way of taking the first step in constructing t. This problem does not occur if the CSD cost is used for Z, as proved in [6]. We evade this problem in H3 and H4 by caching solutions as they are found. We will reuse C_2, which was already computed for the distance 3 test $t/C_2 \cap S \neq \emptyset$.

We can make computationally efficient partial distance 4 tests with only a slight modification to Hcub's estimators. If $\text{dist}(R,t) = 4$, then $\text{dist}(R \cup \{s\}, t) = 3$ for a useful $s \in S$. From (6.3), this happens when $csd(\mathcal{A}(t,s)) = 2$, $csd(\mathcal{A}(t/C_1,s)) = 1$, or $csd(\mathcal{A}(t, C_1 \cdot s)) = 1$. Since we need to construct these 3 sets in order to evaluate the CSD cost, we may as well *also* check if any element in $\mathcal{A}(t,s)$ is also in S_2 and if any element in $\mathcal{A}(t/C_1,s)$ or $\mathcal{A}(t, C_1 \cdot s)$ is also in S. S_2 denotes the 2^{nd} order successor set, $S_2 = (C_1 \cdot S) \cup \mathcal{A}(R,S)$. However, it is computationally expensive to construct S_2 or to perform distance 2 tests for each element in $\mathcal{A}(t,s)$ (since $\mathcal{A}(t,s)$ is typically a large set). Instead, we use C_2 in place of S_2. In addition to what Hcub's partial graph estimators can already detect, by intersecting with S or C_2, we can detect some targets at distance 4 with very little extra computation (both S and C_2 are sorted). Furthermore, once we confirm that a target is distance 4, we no longer need to evaluate the CSD cost (only quick lookups with binary search are done).

Distance 3 tests are done *before* estimation to first confirm that t is *more than* 3 adders away from R. Since $\mathcal{A}(t,s)$ was already computed by the distance 3 test in Figure 6.3(d), we reuse it in the $\mathcal{A}(t,s) \cap C_2 \neq \emptyset$ test. This idea of reusing sets already computed for the exact distance tests will be revisited in section 6.1.4.

6.1.3.4 The H(k)+ODP Estimator

We have efficient distance 4 estimators, but the average SCM cost is about 6 adders at 32 bits, for example. We still need a good *large distance* estimator. One key observation is that typically once the first three terms of a 32 bit target t are constructed, the exact distance tests will find all of the possible ways to finish constructing t from here (as t would then be 3 adders away from construction). Since adder distance tests close

to t are exact, the *challenge* is to find useful intermediate terms that are *far away* from t. Another key observation is that the first term for constructing t must be in C_1 and thus does need to be built off of other terms. Generally, terms far away from t are of low adder cost (elements of C_n with small n) and thus their construction is unlikely to be assisted by other high cost intermediate terms. Based on our earlier analysis of redundancy, a CSE based algorithm such as H(k)+ODP is well suited to find these low cost useful intermediate terms which are far away from t.

Recall that H(k)+ODP searches and substitutes patterns in each SD form of the constant individually and then selects the best answer at the end. Modifying H(k)+ODP to return *all* of the equally good SCM solutions requires very little extra computation (we just need to count and store the solutions). Although H(k)+ODP is a *heuristic* whereas the BIGE algorithm is *optimal*, modifying the BIGE algorithm to return *all* of the possible solutions would require us to exhaustively search at n adders. Recall from section 5.2.4 that the bounding heuristic is often optimal, so the exhaustive search only needs to be done up to $n-1$ adders. Exhaustively searching one more adder results in an increase in run time by a couple orders of magnitude.

In the H3 algorithm, we will use H(1)+ODP. Although H(2)+ODP would likely provide better and/or more SCM solutions with the minimum cost, this typically requires one order of magnitude more run time. In both the H3 and H4 algorithms, the best SCM solutions are found and stored *only on the first iteration*. For each target, the SCM solutions are updated as they are traversed. If a term gets constructed and is also in a SCM solution of t, the remaining SCM cost of t is decremented by 1 and the remaining SCM solutions of t are updated (we only store the parts of the SCM solutions that have not been traversed yet). Since we track the remaining SCM cost

for each t, if any $s \in S$ is also in a SCM solution of t, then $\text{dist}(R \cup \{s\}, t)$ is the remaining SCM cost minus 1. Otherwise, given that s is not in a SCM solution of t, then $\text{dist}(R \cup \{s\}, t)$ is simply the remaining SCM cost (as this s does not allow us to traverse any remaining SCM solution of t).

In the H3 and H4 algorithms, we use both the modified partial graph topology estimators and the H(k)+ODP estimator. For each s, $\text{dist}(R \cup \{s\}, t)$ is taken as the minimum from either estimator. H(k)+ODP searches for good solutions and then tries to match these to the successor set. The partial graph tests do the opposite, they estimate how much redundancy a successor has within a target. While the latter approach may seem more viable, it is the *accuracy of the estimator* that is paramount.

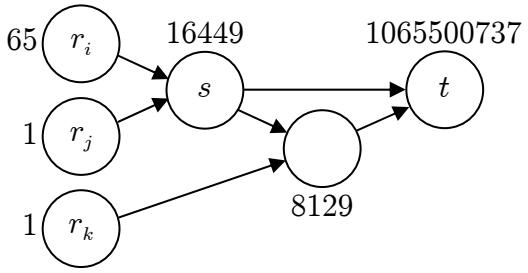
6.1.3.5 The Vertex Reordering Problem and the Caching of Solutions

In the H3 algorithm, we enforce $\mathcal{A}(x, y) \leq 2^b$, where b is the bit width of the largest target. Conversely, Hcub enforces $\mathcal{A}(x, y) \leq 2^{b+1}$. In both H3 and Hcub, the exact adder distance tests are not *truly* exact (as explained in detail in section 5.2.3), however our results suggest that they remain sufficiently accurate.

By imposing tighter restrictions on the adder-operation, we obtain smaller sets at each node when doing the graph tests, thus the run time is reduced. One important observation is that $\mathcal{A}(x, y)$ has a better coverage of small numbers than large numbers. As the shifts in the adder-operation increase, the numbers that $\mathcal{A}(x, y)$ produce become sparser. For example, $C_1 = \{3, 5, 7, 9, 15, 17, 31, 33, 63, 65, 127, \ldots\}$ exhibits this behavior. In the graph tests, we compute a back-projected set $f(\cdot)$ using adder-operations and then intersect it with S. By construction, $f(\cdot)$, R, S, and C_n have a higher density of small numbers, and thus so does $f(\cdot) \cap S$, for example. Even

$$65x = (x \ll 6) + x$$
$$16449x = (x \ll 14) + (65x)$$
$$8129x = (16449x) - ((65x) \ll 7$$
$$1065500737x = ((8129x) \ll 17) + (16449x)$$

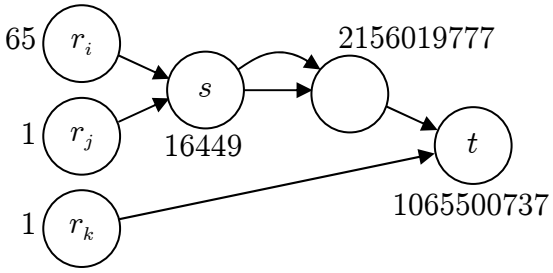
(a) SCM solution found by H(k)+ODP.



(b) After 65 is constructed ($R = \{1, 65\}$), the remaining topology is $t \in \mathcal{A}(\mathcal{A}(s, R), s)$.

$$65x = (x \ll 6) + x$$
$$16449x = (x \ll 14) + (65x)$$
$$2156019777x = ((16449x) \ll 17) + (16449x)$$
$$1065500737x = (2156019777x) - ((65x) \ll 24)$$

(c) Remapped version of the SCM solution found by H(k)+ODP.



(d) After 65 is constructed, the remapped remaining topology is $t \in \mathcal{A}(\mathcal{A}(s, s), R)$.

Figure 6.4: An illustration of the vertex reordering problem caused by the $\mathcal{A}(x, y) \leq 2^b$ constraint. Nodes are labeled with values, edges are not labeled. By using vertex reduction, $1065500737x = (16449x) + ((16449x) \ll 17) - ((65x) \ll 24)$.

for large targets, the back-projection $f(\cdot)$ inherently favors smaller numbers. We have observed that this is reflected in the solutions of most MCM problems, as the intermediate terms are typically smaller than the targets.

We discard SCM solutions that have terms larger than 2^b. Since exhaustive distance 3 tests subject to $\mathcal{A}(x, y) \leq 2^b$ are used, it is expected that we can transition from traversing a SCM solution at distance 4 to using an exact test at distance 3. However, a problem arises due to vertex reduction. Consider a MCM problem with $T = \{65, 1065500737\}$. The only cost 4 solution that H(k)+ODP finds which uses 65 is shown in Figure 6.4(a). Once 65 is constructed, the remaining topology is $t \in \mathcal{A}(s, s, R)$, where $s = 16449$. In order to satisfy the $\mathcal{A}(x, y) \leq 2^b$ constraint, we must use a topology of $t \in \mathcal{A}(\mathcal{A}(s, R), s)$, as shown in Figure 6.4(b). However, the distance 3 test uses a topology of $t \in \mathcal{A}(\mathcal{A}(s, s), R)$ in order to minimize the computation required. The solution can be remapped to this topology, as shown in Figures 6.4(c) and 6.4(d), however the remapped solution does not satisfy $\mathcal{A}(x, y) \leq 2^b$.

Because of vertex reduction, terms can be combined *in the wrong order*, thereby preventing the algorithm from following a solution that satisfies the $\mathcal{A}(x, y) \leq 2^b$ constraint. To avoid this problem, any cached solution can be traversed until the distance 2 tests can be used (the distance 1 and 2 tests cover all of the *non-vertex reduced* graph topologies). Cached solutions include: SCM solutions, solutions found by partial graphs, and solutions found by the exact distance 4 tests in the H4 algorithm.

6.1.3.6 Post-removal of Unnecessary Intermediate Terms

The idea of removing unnecessary intermediate terms *after* the solution is found was first proposed in [36]. As illustrated in section 4.6, for each intermediate term $r \in R$

where $r \notin T$, we check whether $R \backslash \{r\}$ is still a valid solution (i.e. it still satisfies the $r_k \in \mathcal{A}(r_i, r_j)$ constraint for each $k \geq 1$ where $0 \leq i, j < k$ from Definition 3). This may involve reordering the elements in R. Essentially, we solve a MCM problem for $T = R \backslash \{r\}$. If this can be solved using *only* the optimal part of RAG-n, then $R \backslash \{r\}$ is valid solution and we permanently remove r from R, otherwise we cannot remove r from R. This process is done for each $r \in R$.

Although we only consider the optimal part of RAG-n, we never construct the successor set. Instead, we check if $\mathcal{A}(t, R) \cap R \neq \emptyset$, as the symmetry in this can be exploited. This tests the equivalent of $t \in \mathcal{A}(R, R) = \{z \mid z = |2^m r_i \pm 2^n r_j|, r_i \in R, r_j \in R\}$, but since r_i and r_j go through all possible values of R and we use the shifts $m > 0$ and $n = 0$, we do not need to consider $m = 0$ and $n > 0$. The R is never shifted when computing $\mathcal{A}(t, R)$. We absorb this shift with the R we intersect with. When $\mathcal{A}(t, R)$ produces an even integer, this is divided by 2 until odd *before* we intersect with R (this division is equivalent to left shifting the R we intersect with).

Only one pass is done through R. While this may lead to local minima, the run time to use an exhaustive branching approach is not practical.

6.1.3.7 A Summary of the H3 Algorithm

H3 uses more efficient exhaustive distance 3 tests than Hcub. With the modified partial graph estimators, we can detect some targets at distance 4 that Hcub cannot by using only a little extra computation. H(1)+ODP is used to provide much better large distance estimates. We use a tighter bound on the adder-operation to improve the run time. Finally, the post-removal of unnecessary intermediate terms further increases the amount of redundancy exploited both within and between constants.

6.1.4 The H4 Algorithm

Our proposed algorithm H4 is essentially a stronger version of H3 and it reuses (6.1) and (6.2) from Hcub. With H3, we showed that better solutions can be obtained in less run time than Hcub. However, for problems with only a few large constants, the *absolute* run time is small. With H4, we will show that *significantly* better solutions can be obtained, but at the expense of run time. Even so, the absolute run time is still tolerable in many cases (results are presented in section 6.1.5).

We enforce $\mathcal{A}(x, y) \leq 2^{b+2}$ in the H4 algorithm. This requires more computation compared to Hcub and H3, but more solutions will be found by any graph-based test, so there will be a higher chance of sharing useful successors between targets. H4 performs a post-removal of unnecessary intermediate terms (using the same method as H3). The vertex reordering problem can still occur, so all cached solutions must be stored up to distance 2 (so that they can be traversed this far). In addition to the SCM and partial graph estimators, we must cache solutions found at distance 4 otherwise we may not find a solution with the distance 3 tests subject to $\mathcal{A}(x, y) \leq 2^{b+2}$.

The H4 algorithm uses exhaustive graph tests up to distance 4. Distance 2 and distance 3 tests were discussed in sections 3.2.6 and 6.1.3.1, respectively. Several examples were illustrated in section 5.2.2 in which we demonstrated how to apply the inverse graph traversal method for designing an adder distance test for a given graph topology. By applying the same technique to every possible graph topology with 4 adders, we obtain all of the distance 4 tests, which are summarized in Table 6.1. The distance 2 and 3 tests must be done *before* the distance 4 tests in order to confirm that a target has a distance of *more than* 3. Therefore, we can reuse $\mathcal{A}(t, R)$, $\mathcal{A}(t/C_1, R)$, $\mathcal{A}(t, R)/C_1$, and $\mathcal{A}(t, S_{update})$, as these sets were created and cached in

Table 6.1: A summary of the distance 4 tests.

Case	Test	Graphs covered in [2]
1	$\frac{t}{C_3} \cap S \neq \emptyset$	5*, 6, 7*, 8, 9*
2	$\mathcal{A}\left(\frac{t}{C_2}, R\right) \cap S \neq \emptyset$	9, 10
3	$\frac{\mathcal{A}\left(\frac{t}{C_1}, R\right)}{C_1} \cap S \neq \emptyset$	7
4	$\frac{\mathcal{A}(t, R)}{C_2} \cap S \neq \emptyset$	3*, 4
5	$\mathcal{A}\left(\frac{\mathcal{A}(t, R)}{C_1}, R\right) \cap S \neq \emptyset$	3
6	$\mathcal{A}\left(\frac{t}{C_1}, S\right) \cap S \neq \emptyset$	5
7	$\frac{\mathcal{A}(t, S)}{C_1} \cap S \neq \emptyset$	2
8	for each $s \in S : \mathcal{A}\left(\frac{\mathcal{A}(t, s)}{C_1}, s\right) \cap R \neq \emptyset$	11
9	$\mathcal{A}(t, R, S) \cap S \neq \emptyset$	1

the distance 2 and 3 tests. Unlike the distance 5 tests in section 5.2.2, we cannot stop after the first solution is found by a distance test. In order to compute $\text{dist}(R \cup \{s\}, t)$ for *every* successor s, all of the tests must be done to completion.

Distance estimation is needed for distance 5 and higher in the H4 algorithm. Many of the ideas presented in H3 are reused. For example, the large distance estimator is based on traversing SCM solutions found by H(2)+ODP. Compared to H(1)+ODP (used in the H3 algorithm), typically H(2)+ODP produces solutions with fewer adders and/or more equally good solutions (simply because it searches more SD forms). With more solutions, there is a better chance that a term in a SCM solution of a target

Table 6.2: A summary of the distance 5 and 6 partial graph estimators.

Case	Corresponding Distance 4 Case	Distance 5 Estimator	Distance 6 Estimator
1	6	$\mathcal{A}\left(\frac{t}{C_1}, S\right) \cap C_2 \neq \emptyset$	$\mathcal{A}\left(\frac{t}{C_1}, S\right) \cap C_3 \neq \emptyset$
2	7	$\frac{\mathcal{A}(t,S)}{C_1} \cap C_2 \neq \emptyset$	$\frac{\mathcal{A}(t,S)}{C_1} \cap C_3 \neq \emptyset$
3	8	for each $s \in S$: $\mathcal{A}\left(\frac{\mathcal{A}(t,s)}{C_1}, s\right) \cap S \neq \emptyset$	for each $s \in S$: $\mathcal{A}\left(\frac{\mathcal{A}(t,s)}{C_1}, s\right) \cap C_2 \neq \emptyset$
4	9	$\mathcal{A}(t, R, S) \cap C_2 \neq \emptyset$	$\mathcal{A}(t, R, S) \cap C_3 \neq \emptyset$

may also be in the SCM solution of another target, thereby facilitating more sharing of terms between targets. H3 does not use H(2)+ODP since H(2)+ODP is slower than Hcub. Given the simplicity of traversing a SCM solution, since the run time of H(2)+ODP is much faster than H4, the H(k)+ODP estimator contributes little to the total run time of H4 (thus the graph tests require most of the computation).

For targets that are not too far in terms of adder distance but still out of the reach of the exact tests, the partial graph estimator is used. We can reuse sets that are computed by the exhaustive distance 4 tests, thereby performing estimation with little extra computation. From Table 6.1, all of the distance 4 tests except case 8 have the form

$$f(\cdot) \cap S \neq \emptyset$$

where $f(\cdot)$ is the back-projected set. It follows that the corresponding distance 5 estimation test is

$$f(\cdot) \cap C_2 \neq \emptyset$$

In general, if $f(\cdot) \cap S \neq \emptyset$ is an exact distance k test, then $f(\cdot) \cap C_2 \neq \emptyset$ is a distance

$k+1$ estimator. It also follows that $f(\cdot) \cap C_n \neq \emptyset$ is a distance $k+n-1$ estimator. Since the estimator does not intersect with S, in order to know *which successors* were useful towards constructing t (to estimate dist$(R \cup \{s\}, t)$), the back-projected set $f(\cdot)$ *must be a function of* S. From Table 6.1, cases 6-9 have a $f(\cdot)$ which uses S. Cases 6, 7 and 9 have the form $f(\cdot) \cap S \neq \emptyset$, so the corresponding distance 5 and 6 estimators are $f(\cdot) \cap C_2 \neq \emptyset$ and $f(\cdot) \cap C_3 \neq \emptyset$, respectively. By a similar argument, since case 8 from Table 6.1 has the form $f(\cdot) \cap R \neq \emptyset$, the corresponding distance 5 and 6 estimators are $f(\cdot) \cap S \neq \emptyset$ and $f(\cdot) \cap C_2 \neq \emptyset$, respectively. A summary of the distance 5 and 6 estimators is provided in Table 6.2.

The successor set S is typically a large set, so it is not practical to store all of the $f(\cdot)$ sets needed for distance estimation *for every* $s \in S$. Thus, distance estimation is done at the same time as the distance 4 tests. The $f(\cdot)$ sets are computed depth first to minimize the amount of memory needed. Every time an element in $f(\cdot)$ is generated, we check whether it is also in S, C_2, and C_3 for cases 6, 7, and 9, whereas for case 8, we check if it is in R, S, and C_2. If a target has already been confirmed as distance 5, we do not test for membership in the last set (C_3 in cases 6, 7, and 9, C_2 in case 8). Likewise, if a target has already been confirmed as distance 4, no distance estimation is done. Both the SCM and partial graph estimators are used up to distance 6. Unlike H3, the partial graph estimators do not consider the CSD cost. When t is more than 6 adders from R, only the SCM estimator is used and we approximate dist(R, t) with the SCM cost of t (or the remaining SCM cost if we have partially traversed a SCM solution). In general, as the adder distance from t to R increases, the solutions with the least number of adders to construct t become less dependent on the existing terms in R.

Table 6.3: The average number of adders in MCM problems with only a few large constants. Note the percent improvement over Hcub is shown in italics.

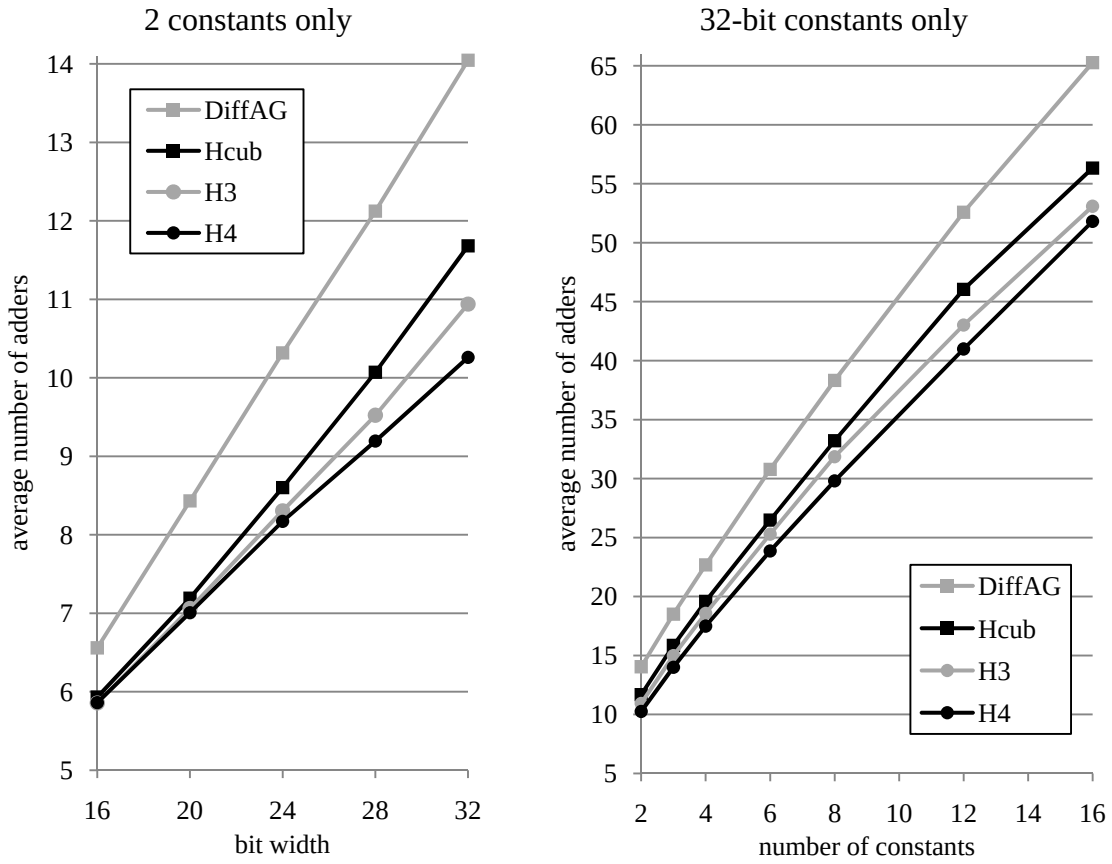
		2 constants		4 constants		8 constants		16 constants	
20 bits	Hcub	7.191		11.862		19.680		33.593	
	H3	7.061	*1.8%*	11.674	*1.6%*	19.474	*1.0%*	33.310	*0.8%*
	H4	7.006	*2.6%*	11.666	*1.7%*	19.479	*1.0%*	33.313	*0.8%*
24 bits	Hcub	8.601		14.114		23.764		39.886	
	H3	8.305	*3.4%*	13.816	*2.1%*	23.304	*1.9%*	39.432	*1.1%*
	H4	8.171	*5.0%*	13.575	*3.8%*	23.157	*2.6%*	39.322	*1.4%*
28 bits	Hcub	10.070		16.710		28.010		47.870	
	H3	9.523	*5.4%*	16.121	*3.5%*	27.036	*3.5%*	46.570	*2.7%*
	H4	9.194	*8.7%*	15.464	*7.5%*	26.333	*6.0%*	46.290	*3.3%*
32 bits	Hcub	11.681		19.593		33.211		56.330	
	H3	10.939	*6.4%*	18.576	*5.2%*	31.851	*4.1%*	53.090	*5.8%*
	H4	10.260	*12.2%*	17.493	*10.7%*	29.804	*10.3%*	51.810	*8.0%*

6.1.5 Experimental Results

The same uniformly distributed random constants were used by all algorithms at each bit width and number of constants. All experiments were benchmarked using identical 3.06 GHz Pentium 4 Xeon workstations running Linux. We implemented H3, H4 and DiffAG in C. We used the authors' C++ implementation of Hcub from [46]. Our implementation of DiffAG is more efficient than the author's version since we *incrementally* update the difference sets $D_{i,j}$ and the successor set (by considering only *new* pairings, as shown in section 4.3.3). In [28], incremental updating was not discussed and run times were not reported. All shared components, such as adder-operations, are computed in the same manner as H3 and H4.

6.1.5.1 A Few Large Constants

We expect the improvement of H3 and H4 (over Hcub) to increase as the amount of redundancy *within* each constant increases, which happens as the bit width increases



(a) A fixed number of constants

(b) A fixed bit width

Figure 6.5: The average number of adders in MCM with a few large constants.

and as the number of constants decreases. We used 1000 random instances at each bit width and each number of constants, except for 28-32 bits and 12-16 constants where 100 MCM instances were used.

In Figure 6.5(a), we show the average number of adders for MCM problems with 2 constants (with varying bit widths). In Figure 6.5(b), the average number of adders is shown for MCM problems with 32 bit constants. DiffAG is included in the comparison to illustrate the penalty incurred by using a heuristic which favors the wrong type of redundancy compared to the given MCM problem instances. More results on the

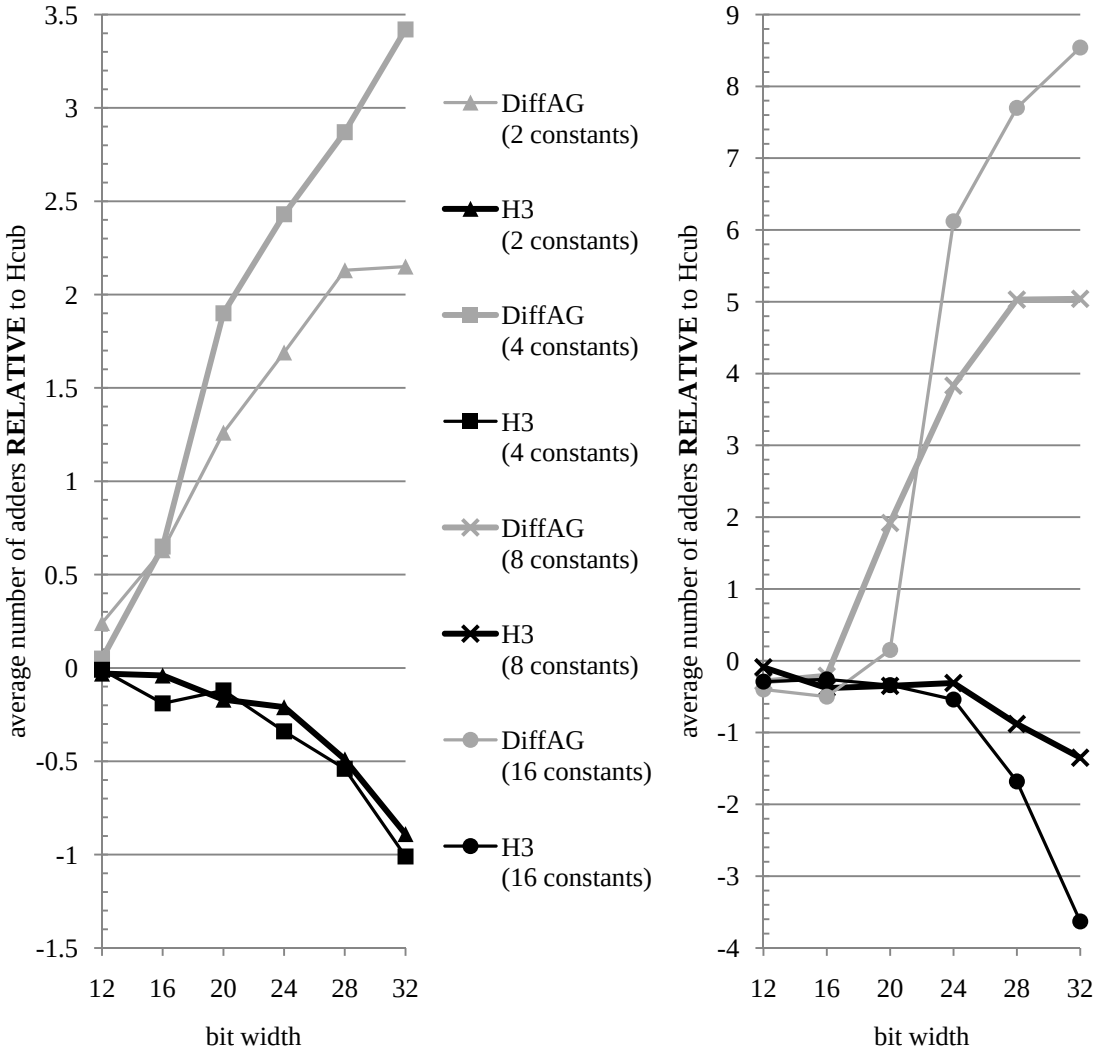
Table 6.4: The average run time (seconds) for MCM with a few large constants.

		2 constants	4 constants	8 constants	16 constants
24 bits	Hcub	0.14	0.44	1.49	6.23
	H3	0.08	0.23	0.57	1.49
	H4	1.47	6.19	18.63	52.61
32 bits	Hcub	0.70	2.64	11.14	55.25
	H3	0.57	2.04	6.78	19.79
	H4	20.01	170.30	1235.35	7537.37

average number of adders are presented in Table 6.3. Run times are shown in Table 6.4. The results confirm that more improvement is obtained where it was expected. At 32 bits and 2 constants, H3 and H4 achieve an average improvement of 6.4% and 12.2% over Hcub, respectively. On average, H3 is faster and produces better solutions than Hcub. H4 produces better solutions at the expense of more run time. Even so, the *absolute run time* is tolerable in many cases. For example, given a MCM problem with 4 constants at 32 bits, one may be willing to wait 3 minutes for a 10% reduction in the number of adders (compared to Hcub). Unless the constants in the hardware design are changed, the H4 algorithm only needs to be run once.

6.1.5.2 A General MCM Benchmark

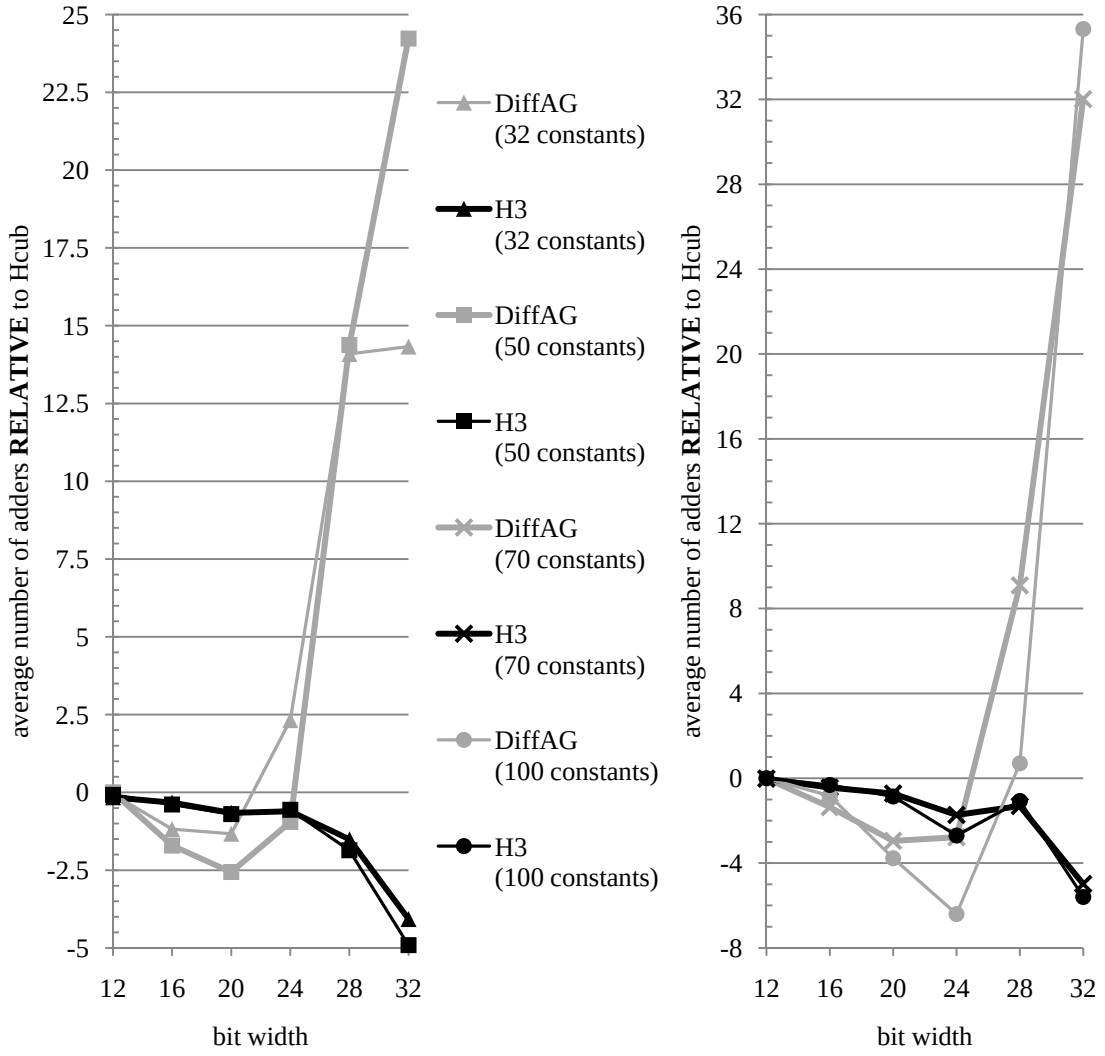
We will compare the performance of H3, Hcub, and DiffAG over the entire spectrum of MCM problem sizes. We will consider 2-100 constants and 12-32 bits. The comparison for SCM was already considered in section 5.1.6 (although results for DiffAG were not shown, they are on average worse than Hcub for SCM). Bit widths less than 12 are not considered since optimal MCM algorithms can solve problems with constants up to 12 bits within reasonable amounts of time, as shown in section 6.3.4.2. At each bit width and each number of constants, 100 random MCM instances were tested.



The average number of adders for Hcub.

	12 bits	16 bits	20 bits	24 bits	28 bits	32 bits
2 constants	4.57	5.83	7.29	8.45	10.04	11.67
4 constants	7.38	9.76	11.87	14.12	16.76	19.54
8 constants	12.04	16.38	19.84	23.65	28.12	33.34
16 constants	19.63	26.90	33.56	39.89	48.39	55.66

Figure 6.6: The average number of adders for H3, DiffAG, and Hcub in the general MCM benchmark, part 1 of 2.



The average number of adders for Hcub.

	12 bits	16 bits	20 bits	24 bits	28 bits	32 bits
32 constants	33.33	45.05	60.01	68.63	81.41	98.38
50 constants	49.68	63.13	82.33	101.85	116.51	137.09
70 constants	68.41	81.56	106.58	138.36	151.70	179.39
100 constants	96.55	108.11	140.25	184.58	206.22	239.37

Figure 6.7: The average number of adders for H3, DiffAG, and Hcub in the general MCM benchmark, part 2 of 2.

Due to the massive amount of experimental results, we present them in a non-standard way. In Figures 6.6 and 6.7, the average number of adders for Hcub is presented in a table format. We are interested in a comparison between H3, Hcub, and DiffAG, so the average number of adders for H3 and DiffAG are given *relative* to Hcub in the diagrams. For example, from the table in Figure 6.6, Hcub produces solutions with an average of 19.54 adders for MCM problems with 4 constants at 32 bits. As shown in the top-left diagram in Figure 6.6, when the H3 algorithm is used with 4 constants at 32 bits, H3 produces solutions with an average of -1 adders *with respect to* Hcub. This means that H3's *absolute* average is 18.54 adders. If the values are *positive* with respect to Hcub, then *more adders* than Hcub are needed. Given the *absolute* values for Hcub and the *relative* values for H3 and DiffAG, one can compute the *absolute* values for H3 and DiffAG if desired.

From Figure 6.7, Hcub requires an average 96.55 adders for 100 constants on 12 bits. The optimal part of RAG-n indicates that we cannot use less than one adder per *unique* target, so this implies that no more than an average of 96.55 unique odd integers were used. We sampled uniformly distributed random constants, so after the constants are preprocessed (made into odd integers by division by 2 until odd), some constants could be the same or could be cost zero (some integer power of 2).

The average run times are provided in Table 6.5. Both H3 and DiffAG are faster than Hcub. H3 is faster is primarily due to the more efficient distance 3 test. From section 6.1.3.1, S_{update} only consists of *new* pairings of R, so it is much smaller than S. Thus for the distance 3 test $\mathcal{A}(t, S) \cap S \neq \emptyset$, constructing $\mathcal{A}(t, S_{update})$ instead of $\mathcal{A}(t, S)$ requires significantly less computation. DiffAG is faster than H3 due to our efficient implementation, which facilitates a fair comparison between the algorithms.

Table 6.5: The average run time (seconds) for DiffAG, H3, and Hcub in the general MCM benchmark.

		16 bits	24 bits	32 bits
4 constants	DiffAG	0.001	0.003	0.015
	H3	0.015	0.111	2.017
	Hcub	0.025	0.233	2.604
16 constants	DiffAG	0.006	0.042	0.323
	H3	0.073	0.749	19.414
	Hcub	0.149	2.988	55.375
50 constants	DiffAG	0.054	0.492	4.863
	H3	0.293	3.888	87.663
	Hcub	0.805	18.147	1075.174
100 constants	DiffAG	0.178	2.009	22.121
	H3	0.642	13.903	348.323
	Hcub	1.428	72.309	9384.150

From Figures 6.6 and 6.7, for each bit width and for each number of constants, H3 on average produces solutions with less or an equal number of adders compared to Hcub. Thus, H3 can effectively replace Hcub, as H3 is also faster than Hcub. However, DiffAG is *sometimes* able to outperform H3. Since DiffAG favors redundancy *between* constants, DiffAG outperforms H3 when there are numerous constants on a small bit width. As shown in Figures 6.6 and 6.7, for a fixed number of constants, as the bit width increases, DiffAG typically performs progressively worse relative to Hcub (because more redundancy *within* each constant can be exploited as the bit width increases). Note that the maximum improvement over Hcub does not happen at the smallest bit width. The results of [15] indicate that Hcub is close to optimal on small bit widths, so there is a *strict limitation* on the amount of achievable improvement. As the bit width increases and Hcub becomes farther from optimal, DiffAG is able to produce better solutions. As the bit width further increases (for a fixed number of constants), eventually H3 performs better. Note that in some cases (such as MCM

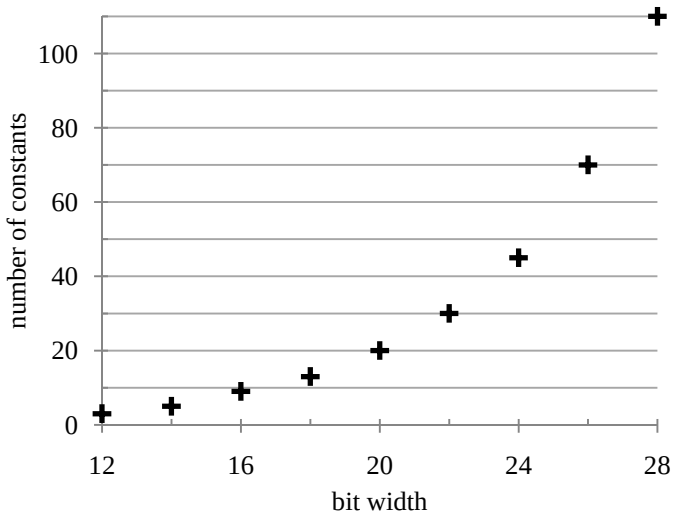


Figure 6.8: The approximate boundary where H3 and DiffAG produce solutions with a similar number of adders on average.

problems with 2 constants in Figure 6.6), we have not examined small enough bit widths to obtain any benefit from DiffAG. However, these problem sizes are small enough to be solved optimally in practice, as shown in section 6.3.4.

There are two regions within the entire spectrum of MCM problem sizes in which H3 and DiffAG produce solutions with a similar number of adders. Since both H3 and DiffAG use the optimal part of RAG-n, one of the regions is obviously the area in which few or no intermediate terms are needed. If there is enough redundancy *between* constants, we can build a target off of other targets without the need for intermediate terms, thus these MCM problems arise when there are numerous constants on small bit widths. The other region of similar performance between H3 and DiffAG is when the average number of adders *per target* is roughly 2. We discovered this via experimental evaluation by iteratively zooming in on the region of interest. The *approximate* boundary is shown in Figure 6.8. We will provide an explanation for this observation in section 6.1.6.

6.1.6 Differential Adder Distance

In this section, we propose the idea of *differential* adder distance. Two targets t_i and t_j are a differential adder distance of n from each other iff $t_j \in \mathcal{A}(t_i, r_1, r_2, \ldots, r_n)$, where each $r_i \in R$. We do *not* define differential adder distance as $\text{dist}(R \cup \{t_i\}, t_j) = n$ because this does *not* imply $\text{dist}(R \cup \{t_j\}, t_i) = n$. The anti-symmetry arises due to *multiplicative* decompositions. For example, if $t_j \in C_1 \cdot t_i$, then $\text{dist}(R \cup \{t_i\}, t_j) = 1$, but is possible that $\text{dist}(R \cup \{t_j\}, t_i) \neq 1$. Conversely, *additive* decompositions are symmetric, hence the use of vertex reduction in the differential adder distance.

There was no notion of differential adder distance in [28] (where DiffAG was proposed). We will introduce it in order to explain the location of the boundary where H3 and DiffAG produce solutions with a similar number of adders. In each node in DiffAG, every element is a differential adder distance of 1 from at least one of the other elements in the same node (in node N_i, for each $t \in N_i$, we must have $\mathcal{A}(t, t') \cap R \neq \emptyset$ where $t' \in N_i$). This results in the property that once any element in a node is constructed, all of the other elements in the same node can then be constructed with only one adder each. Therefore the cost to construct *all* of the terms in node N_i is $\min_{t \in N_i} cost(t) + |N_i| - 1$. We are only concerned with finding the *cheapest* element in N_i, and recall that $|N_i|$ denotes the cardinality of N_i. These nodes can be regarded as how DiffAG *pre-computes* when the optimal part of RAG-n may be used later. However, DiffAG will not detect $t_i \in C_1 \cdot t_j$.

Now consider the *heuristic part* of DiffAG. Let $\mathcal{D}$ denote the union of all the difference sets $D_{i,j} = \mathcal{A}(N_i, N_j)$. One element in $\mathcal{D}$ will be selected on the current iteration. If $\mathcal{D} \cap S \neq \emptyset$, we will construct a useful successor, otherwise a new target $t \in \mathcal{D}$ is added to the set of remaining targets T'. If $\mathcal{D} \cap S \neq \emptyset$, then there exists a

pair of nodes N_i and N_j that are a differential distance of 2. More precisely, there exists a $t_i \in N_i$, $t_j \in N_j$ and $s \in S$ such that $t_i \in \mathcal{A}(t_j, s)$, or equivalently $t_j \in \mathcal{A}(t_i, s)$. Once s is constructed, on the next iteration, nodes N_i and N_j will be merged. Recall that $|N_i| - 1$ of the elements in node N_i will be constructed optimally. Instead of constructing only $|N_i| - 1 + |N_j| - 1$ terms optimally, by merging nodes N_i and N_j, $|N_i| + |N_j| - 1$ will be constructed optimally. It follows that if p pairs of nodes are merged, DiffAG has pre-computed that p *more* terms can be later constructed by the optimal part of RAG-n, hence we pick the s that occurs in the most $D_{i,j}$ sets.

In H3, if a target t is an adder distance of 2 on the current iteration, there exists a $s \in S$ such that $\text{dist}(R \cup \{s\}, t) = 1$. Once this s is constructed, on the next iteration, t will be distance 1, so the optimal part of RAG-n is used to construct t. Likewise in DiffAG, given that $s \in S$ and that $s \in D_{i,j}$, constructing s will later lead to using the optimal part of RAG-n.

As mentioned in section 4.2, if an algorithm can *detect* distance 2 targets as well as find the *useful* intermediate terms, it makes little difference whether a top-down or a bottom-up approach is used. If a target t is distance 2, there must be at least one intermediate term that is distance 1 from R and is also distance 1 from t. In this case, we would like to pick an intermediate term which can *jointly* benefit as many of the targets as possible.

H3 maximizes the joint benefit based on adder distance whereas DiffAG maximizes the joint benefit based on *differential* adder distance. Since R is one of the nodes in DiffAG, intersecting $D_{i,j}$ with S can partially test for distance 2 (DiffAG can find $t \in \mathcal{A}(R, S)$ but not $t \in C_1 \cdot S$). On the current iteration, for some $s \in S$, assume there are k targets in which $\text{dist}(R \cup \{s\}, t) = 1$ and assume that s occurs in m of the

$D_{i,j}$ sets. If we construct s, H3 will construct k targets optimally immediately (over the next k iterations) whereas DiffAG will later be able to construct m *more* targets optimally. If m is typically larger than k, DiffAG will produce better solutions.

DiffAG on average outperforms H3 when the average number of adders *per target* is less than 2 in the final solution. Thus, in these MCM problems, we can deduce that m is typically larger than k. Given less than 2 adders per target, there will be more targets than intermediate terms, so some targets *must* be built off of other targets and *without intermediate terms*. In other words, there is more redundancy *between* constants than *within* each constant, so it is better to join targets first (by merging nodes in DiffAG) rather than first building up to each target. In H3, we are not able to detect if a target can be build off of another target.

Naturally, one could consider a differential distance of 3 in DiffAG. For example, if $\mathcal{D} \cap S = \emptyset$, we then check if $\mathcal{D} \cap \mathcal{A}(R, S) \neq \emptyset$. If this is satisfied, a new target t is added to T', where t is the element in $\mathcal{A}(R, S)$ that occurs in the most $D_{i,j}$ sets. Otherwise (if $\mathcal{D} \cap \mathcal{A}(R, S) = \emptyset$), we use the CSD cost like in the original DiffAG algorithm. We have experimented with this, however the results are negligibly better than DiffAG. The results are not shown in the thesis since they are typically less than 0.5% better. More importantly, the boundary between H3 and DiffAG with differential distance 3 is still around 2 adders per target. The fundamental reason is because constructing a differential distance 2 term provides *optimality* later whereas this is not the case for differential distance 3. For example, even if $t_i \in \mathcal{A}(t_j, t_k, t_l)$, one intermediate term is still required, unlike $t_i \in \mathcal{A}(t_j, t_k)$. In a solution with 3 adders per target, there are twice as many intermediate terms than targets, which suggests that redundancy *within* each constant is more important.

6.1.7 The Hybrid H3+DiffAG Algorithm

The BBB algorithm (better of Bernstein or BHM) was described in section 4.4.1. Given a SCM problem, the Bernstein and BHM algorithms independently solve the same SCM problem and then BBB simply selects the better result. Better solutions are obtained because some SCM instances require a mostly multiplicative decomposition whereas others require a mostly additive decomposition. For the MCM problem, our results in section 6.1.5.2 clearly indicate the importance of using a heuristic that favors the most abundant type of redundancy in the given MCM problem. H3 and DiffAG each produce the best solutions *only* within their respective domains.

We define the H3+DiffAG algorithm in a similar manner to BBB. H3 and DiffAG independently solve the same given MCM problem and the better solution is selected. Over the *entire* spectrum of MCM problem sizes, H3+DiffAG outperforms *every* existing MCM algorithm. The only algorithms that have outperformed Hcub can only do so for problem sizes that are small enough to solve optimally in practice, but the H4 algorithm and optimal MCM algorithms are too slow to solve large problem sizes in practice. In the DiffAG component of H3+DiffAG, we now also perform a post-removal of unnecessary intermediate terms. The average results of H3+DiffAG are *nearly identical* to taking to best of the average results from H3 and DiffAG (a very small improvement occurs near the boundary since both H3 and DiffAG contribute to the final solution over several MCM instances). DiffAG is used first because it is faster. If DiffAG produces an optimal solution (by using the optimal part of RAG-n on every iteration), a second algorithm is not needed. If we know *a priori* that many more than 2 adders per target will be needed (for example, this is typical of MCM problems with 2 constants on 32 bits), we could use only the H3 or H4 algorithm.

6.2 Depth Constrained MCM

In this section, we will modify the H3 from section 6.1.3 to satisfy a user-specified constraint on the adder depth. The new algorithm is denoted as $H3_d$. We introduced the adder depth problem in section 4.7.1. As illustrated in section 6.2.1, the depth constraint can be used to prune the search space. The SCM distance estimator is used in $H3_d$, so a depth constrained version of H(k)+ODP is needed. This will be discussed in section 6.2.2. In section 6.2.3, we will illustrate the *depth reordering* problem, which is similar to the vertex reordering problem that we encountered in H3 by enforcing $\mathcal{A}(x, y) \leq 2^b$. We will also show how we resolve this problem. Finally, experimental results are presented in section 6.2.4.

6.2.1 Using the Depth Constraint to Prune the Search Space

Let us first illustrate how we keep track of the adder depth. Let d_n denote the depth of node r_n. The source node (the $r_0 = 1$ node) corresponds to where the input is applied in the adder tree that implements the constant coefficient multiplier. This node has depth of $d_0 = 0$. On each iteration of $H3_d$, one adder is used to construct a new term (one node is added to the DAG). The DAG indicates how the input is added, subtracted and shifted with itself. Let us construct r_k as $r_k \in \mathcal{A}(r_i, r_j)$, where r_i and r_j are existing terms in R. The number of serial additions from the input to the r_k node must be one more than to get to either of r_i or r_j, thus the depth at node r_k is $d_k = \max(d_i, d_j) + 1$. An analogous technique can also be used for CSE.

In our implementation, we actually keep track of the *remaining depth*. If $\hat{d}$ is the depth constraint, then $\hat{d} - d_n$ is the remaining depth of node r_n. If a term can be constructed in more than one way, we only consider the maximum remaining depth.

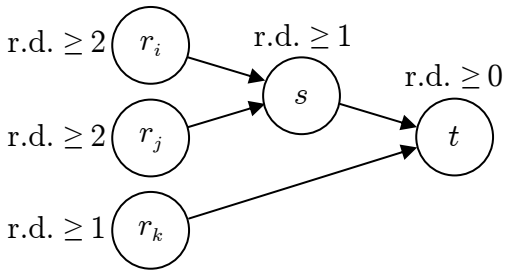


Figure 6.9: The depth constraint is used to prune graph tests. The target satisfies the depth constraint if r.d. ≥ 0 (r.d. is short for remaining depth). To satisfy this, only s and r_k which have r.d. ≥ 1 can be used. In the general case, assertions can be applied further backwards as needed.

When we construct $S = \mathcal{A}(R, R) \backslash R$, we can only use operands in R which have a remaining adder depth of at least 1. An element in $r' \in R$ may have a remaining depth of 0 (which often occurs if r' is a target), but if we construct anything *off of* r', the resulting terms will violate the depth constraint. If we construct a term with 0 remaining depth on the current iteration of H3_d, we do not need to update the successor set S for the next iteration because all of the *new* pairings of R violate the depth constraint. This is a form of pruning.

From in section 3.2.6, the distance 2 tests are $t/C_1 \cap S \neq \emptyset$ and $\mathcal{A}(t, R) \cap S \neq \emptyset$, which test for the topologies $t \in \mathcal{A}(s, s)$ and $t \in \mathcal{A}(r, s)$, respectively, where $r \in R$ and $s \in S$. Given that t must have a remaining depth of at least 0, then both r and s must have a remaining depth of at least 1. This is illustrated in Figure 6.9. Let R' and S' denote the set of elements in R and S, respectively, that have a remaining depth of at least 1. It follows that the depth constrained version of the distance 2 tests are $t/C_1 \cap S' \neq \emptyset$ and $\mathcal{A}(t, R') \cap S' \neq \emptyset$, respectively. Finding R' and S' requires a negligible amount of computation, as we cache the depth of every element in R and S. Since $R' \subseteq R$ and $S' \subseteq S$, the depth constrained distance 2 tests require less computation than the original distance 2 tests.

We use the depth constraint to prune the search space, so less computation is needed compared to the same distance test with no depth constraint. However, due to the depth constraint, the final solution may need more adders, so the *total* amount of computation to find the final solution is not necessarily less.

A similar pruning technique can be used for the exact distance 3 tests and the partial graph distance estimators. For example, consider the $Z = \mathcal{A}(t, s)$ estimator. The target is constructed as $t \in \mathcal{A}(z, s)$ where $z \in Z$. This is the same topology as shown in Figure 6.9, where z takes the place of r_k. Thus, we only consider s which have a remaining depth of at least 1. For distance 4, we check for a common element between Z and C_2. Since every element in C_2 has a depth of 2, we can construct $t \in \mathcal{A}(s, c_2)$ where $c_2 \in C_2$ and $c_2 \in Z$ if the depth constraint $\hat{d}$ is at least 3. For distances larger than 4, we evaluate the CSD cost of each element in Z. Since $t \in \mathcal{A}(z, s)$, we only consider z which have a remaining depth of 1. In other words, z must be constructible with a depth no larger than $\hat{d} - 1$. Based on the bounds presented in section 4.8.1, it follows that z cannot have more than $2^{\hat{d}-1}$ CSD digits, so the maximum CSD cost that z can have is $2^{\hat{d}-1} - 1$.

6.2.2 A Depth Constrained Version of H(k)+ODP

Assume we are given a depth constraint of $\hat{d}$. If the CSD form of a constant has m nonzero digits, then there exists a SCM solution with a depth no larger than $\lceil \log_2 m \rceil$, as shown in section 4.8.1. We can solve the MCM problem by solving a SCM problem for each target and then combining all of the terms in every SCM solution into one set R. Thus, if every target in a MCM problem has no more than $2^{\hat{d}}$ CSD digits (or a CSD cost of no more than $2^{\hat{d}} - 1$), we are guaranteed that a solution exists. We can

halve the number of CSD digits in a constant every time the adder depth increases by one, however the partial graph estimators in $H3_d$ have no mechanism to do this. In order to guarantee that $H3_d$ will find a solution if it exists, we will rely on the large distance estimator (traversing the SCM solutions found by H(k)+ODP).

Without considering the depth constraint, H(k)+ODP may find solutions that *happen* to satisfy the depth constraint. Clearly, if we impose a depth constraint, we can only *lose* some (or all) of the best solutions found by H(k)+ODP (in terms of minimizing adders). Only if none of the solutions found satisfy the depth constraint, then we consider an alternative.

We have observed that H(k)+ODP can typically find solutions with a depth of only one more than the absolute minimum. Thus H(k)+ODP typically violates the depth constraint only if depth constraint is the minimum in which a solution exists. Recall from section 4.8.1 that if the CSD form of the constant has m nonzero digits, the minimum depth is $\lceil \log_2 m \rceil$. Assuming a solution exists, one simple method which guarantees that we will produce a solution with the minimum adder depth is to apply a breadth-first collection of terms in the CSD form of the constant. For example, if the CSD form is $A0A00A0\overline{A}00A0A0\overline{A}$, we substitute all of $B = A0\overline{A}$, $C = A00\overline{A}$ and $D = A00A$ to get $A0000D0000\overline{C}000B$. Now that all of the lowest depth terms have been collected, we continue by substituting $E = C000\overline{B}$ and $F = A0000D$ to get $00000F00000000\overline{E}$, and so on.

In the above example, we have substituted patterns that only occured once, thus not reducing the CSD cost. However, we can still make one maximally occurring substitution per depth and still be able to collect terms in a minimal depth manner. For example, in $A0A00A0\overline{A}00A0A0\overline{A}$, we can substitute $B = A0A$ and $C = A000000A$ to

get $00B00A000000B0\overline{C}$. Since we have an odd number of nonzero digits, we will have one leftover lowest depth term (every possible collection of the lowest depth terms has been made). Now we substitute $D = B0\overline{C}$ and $E = B00A$ to get $00000E00000000D$. We have still collected the digits in a minimal depth manner, but by changing the *order in which digits are collected*, we obtain a solution with fewer adders.

Using a minimal depth collection of digits, we have observed that most of the minimum adder solutions can be obtained by substituting a maximally occurring pattern on *only* the first substitution in the CSD form of the constant. Since these SCM solutions are used for distance *estimation* in $H3_d$, in order to reduce the amount of computation, we do not consider other initial SD forms other than CSD and maximally occurring patterns are only considered for the first substitution. However, on the first substitution, we still substutite every maximally occurring pattern one at a time in order to produce a *variety* of solutions, which facilitates the sharing of intermediate terms between constants in $H3_d$.

6.2.3 The Depth Reordering Problem

6.2.3.1 An Introduction and an Example

The SCM solutions are computed at the beginning of the $H3_d$ algorithm and we only cache the ones that satisfy the depth constraint. Thus, $H3_d$ does not need to check the depth of the useful successors that enable us to traverse a SCM solution.

It is possible to construct a term in a SCM solution with less remaining depth than the amount of remaining depth it would have had if we traversed the SCM solution from the beginning. For example, a useful term v found by a distance 2 test for one target t_1 may also be in a SCM solution of another target t_2. After constructing v, if

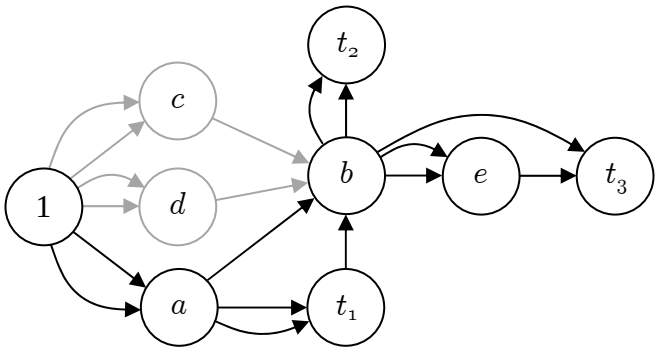


Figure 6.10: An example which illustrates the depth reordering problem.

we continue to traverse the SCM solution all the way up to the construction of t_2, we may not be able to satisfy the depth constraint. The partial graph estimators are not guaranteed to find a solution that satisfies the depth constraint, so this SCM solution could be the only known valid path for constructing t_2. Unless we fix the remaining depth of v, H3$_d$ will not be able to find a solution to this MCM problem.

The *depth reordering* problem arises from *partially* traversing a pre-computed solution, such as a SCM solution. To the best of our knowledge, no existing MCM algorithm allows the partial traversal of a pre-computed solution. Although a partial traversal uses less adders than a full traversal, it can create complications.

Let us illustrate the depth reordering problem in more detail with an example. Without a depth constraint, one solution to the MCM problem for $T = \{t_1, t_2, t_3\}$ is shown by the black part of Figure 6.10. Notice that t_3 has a depth of 5. Now suppose we enforce a depth constraint of 4. Assume there is only one way in which t_3 can be constructed with a depth of 4, which is the SCM solution consisting of the terms c, d, b and e in Figure 6.10. Both H3 and H3$_d$ reuse the weight benefit function (4.8) from section 4.3.3, and due to the $10^{-\text{dist}(R \cup \{s\}, t)}$ exponent in (4.8), the successor selected by the heuristic is mostly based on the closest targets. Thus,

since t_1 is cost 2 in Figure 6.10, we will first construct the terms a and t_1 (over two iterations of H3$_d$). From here, t_2 is now distance 2 and b is a useful successor, thus b and t_2 will be constructed. Without the depth constraint, given that b is constructed, t_3 would be distance 2. However, b has a remaining depth of 1 (the depth is 3 and the depth constraint is 4). This means that the successor e has a remaining depth of 0, so the depth constrained distance 2 tests will indicate that t_3 cannot be constructed with 2 more adders. Recall from section 6.1.3.5 that pre-computed solutions are only traversed up to distance 2 because the vertex reordering problem does not occur at distance 2. The only way to resolve the depth reordering problem is to create the terms c and d (which are shown in gray in Figure 6.10) so that the remaining depth of b can be updated to 2. Now that e has a remaining depth of 1, the distance 2 tests will determine that e is a useful successor. By first constructing c and d, we can then construct e and t_3 without violating the depth constraint.

6.2.3.2 Dynamically Updating the Depth

Two important observations can be made from the above example. Firstly, by having a smaller depth constraint, the final solution may have more adders. In addition to losing some ways of constructing a target, in order to reduce the depth of a term so that the depth constraint will later be satisfied, supporting terms must be added (these would not be needed if there was no depth constraint). Secondly, the depth of a term can change as new terms are later constructed.

In H3$_d$, we need to *dynamically* update the depth of *all* of the terms in R and S after each iteration. The process of updating the depth proceeds as follows. Let r' denote the term that was constructed at the end of the current iteration. As before,

$R_{new} = r' \cup R_{old}$, $S_{update} = (C_1 \cdot r') \cup \mathcal{A}(r', R_{old})$, and $S_{new} = (S_{old} \cup S_{update}) \backslash R_{new}$. For each $w \in S_{update}$, if w is also in S_{old} or in R_{old}, it is possible that the remaining depth of w has increased. In this case, we update the depth (for each term in R and S, we store the *maximum* remaining depth).

If a term in R_{old} has its depth updated, any term built off of it may also need its depth updated, hence this may cause a chain reaction. Before the chain reaction begins, only terms in S_{update} can have their depth updated, thus the chain reaction can only be started by an element in both R_{old} and S_{update}. We resolve this chain reaction recursively by treating every element that is in both R_{old} and S_{update} as if it were the newly constructed r' (a corresponding S_{update} for this element is constructed and we check for depth updates and new elements that can be treated like r', and so on). On each level of recursion, the remaining depth of an element treated like r' decreases by one (the remaining depth of any term in S_{update} is one smaller than the remaining depth of r', and only a term in both R_{old} and S_{update} can become the new r' on the next level of recursion). Therefore the number of recursions is bounded by the depth constraint $\hat{d}$ (all terms must have a remaining depth between 0 and $\hat{d}$).

6.2.4 Experimental Results

Much of the C code from H3 (section 6.1.3) was reused for $H3_d$, which was compiled with gcc 3.2.3. The benchmarks were performed on a 3.06 GHz Pentium 4 Xeon workstation running Linux. We used 1000 random MCM instances (with uniformly distributed random constants) with 8 and 48 constants and at bit widths of 15 and 24 bits. At each bit width and number of constants, we varied the depth constraint. Given a 15 bit constant, the bit width of the CSD form is at most 16 bits. This

can contain up to 8 nonzero digits, as there are no consecutive nonzero digits in the CSD form. It follows from section 4.8.1 that a solution with an adder depth of 3 is guaranteed to exist. Likewise, an adder depth of 4 can always be satisfied at 24 bits.

A comparison with the H3 algorithm is also performed. The results are presented in Figure 6.11. The horizontal lines are the results of H3. Since we keep track of the depth of every successor in $H3_d$, the minimum depth is used to break a tie when there are several equally most beneficial successors on any iteration in $H3_d$. Thus, even if the depth constraint is large enough that it has little or no influence on the solution, $H3_d$ will produce different solutions than H3. This is the primary reason why the average depth of the solutions found by $H3_d$ are notably lower than that of H3, even when the depth constraint is large.

Clearly, there is a stronger tradeoff between minimizing the adder depth versus minimizing the number of adders when the depth constraint is small. Sharing terms *between* targets can reduce the number of adders but not the depth. As the depth constraint decreases, we become forced to construct targets in ways that generally require more adders. For example, even if $t_i \in \mathcal{A}(t_j, t_k)$, unless both t_j and t_k have a remaining depth of at least 1, t_i will have to be constructed some other way, such as by traversing a SCM solution. If the depth constraint is too small, no solution may exist. No solutions were found when we tested $H3_d$ with a depth constraint of 2 at 15 bits. With a depth constraint of 3 at 24 bits, only 4 in 1000 solutions were found with 8 constants, and none were found with 48 constants.

Generally, as the depth constraint increases, the average number of adders in the solution of $H3_d$ decreases and eventually approaches that of H3. Interestingly, in the case of 8 constants on 24 bits, the average number of adders appears to remain

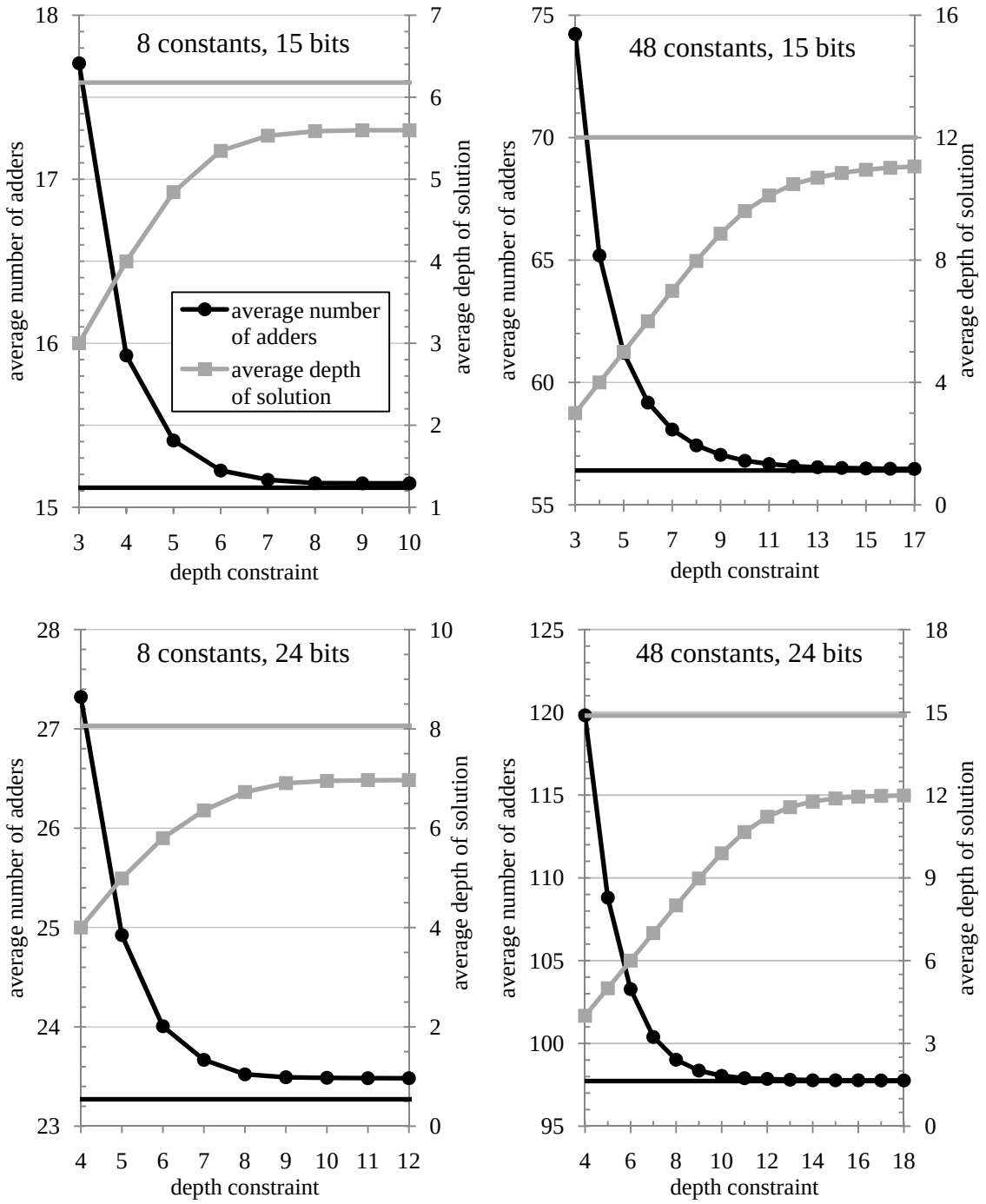


Figure 6.11: Experimental results of $H3_d$ as the depth constraint is varied for a fixed number of constants and bit width. The results of H3 are shown as horizontal lines, as H3 does not consider the adder depth.

higher than H3 even as the depth constraint gets large. When the depth constraint is large, the only difference between $H3_d$ and H3 is that the minimum depth is used to break the tie among equally beneficial successors on each iteration of $H3_d$. The choice made now may affect which useful terms are found later. Also, the traversal of the cached solutions found by the partial graph estimators is typically different. When we find the CSD cost of each element in $Z = \mathcal{A}(t, s)$, for example, we store the CSD solution of $z \in Z$ in a minimal depth manner (by collecting the terms in a breadth-first manner, as illustrated in section 6.2.2). Since the depth is used to break ties in $H3_d$, the solutions found by the partial graphs tend to be traversed through z first (instead of through s first) whereas there is no bias in H3. By traversing the cached solution through s first, we may be able to find a cheaper way to construct z compared to the CSD method, but then z would likely have less remaining depth.

Among all of the cases tested, 8 constants on 24 bits requires the most number of adders *per target*. In the other cases (with roughly 2 adders per target or less), distance estimation is infrequently used and thus there is little difference in the number of adders used by $H3_d$ and H3 when the depth constraint is large.

In Table 6.6, the average run times of $H3_d$ with and without the post removal of unnecessary intermediate terms is shown. The average run time of H3 is also shown. In H3, the post removal process typically requires 1 to 5% of the run time. Given that R is a valid solution, for each r such that $r \in R$ and $r \notin T$, we check whether $R \backslash \{r\}$ can be constructed without intermediate terms. We construct each term in $R \backslash \{r\}$ one term per iteration, but not necessarily in the original order. On each iteration in H3, we only need to find *one* way to construct *one* of the remaining terms, but in $H3_d$, we must evaluate *all* of the possible ways that *any* of the remaining terms can

Table 6.6: The average run time (seconds) of $H3_d$ and H3. H3 has no depth constraint. The values shown in brackets indicate the run time without performing the post removal of unnecessary intermediate terms. Beyond the highest depth shown, the run time changes insignificantly.

depth	8 constants		48 constants	
constraint	15 bits	24 bits	15 bits	24 bits
3	0.030 (0.023)	---	1.22 (0.20)	---
4	0.026 (0.023)	0.57 (0.51)	0.57 (0.14)	27.23 (6.67)
5		0.58 (0.55)	0.42 (0.14)	17.80 (6.28)
6		0.60 (0.58)	0.35 (0.15)	14.89 (6.33)
7				13.59 (6.33)
8				13.08 (6.31)
9				12.84 (6.30)
none (H3)	0.021	0.56	0.16	6.98

be constructed so that the term with the minimum depth can be selected. Clearly, this requires more computation. However, we try to maximize the chance that $R\backslash\{r\}$ will satisfy the depth constraint by taking a steepest descent approach (with respect to minimizing the depth) when constructing $R\backslash\{r\}$.

As shown in Table 6.6, the depth constrained post removal of unnecessary intermediate terms can require more run time than all of the other parts of $H3_d$. If we disregard the post removal process, $H3_d$ and H3 have similar run times. In both $H3_d$ and H3, the post removal process typically reduces the number of adders by 0.1 to 0.5%. The *absolute* run time of $H3_d$ with the post removal process is tolerable for small problem sizes, however the extra computation may not be worthwhile for large problem sizes. Note that the post removal process was used to generate the results shown in Figure 6.11. Although not shown, by using the post removal of terms, the adder depth typically increases by about 1% (in the cases where the depth constraint will still be satisfied).

6.3 Optimal MCM

In this section, we will propose an optimal branch and bound MCM algorithm. To the best of our knowledge, we are the first to introduce graph-based pruning and a bounding heuristic. Prior work is discussed in section 6.3.1. The bounding heuristics are discussed in section 6.3.2. In section 6.3.3, we will discuss the implications of performing an exhaustive search breadth-first versus depth-first and we will introduce our proposed graph-based pruning method for multiple constants. Finally, in section 6.3.4, we will present the experimental results as well as examine how the size of the MCM problem changes with respect to the bit width and the number of constants.

6.3.1 Prior Work

As discussed in section 4.1.3, [16--21] each propose an exact CSE-based algorithm that maximizes the sharing of intermediate terms using integer linear programming. CSE-based methods are inherently limited by the *representation* of the constant whereas graphs do not impose any restrictions. To the best of our knowledge, the only existing graph-based optimal MCM algorithm is BFS_{mcm} in [15] (short for breadth-first search MCM). Interestingly, the results of [15] show that Hcub, which is a *heuristic* graph-based algorithm, is closer to optimal than all of [16--21].

BFS_{mcm} does not use *pruning* but instead takes advantage of pre-computed sets. However, the number of pre-computed sets grows extremely quickly with respect to the MCM problem size. For example, at 13 bits, for 1, 2, 3, and 4 non-target terms, there are 25, 1188, 80907, and 458873308 pre-computed sets, respectively [15]. At 13 bits, some of the optimal solutions have 7 non-target terms, but the pre-computed sets for this would likely not fit in a hard drive with terabytes of storage (we suspect that [15]

happened to randomly select easy cases, as difficult cases are not reported in their benchmark). By non-target terms, we are referring to the supporting intermediate terms. Assume R is a valid MCM solution for the target set T, then $r \in R$ is a non-target term if $r \notin T$.

In our proposed optimal MCM algorithm BDFS (short for bounded depth-first search), we use pruning. We do not use precomputed sets since these are impractical to store, even for problem sizes that can be solved within a few minutes.

6.3.2 The Bounding Heuristic

Like the BIGE algorithm (optimal SCM, section 5.2), we also use a bounding heuristic in BDFS. Given a heuristic solution with n adders, the exhaustive search only needs to consider up to $n - 1$ adders. Because of this (as well as the pruning discussed throughout section 6.3.3), BDFS is more compute-efficient than BFS_{mcm} from [15].

In order for this approach to be viable, a nearly optimal bounding heuristic is needed. If the heuristic never produces a solution as good as the optimal, we may as well use only the exhaustive search to find the optimal solution. However, the heuristic requires little computation, so even if the heuristic only sometimes produces an optimal solution, the average run time of BDFS will decrease significantly. The exhaustive search requires exponential run time, thus not needing to search the last adder considerably reduces the computational effort.

As shown in [15], Hcub is close to optimal. In order to optimally solve a MCM problem in a reasonable amount of time, either the number of constants must be small or the bit width must be small. In MCM problems with a small bit width (and an arbitrary number of constants), both H3 and DiffAG generally outperform Hcub, thus

Table 6.7: A summary of the bounding heuristics used by BDFS.

Types of MCM problem	First heuristic	Second heuristic	When to use second heuristic
Small bit width (any number of constants)	DiffAG	H3	DiffAG requires ≥ 4 non-target terms
Few constants (small/moderate bit width)	H3	H4	H3 requires ≥ 5 non-target terms

we use the H3+DiffAG hybrid as the bounding heuristic for these MCM problems. When the number of constants is small, both H3 and H4 are used as the bounding heuristics (both typically outperform Hcub).

Recall from section 5.2.1.2 that one of reasons for creating the BIGE algorithm was to reduce the run time of H(k)+ODP. If we can confirm that H(1)+ODP has produced an optimal solution, H(2)+ODP is not needed. However, once we have confirmed that the solution requires *more than* 5 adders, we first tighten the heuristic bound (by using H(2)+ODP) before exhaustively searching at 6 adders. The heuristic bound requires little computation, and since it may eliminate the need to exhaustively search 6 adders, the average run time is reduced.

Since DiffAG, H3, and H4 do not have an adjustable parameter like H(k)+ODP, we emulate the above approach by using *two* bounding heuristics in BDFS. In MCM problems with a small bit width, DiffAG is used to find the initial upper bound since it is typically faster than H3. H3 is used to tighten the upper bound when we *estimate* that more computation would be needed to perform the exhaustive search. Thus, the optimal BDFS algorithm is sometimes faster than the heuristic H3+DiffAG algorithm. In MCM problems with only a few constants, H3 is used first and H4 may later be used to tighten the bound (again, BDFS is sometimes faster than H4). A summary of the heuristics used in BDFS is provided in Table 6.7.

Unlike H(k)+ODP, both H3 and DiffAG reuse the optimal part of RAG-n, so H3 and DiffAG may produce a solution that is *guaranteed* to be optimal. Obviously no further computation is done in BDFS when this happens. As stated in [15], in a MCM problem with $|T|$ targets, if a solution with $|T|$ or $|T|+1$ adders is produced by an algorithm that uses the optimal part of RAG-n, the solution is optimal. If a solution does not require intermediate terms, it will have $|T|$ adders. Using the optimal part of RAG-n, we will either find this solution or verify that it does not exist (in which case a solution with $|T|+1$ adders would be optimal).

6.3.3 An Exhaustive Search for Multiple Constants

6.3.3.1 Formulation of the Exhaustive Search

An exhaustive search can be done as follows. Starting with $R = \{1\}$, the optimal part of RAG-n is used repeatedly until none of the remaining targets are distance 1. At this point an intermediate term is needed. In order to cover every possible solution, for each successor $s \in S$, we construct s one at a time and then continue forward (repeatedly use the optimal part of RAG-n or construct every successor one at a time). This branching search continues until every target has been constructed. We could first process every s (breadth-first search) or first follow the solution until every target has been constructed (depth-first search). In both cases, the current depth of the search corresponds to how many adders have been used to construct the solution so far. Thus, we are interested in finding the solution with the minimum depth.

The solution space is exponential in size with respect to the depth. The base of the exponent corresponds to the typical branching factor (how many successors we need to consider when an intermediate term is needed). Recall that $S = \mathcal{A}(R, R) \backslash R$,

and there are many ways to shift and add *any* two elements in R. The base is usually a few times larger than the bit width, even after eliminating redundant searching (for example, we should create one of $R = \{1, 3, 5\}$ or $R = \{1, 5, 3\}$ but not both).

6.3.3.2 Breadth-First Versus Depth-First Exhaustive Searching

Due to the exponential size of the solution space, an impractical amount of memory is needed to cache solutions in a breadth-first search, even for problems that can be solved within a few hours. In addition to caching the terms constructed so far, we must also cache other sets to eliminate redundant searching.

A breadth-first search *without* caching can be emulated by performing an iterative depth-first search. For example, we would do a depth-first search up to depth 1, then up to depth 2, then depth 3, and so on. This requires a minimal amount of memory. Although we must recreate the solutions up to depth $n - 1$, this requires much less computation than searching at depth n since the solution space is exponential in size.

Alternatively, we could use a purely depth-first search. Given a heuristic solution with h adders, the depth-first search only needs to be done to depth $h - 1$. Every time a solution is found (assume it has m adders), the search is continued but only up to depth $m - 1$. This bound can be repeatedly tightened. Unlike the breadth-first in which the first solution found is the optimal, the depth-first search must go to completion. Even so, the remaining search (up to depth $m - 1$) will be much faster due to the exponential size of the solution space.

Although the purely depth-first search does not need to recreate solutions, it has a potential weakness. If the bounding heuristic is far from optimal, we may exhaustively search parts of the solution space with *more* adders than the optimal. This can be

very wasteful due to the exponential size of the solution space, however we have also used tighter bounding heuristics than Hcub.

To determine the better option, we implemented *both* the emulated breadth-first search and the purely depth-first search. It turns out that the purely depth-first search is generally faster in both the average and the worst case run time. Let n denote the optimal number of adders. In most of the cases we tested, the bounding heuristic found a solution as good as the optimal. When this happens, the exhaustive search will search the *entire* solution space up to $n-1$ adders, regardless of whether it is done depth-first or breadth-first. By using a purely depth-first search, we do not recreate solutions, thus saving run time. In the worst case (among the cases we tested), the bounding heuristic provided a solution with no more than $n+2$ adders. The purely depth-first exhaustive search initially considers solutions with $n+1$ adders, but the results infer that a solution with $n+1$ adders is found quickly so that the remaining search is done only up to n adders. Since there are many more solutions with $n+1$ adders than with n adders, it is much easier to find one with $n+1$ adders.

6.3.3.3 Elimination of Redundant Searching

As mentioned earlier, if 3 and 5 are intermediate terms, we should create one of $R = \{1, 3, 5\}$ or $R = \{1, 5, 3\}$, but not both. We will propose a method to prevent the creation of redundant R while maintaining an exhaustive search. In addition to caching the successor set S, we must also keep track of the *blocked successor set* $\hat{S}$.

Recall that the depth corresponds to how many adders have been used to construct the solution so far. We will use subscripts to denote the depth associated with a set, for example, R_d and S_d denote the ready and successor sets at depth d, respectively.

Suppose we are at depth d in the search and no target is distance 1, thus we must construct each $s \in S_d$ one at a time for depth $d+1$. It follows that $R_{d+1} = \{s\} \cup R_d$ and $S_{d+1} = (\mathcal{A}(s, R_{d+1}) \cup S_d) \backslash \{s\}$. Aside from removing s from the new S_{d+1}, everything that was in S_d will remain S_{d+1} (any term that could have been created with 1 adder will still be creatable after we add something to the set of existing terms R). Thus, if we have to construct all of the possible intermediate terms at depth $d+1$, we will cover some of the same paths that originate from depth d. However, some elements in $\mathcal{A}(s, R_{d+1})$ (which are the new pairings of R in the successor set) will not have been in S_d, so *new* paths will also emerge.

Given that s was constructed at depth d, we want to *block* redundant successors from being considered at depth $d+1$. At depth $d+1$, we define the blocked successor set as $\hat{S}_{d+1} = \{z \mid z \in S_d, z \leq s\} \cup \hat{S}_d$. Any successor that is currently blocked will remained blocked for the entire remaining branch in the search tree. We do not construct any of the blocked successors, but somewhere further up the branch in the search tree (at a lower depth) there is an equivalent *non-blocked* path that will produce the same set of terms. By blocking only values smaller than s, all *redundant* paths are forced to be traversed by constructing the terms in an ascending order.

In Algorithm 3, we describe a purely depth-first exhaustive search in which redundant R are prevented. Aside from eliminating redundant R, there is *no pruning* in Algorithm 3. Note that when the optimal part of RAG-n is used, there is no branching in the search, thus the blocked successor set is not updated for this.

As a example, when $R = \{1\}$, the successor set will contain both 3 and 5. If we first construct 3, we can then construct 5, thus $R = \{1, 3, 5\}$ is permitted. Conversely, if 5 is first constructed, 3 will be added to the blocked successor set thereby preventing

Input : the MCM targets T, the best existing solution H (initially heuristic)
Output: the optimal MCM solution for T

comment: the base case of the recursive function is called as follows
opt_MCM_sol = Search($\{1\}$, C_1, T, 0, get_heuristic_solution(T), $\emptyset$)

```
1  Search(R, S, T', depth, H, blocked_S) {
2      [check for optimal part of RAG-n]:
3      for each t ∈ T' {
4          if t ∈ S {
5              R ← {t} ∪ R
6              S ← A(t, R) ∪ S
7              T' ← T'\{t}                    comment: T' is the remaining targets
8              depth ← depth + 1
9              goto [check for optimal part of RAG-n]
           }
       }
10     if (T' = ∅) {
11         if (|R| < |H|) { H ← R }           comment: update best solution
       }
12     else if (depth < |H| − 1) {
13         for each s ∈ S AND s ∉ blocked_S {
14             R_new ← {s} ∪ R
15             S_new ← A(s, R_new) ∪ S
16             blocked_S_new ← {z | z ∈ S, z ≤ s} ∪ blocked_S
17             Search(R_new, S_new, T', depth + 1, H, blocked_S_new)
           }
       }
18     return H
   }
```

Algorithm 3: A depth-first exhaustive MCM search with redundant R eliminated (but no other pruning). Note that R and S are initialized to $\{1\}$ and C_1, respectively. H contains the best existing solution found so far and is initialized with one or more bounding heuristics (according to Table 6.7).

the creation of $R = \{1, 5, 3\}$. It follows that for any element x, $R = \{1, 5, 3, x\}$ and $R = \{1, 5, x, 3\}$ will never be considered (when 5 is first constructed, we *always* block 3, even in later searches). However, when $R = \{1\}$, 27 was not in the successor set and thus it cannot blocked no matter which term was constructed when $R = \{1\}$. Since $27x = ((3x) \ll 3) + (3x)$ and $27x = (x \ll 5) - (5x)$, both $R = \{1, 3, 27\}$ and $R = \{1, 5, 27\}$ are permitted.

6.3.3.4 Reducing the Computation When No Non-Target Terms Remain

Given $|T|$ unique targets and $|H|$ adders in the best existing solution, there are currently $|H| - |T|$ non-target terms. Since $T \subseteq R$ is a constraint in the MCM problem, in order to find a better solution, we can only have up to $|H| - |T| - 1$ non-target terms. If we have used up all of the $|H| - |T| - 1$ non-target terms, we try to achieve $T' = \emptyset$ by using lines 3-9 in Algorithm 3. In other words, we have to construct all of the remaining targets with one adder each.

This repeated use of the optimal part of RAG-n is identical to the reconstruction process within the post-removal of unnecessary intermediate terms (in section 6.1.3.6, given an existing solution R, for each intermediate term $r \in R$ where $r \notin T$, we try to *optimally* solve a new MCM problem with targets $T = R \backslash \{r\}$). As shown in section 6.1.3.6, instead of checking if $t \in S$, we use an equivalent but more compute-efficient test: $\mathcal{A}(t, R) \cap R \neq \emptyset$. Due to the symmetry in the adder-operation, the R in $\mathcal{A}(t, R)$ never needs to be shifted, as this shift can be absorbed by the R we intersect with.

We use the $\mathcal{A}(t, R) \cap R \neq \emptyset$ test in BDFS only once we have exhausted all of the $|H| - |T| - 1$ non-target terms. The final sequence of repeatedly using the optimal part of RAG-n is performed *without computing the successor set*. The modifications

to Algorithm 3 should be obvious. Since this process is in the innermost loop of the exhaustive search (at the leaves in the search tree), a significant improvement in the run time is obtained.

6.3.3.5 Pruning From One Non-Target Term Away

Assume the best existing solution has n non-target terms. Suppose in Algorithm 3 we have constructed $n-2$ non-target terms and there are no distance 1 targets (which means an intermediate term is needed). In order to find a *better* solution than the best existing solution, we can only construct *one more* non-target term.

Instead of blindly constructing all of the non-blocked successors, we can use pruning. A non-blocked successor $s \in S \backslash \hat{S}$ is *useless* if it cannot lead to the immediate construction of *at least one* of the remaining targets in T'. In other words, after constructing s, another intermediate term would still be needed. Thus, we do *not* construct any $s \in S \backslash \hat{S}$ in which $\text{dist}(R \cup \{s\}, t) > 2$ for all $t \in T'$. Conclusively, we will need to perform distance 2 tests but only for non-blocked successors $S \backslash \hat{S}$ (for all $t \in T'$, check if $(t/C_1) \cap (S \backslash \hat{S}) \neq \emptyset$ and if $\mathcal{A}(t, R) \cap (S \backslash \hat{S}) \neq \emptyset$).

If we have constructed $n-2$ non-target terms so far, the condition of line 13 of Algorithm 3 will now need to include distance 2 tests. We only construct *useful* non-blocked successors one at a time (lines 14-17 in Algorithm 3), however we do *not* update S or $\hat{S}$ because the next thing that will be computed is the final sequence of repeatedly using the optimal part of RAG-n (using $\mathcal{A}(t, R) \cap R \neq \emptyset$).

Both the distance 2 pruning and the final sequence of optimal tests require $\mathcal{A}(t, R)$ to be computed, thus it may be argued that distance 2 pruning is redundant since it is followed by optimal tests. However, we have observed that typically only a small

fraction of the non-blocked successors are useful (if any are useful at all). Distance 2 pruning involves computing $\mathcal{A}(t, R)$ *only once* for all $t \in T'$. Conversely, each non-blocked successor that is constructed is followed by the optimal tests, so $\mathcal{A}(t, R)$ for all $t \in T'$ is computed *every time* a non-blocked successor is constructed. Clearly, blindly constructing all of the non-blocked successors requires much more computation if only a small fraction of these are useful.

Distance 2 pruning is used near the innermost loop in BDFS (one level above the leaves in the exhaustive search tree). It often prevents the unnecessary use of the final sequence of optimal tests, hence it provides an extensive decrease in the run time.

6.3.3.6 Pruning From Several Non-Target Terms Away

Clearly, pruning is only considered when intermediate terms are needed. If the best existing solution has n non-target terms and we have constructed k non-target terms in the solution so far, we can use all of the distance m tests for $m \leq n - k$ in order to determine which non-blocked successors are useful.

For example, if we have already constructed $n - 3$ non-target terms, we can only construct *up to two more* non-target terms if we are to find a better solution. Thus, any non-blocked successor s is useless if it has $\text{dist}(R \cup \{s\}, t) > 3$ for all $t \in T'$ (this implies that *both* the distance 2 and the distance 3 tests found no solution). As before, we only construct useful non-blocked successors one at a time. Following each construction, the optimal part of RAG-n is used until no targets are distance 1. Assuming a solution is not found ($T' \neq \emptyset$), another intermediate term is needed. Now that $n - 2$ non-target terms have been constructed so far, we continue with distance 2 pruning as described in the previous section. Notice that this is similar to the *following*

the solution towards construction technique in the SBAC algorithm (section 5.3.2.3). However, since we have multiple constants in BDFS, we must introduce the optimal part of RAG-n in between the construction of each intermediate term.

Since the search is still *exhaustive*, we cannot prune more aggressively. For example, if we can have up to two more non-target terms, $\text{dist}(R \cup \{s\}, t) > 2$ for all $t \in T'$ is *not* sufficient to prove that s is useless (this only indicates that after s is constructed, at least one more intermediate term will be needed).

Distance 2, 3, and 4 tests are summarized in sections 3.2.6, 6.1.3.1, and 6.1.4, respectively. However, large distance tests provide little or no pruning if a target is close to R in terms of adder distance. For example, let us prune from distance 4. If there is a distance 2 target t, at least one successor s satisfies $\text{dist}(R \cup \{s\}, t) = 1$. However, *numerous* successors s' in which $\text{dist}(R \cup \{s'\}, t) \neq 2$ will likely satisfy $\text{dist}(R \cup \{s'\}, t) = 2$ or $\text{dist}(R \cup \{s'\}, t) = 3$ since it is much easier to construct a target with extra adders. These s cannot be pruned. A path longer than the minimum can be followed to construct t because this may assist in the construction of other targets. We are interested in the shortest *joint* path to construct *all* of the targets.

Due to the above limitation, we use distance 2 pruning in MCM problems where the bit width is small (arbitrary number of constants) and distance 3 pruning in MCM problems with only a few constants. The modifications to Algorithm 3 are obvious.

6.3.3.7 A Summary of BDFS

BDFS uses two bounding heuristics, a purely depth-first search, the elimination of redundant sets (using the blocked successor set), the $\mathcal{A}(t, R) \cap R \neq \emptyset$ test when zero non-target terms remain, and graph-based pruning using the adder distance tests.

6.3.4 Experimental Results

Our benchmark was performed on a set of identical 3.06 GHz Pentium 4 Xeon workstations running Linux. We implemented BDFS in C and we constrained $\mathcal{A}(x, y) \leq 2^{b+2}$, where b is the bit width of the largest target. Like the BIGE and SBAC algorithms (sections 5.2 and 5.3, respectively), this choice is arbitrary and more research is needed to establish to minimum bound to guarantee optimality.

The solution space (and thus the run time) is exponential and it is related to the number of non-output terms in the solution and the bit width. For each non-output term, branching is needed in the search, hence the number of non-output terms is the *power* in the exponent. The typical branching factor (how many non-blocked successors we need to consider) is the *base* of the exponent. As the bit width increases, the typical branching factor will also increase.

6.3.4.1 Comparison with $\mathbf{BFS}_{\mathrm{mcm}}$

We are able to solve problem sizes within a few minutes in which require an impractical amount of storage for pre-computed sets. We therefore cannot do a complete comparison against $\mathrm{BFS}_{\mathrm{mcm}}$ [15]. Furthermore, we are able to solve problem sizes within a few hours that require an impractical amount of memory for caching solutions in a breadth-first search. As explained in section 6.3.3.2, a breadth-first search *without* caching can be implemented with a depth-first search by incrementing the depth limit after each search, however a purely depth-first search is faster (we implemented and experimentally evaluated both types of searching). In conclusion, for non-trivial problem sizes, BDFS is *by construction* more compute-efficient than $\mathrm{BFS}_{\mathrm{mcm}}$ due to the tight bounding heuristic and the use of pruning.

In [15], the run times were only reported for a few examples in which the solutions had no more than 3 non-output terms. These problems require little computation to solve. We found the coefficients from [47] for one example in [15]. We were not able to obtain the same coefficients with the *remez* algorithm from MATLAB (as specified in [15, 47]), so we can only do a comparison with this example. The coefficients are $T = \{35, 266, 327, 398, 499, 505, 582, 662, 699, 710, 1943, 2987, 3395\}$. Using a 2.4 GHz Intel Core 2 quad-core processor, BFS_{mcm} requires 210.8 seconds [15] whereas BDFS requires 0.08 seconds (on a single core 3.06 GHz Pentium 4 Xeon).

6.3.4.2 MCM Problems with Small Bit Width Coefficients

We tested BDFS using 2, 4, 6, 8, 12, 16, 24, 32, 40, 50, 60, 70, 80, 90, and 100 constants on bit widths of 10-13 bits inclusive. Since the bit width is small, distance 2 pruning is used and the heuristic bounding function consists of DiffAG and H3. At each bit width and number of constants, we tested 100 MCM instances using uniformly distributed random constants. The average number of adders in the optimal solution is presented in Table 6.8. In Figures 6.12 and 6.13, we show the average and worst case bound on the number of non-output terms (found by the heuristic). As explained above, this provides an estimate of how much computation will be needed to optimally solve the problem. Also in Figures 6.12 and 6.13, we show how many *more* adders the heuristic uses compared to the optimal on average (although not shown, the worst case over all the cases we tested was 2 adders more than optimal, which only happened at 13 bits with between 6 and 40 constants inclusive).

The average and worst case run time of BDFS is provided in Table 6.9. When the *absolute* run time of BDFS is small, the run time is dominated by the bounding

Table 6.8: The average number of adders in BDFS.

number of constants	10 bits	11 bits	12 bits	13 bits
2	3.79	4.04	4.45	4.79
4	6.06	6.40	7.13	7.51
6	7.98	8.58	9.21	9.92
8	9.73	10.64	11.31	12.14
12	13.13	14.17	15.22	16.34
16	16.39	17.64	18.87	---
24	23.25	24.52	25.84	---
32	30.34	31.62	33.11	---
40	37.42	39.05	40.39	42.14
50	46.36	48.18	49.29	51.03
60	54.74	57.20	58.73	60.16
70	63.57	66.54	68.44	69.59
80	71.00	75.21	77.62	79.22
90	78.90	84.05	87.11	88.85
100	86.43	93.17	96.40	98.44

Table 6.9: The average and worst case run time (seconds) of BDFS. The run time at 10 bits is faster than at 11 bits, so it has little significance.

number of	11 bits		12 bits		13 bits	
constants	avg.	worst	avg.	worst	avg.	worst
2	<0.01	0.01	<0.01	0.01	0.01	0.04
4	<0.01	0.07	0.04	0.70	0.20	2.12
6	0.01	0.16	0.42	12.86	8.89	445.31
8	0.06	0.57	1.65	40.68	176.33	10372.00
12	0.17	6.65	20.23	727.83	2761.73	81264.00
16	0.05	1.65	8.77	291.10	---	---
24	<0.01	0.08	5.14	247.98	---	---
32	<0.01	0.01	0.08	5.78	---	---
40	<0.01	0.01	0.01	0.02	293.59	18498.00
50	<0.01	0.01	<0.01	0.01	0.71	49.31
60	<0.01	0.01	0.01	0.02	0.02	0.24
70	0.01	0.01	0.01	0.02	0.02	0.06
80	0.01	0.02	0.01	0.02	0.02	0.07
90	0.01	0.02	0.01	0.02	0.02	0.04
100	0.01	0.02	0.01	0.03	0.02	0.05

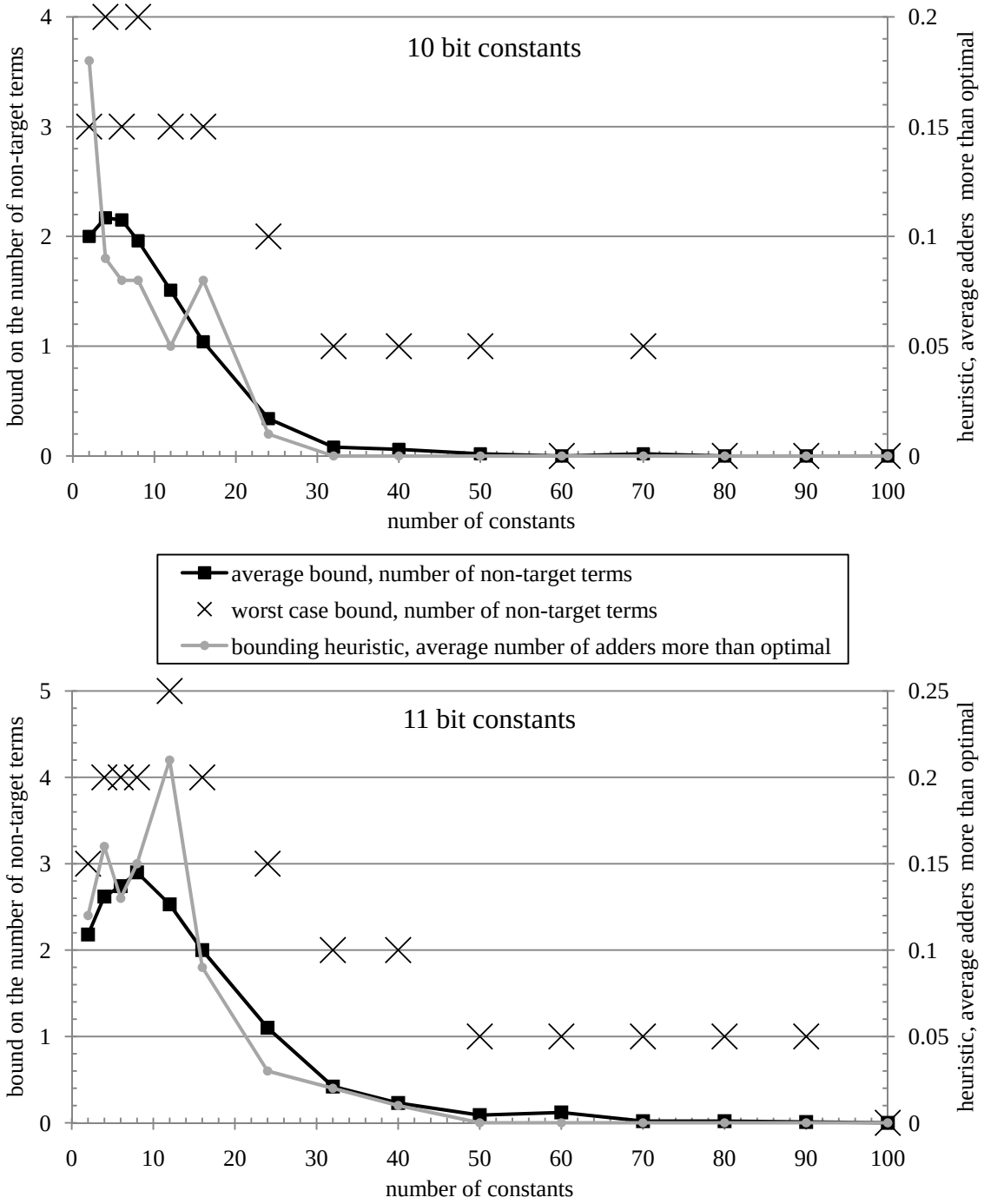


Figure 6.12: A comparison between the heuristic bound and the optimal part of BDFS for constants of small bit width, part 1 of 2.

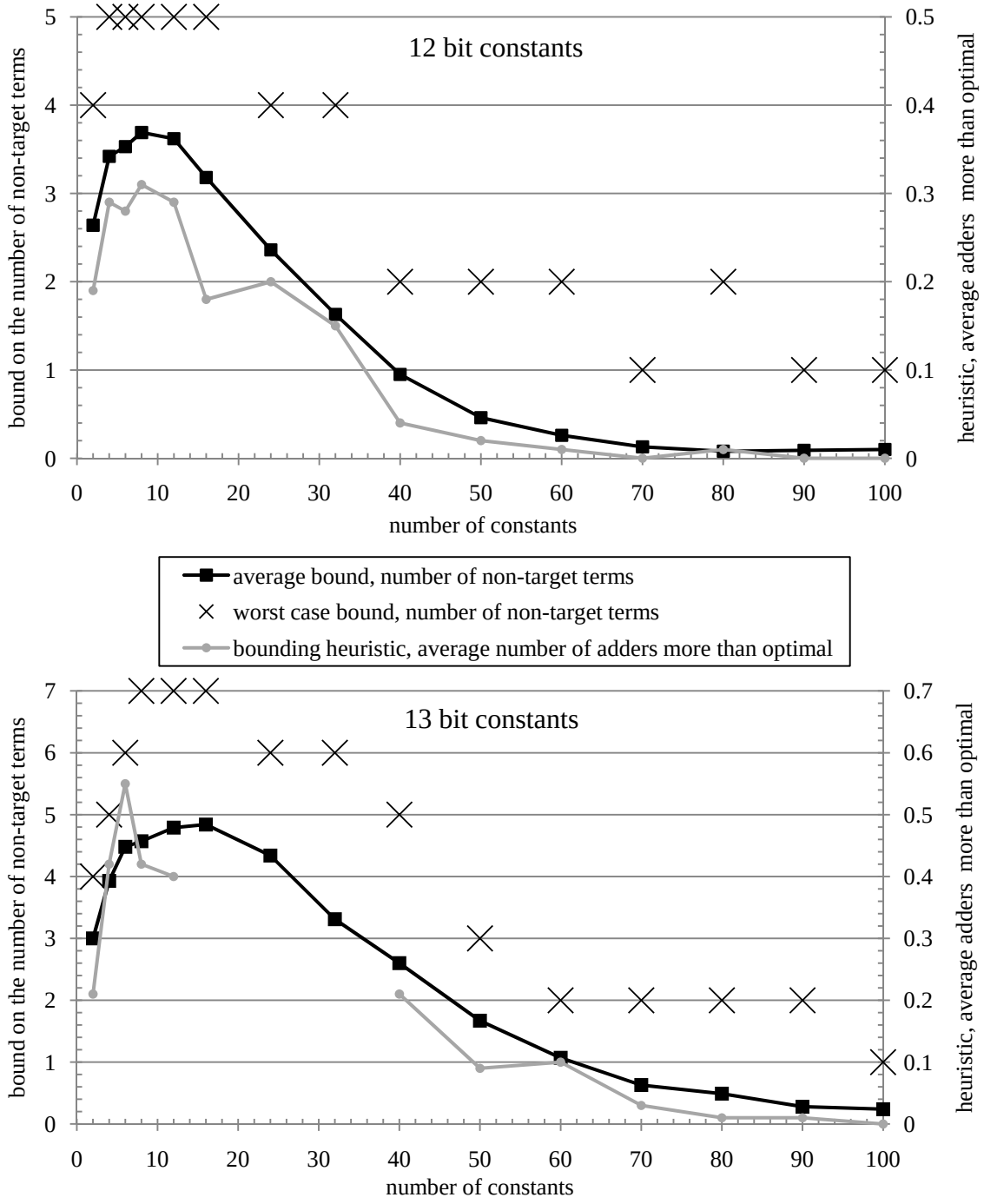


Figure 6.13: A comparison between the heuristic bound and the optimal part of BDFS for constants of small bit width, part 2 of 2.

heuristic (recall from section 6.3.2 that if the heuristic finds a solution with $|T|$ or $|T|+1$, this solution is optimal). For larger problem sizes, the bounding heuristic is still fast, so the time becomes dominated by the exhaustive search.

We set a time-out of 24 hours. All cases completed within the time limit except for at 13 bits with 16, 24, and 32 constants, in which only 93%, 92% and 97% of the cases completed, respectively. As an experiment, we let one case continue (13 bits, 16 constants) which resulted in a run time of 6 days. The heuristic was one off from the optimal, so the exhaustive search never considered solutions with more adders than the optimal. The optimal solution for this case had 6 non-output terms, which is *impractical* for BFS_{mcm} to solve. In [15], results using uniformly distributed random constants were reported on 14 bits with 20 constants (which is *more* difficult to solve than the cases in which BDFS timed out). We believe that either [15] happened to randomly select easy cases or some of the run times, especially the worst case, would have been on the order of weeks (the run times were not reported in [15]). This is a conservative estimate considering BFS_{mcm} does *not* use pruning.

The results on 14 bits are not shown in the thesis since many of the cases timed-out. At 14 bits, with 12, 16, 24, and 32 constants, the worst case heuristic bound on the number of non-target terms was 9. We estimate these cases would require a few days to weeks to compute with BDFS (and likely even longer with BFS_{mcm}).

For a fixed bit width, as we increase the number of constants, there exists a *saturation point* at which if the number of constants is further increased, the MCM problem becomes *easier* to solve. When there are numerous constants, we are able to build targets off of each other, thus the number of non-target terms may decrease as the number of constants increases. At 100 constants, for example, the MCM problem

is easy to optimally solve (at least for bit widths up to 13) and this is reflected by the small run time in Table 6.9. From Figures 6.12 and 6.13, for bit widths 10, 11, 12, and 13, it appears the saturation point is approximately 5, 8, 10, and 16 constants, respectively. As the bit width increases, the saturation point will increase.

Due to this saturation, it is feasible to optimally solve MCM problems up to 12 bits *for any number of constants* in practice. As shown in Table 6.9, the average time is up to 20 seconds and the worst case is about 12 minutes. BDFS was able to solve some problems with 7 non-output terms within one hour. If there are less than 6 non-output terms, BDFS will finish in at most a few hours (although the average run time is typically much faster). BDFS is fast when the number of non-target terms is small. This occurs when there are few or many constants, but not a moderate amount. Thus, given the appropriate number of constants, BDFS is fast enough to be used in practice at larger bit widths.

6.3.4.3 MCM Problems with Two Constants Only

On larger bit widths, we are only able to optimally solve MCM problems with very few constants within a reasonable amount of time. In these MCM problems, there are typically many adders *per target*, thus adding one more target causes the number of non-target terms to significantly increase. In order to maintain reasonable run times, this must be offset by considerably reducing the bit width.

We benchmarked up to 24 bits with only two constants. At each bit width, we used 100 MCM instances with uniformly distributed random constants. For these MCM problems, BDFS used distance 3 pruning and the bounding heuristic was composed of H3 and H4. The results are shown in Figure 6.14 and Table 6.10.

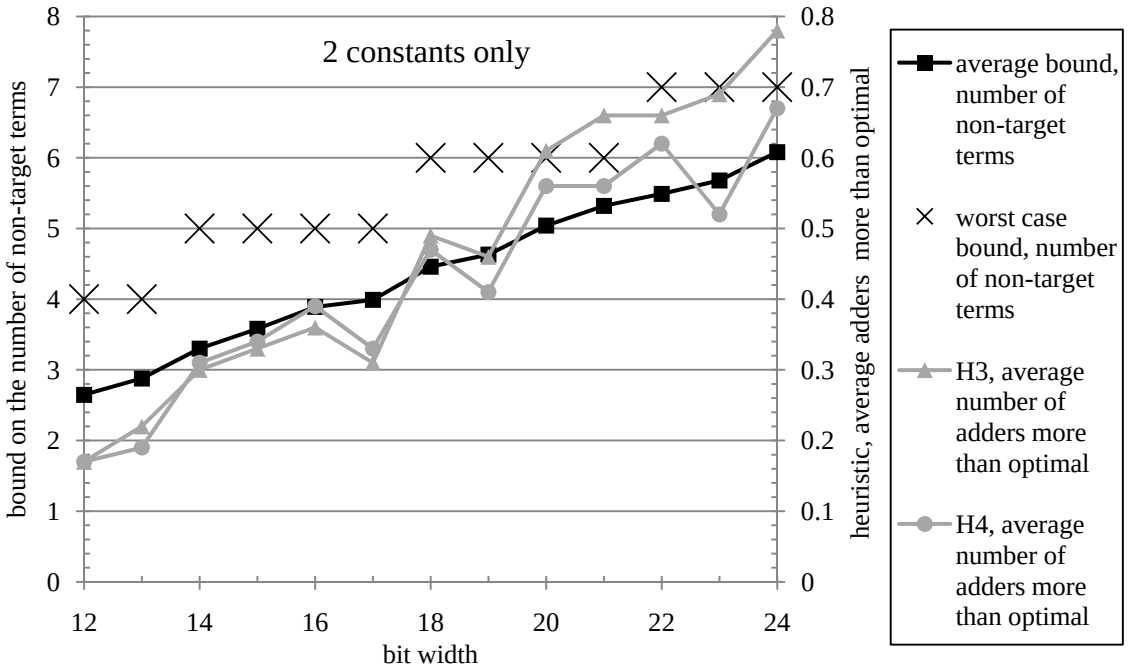


Figure 6.14: A comparison between the heuristic bound and the optimal part of BDFS for two constant MCM.

Table 6.10: Results for the BDFS benchmark with 2 constants.

bit width	optimal average number of adders	% of cases where heuristic ended up being optimal	run time (seconds)			
			optimal		H3	H4
			average	worst case	average	average
12	4.40	82%	0.002	0.02	0.002	0.008
13	4.64	80%	0.005	0.04	0.002	0.011
14	5.02	72%	0.012	0.1	0.003	0.018
15	5.28	72%	0.024	0.2	0.004	0.024
16	5.54	65%	0.069	0.95	0.006	0.031
17	5.75	76%	0.115	1.13	0.009	0.053
18	6.02	57%	0.342	3.38	0.012	0.088
19	6.26	63%	0.737	10.24	0.015	0.116
20	6.54	51%	2.934	42.42	0.028	0.261
21	6.80	50%	19.190	287.24	0.035	0.375
22	6.98	51%	103.582	1382.74	0.042	0.665
23	7.22	56%	210.508	6912.00	0.055	0.741
24	7.48	45%	1087.236	15672.00	0.076	1.479

The *optimal* BDFS algorithm is sometimes *faster* than *heuristic* H4. Like the BIGE algorithm (optimal SCM, section 5.2), this happens because the bounding heuristic may be used more than once and at different strengths. For example, in the BIGE algorithm, if we can prove that a solution found by H(1)+ODP is optimal, we do not need to use H(2)+ODP. From Table 6.10, up to a bit width 15 in BDFS, it is faster to check whether a solution found by H3 is optimal instead of using H4. As the bit width further increases, H4 becomes relatively less expensive due to the exponential size of the exhaustive search.

For two constants, BDFS is fast enough to use in practice for up to about 20 bits. As shown in Table 6.10, the bounding heuristic often finds a solution as good as optimal thereby facilitating a much smaller exhaustive search. The bounding heuristic was *always* within two adders of the optimal in the cases we tested.

We enforce $\mathcal{A}(x, y) \leq 2^{b+2}$ and vertex reduction is used in the distance 3 tests, thus like the BIGE algorithm, there is a negligibly small potential to miss some of the optimal solutions due to the vertex reordering problem (as illustrated in section 6.1.3.5). However, in our experimental evaluation, we did not observe any difference in the number of adders between using BDFS with distance 3 pruning and BDFS with distance 2 pruning (the vertex reordering problem cannot occur at distance 2 since the distance 1 and 2 tests cover all of the possible non-vertex reduced graphs).

6.3.4.4 The Size of the Solution Space in MCM

At the beginning of the experimental results we indicated how the size of the exhaustive solution space is exponential. Branching occurs in BDFS every time a non-target term is created, so the number of non-target terms is the *power* in the exponent. The *base*

of the exponent is the typical branching factor (how many non-blocked successors are considered when an intermediate term is needed). We will now examine how these change with respect to the number of constants and the bit width.

Assume the bit width is fixed. If we increase the number of constants, the total number of adders typically increases. When there are few constants, adding an extra constant generally increases the total number of adders by *more than one*, in which case the number of non-target terms will increase. Conversely, when there are many constants, increasing the number of adders by less than one results in a decrease in the number of non-target terms. There exists a maximum when the number of constants is moderate, which we earlier described as the *saturation point*. The branching factor will generally increase as the number of elements in R increases (since this creates more successors). This increases when there are more constants.

Our experimental run times (Table 6.9) indicate that the MCM problem size is typically the largest when there are slightly more constants than the saturation point. Before the saturation point, both the number of non-target terms and the branching factor are increasing. Just after the saturation point, we can infer that the branching factor increases the run time more than the number of non-target terms decreases the run time. If we continue further, eventually the effect of the number of non-target terms will become stronger, hence we observe a decrease in run time.

Now assume the number of constants is fixed. Clearly, both the number of adders and the number of non-target terms increase as the bit width increases. Since we constrain $\mathcal{A}(x, y) \leq 2^{b+2}$, the branching factor increases *faster* than the bit width b. In the worst case, the branching factor increases exponentially with respect to b, but in practice, this is limited by pruning techniques, such as the blocked successor

set. There is no saturation point for the bit width since we are always increasing the redundancy *within* constants (the number of non-target terms only decreases as the redundancy *between* constants gets large).

We conjecture that the MCM problem size generally increases faster with respect to the bit width than the number of constants, although there is no analytical proof. By adding more constants (which adds more elements to R), the branching factor likely increases no more than quadratically with respect to the number of constants since the number of successors is related to number of *pairings* in R ($S \subseteq \mathcal{A}(R, R)$). However, by increasing the bit width b, many more terms will be considered since the $\mathcal{A}(x, y) \leq 2^{b+2}$ constraint has been relaxed. Now let us consider the number of non-target terms. Assume there are currently n adders *per target*. If one more target is added to a MCM problem, the number of non-target terms will on average increase by $n - 1$. For MCM problems, usually n is small (3 or less in many cases). However, if we increment the bit width by 1, *several* targets will likely need an extra adder, thereby increasing the number of non-target terms by a larger extent. Also notice that no new targets are needed, thus the increase is purely in the number of non-target terms. In conclusion, the bit width generally has a larger impact on both contributing factors to the size of the exhaustive solution space in the MCM problem.

6.4 Concluding Remarks on MCM

In section 6.1, we have proposed two MCM heuristics H3 and H4 which are able to exploit the redundancy *within* constants better than any existing heuristic. MCM problems with more redundancy *between* constants are easier to solve optimally. As the compute power continues to increase in the future, optimal exhaustive searches will

be able to solve larger problem sizes within a reasonable amount of time. Algorithms that favor the redundancy between constants, such as DiffAG, will likely be the first to be replaced by optimal algorithms in the future. The MCM problem is more difficult to solve on large bit widths and this is where H3 and H4 obtain the most improvement over existing methods. Thus, it is expected that H3 and H4 will be the last heuristics to be replaced by an exhaustive search in the future.

For the MCM problems that are presently too large to solve optimally, our proposed H3+DiffAG hybrid is the best performing heuristic over the *entire* spectrum of MCM problem sizes. On average, H3+DiffAG produces better solutions in less run time than the best existing heuristic Hcub.

In section 6.2, we presented a depth-constrained version of H3. We illustrated the tradeoff between the adder depth and the number of adders and we showed how the depth constraint can be used to prune the search space. By reducing the depth, the logic circuit can be clocked faster, thus increasing the computational throughput.

Finally, in section 6.3, we proposed a graph-based depth-first exhaustive search to optimally solve MCM. To the best of our knowledge, we are the first to introduce a bounding heuristic, perform graph-based pruning with the adder distance tests, and prevent redundant searching by using the blocked successor set. Within a few hours, we can solve problems in which an infeasible amount of storage is needed for pre-computed sets and for caching solutions in a breadth-first search. Our proposed algorithm BDFS is more compute-efficient than the only existing exhaustive search BFS_{mcm}. BDFS can optimally solve MCM for two 24-bit constants in an average of only 18 minutes. Optimally solving this as well as other MCM problems that require 6 non-target terms was previously uncharted territory to the best of our knowledge.

Chapter 7

Conclusion

7.1 A Summary of the Contributions

By extending the analysis of prior work and providing new insight, in many cases our proposed algorithms are able to produce solutions in less run time with no more adders than the existing algorithms. In our optimal exhaustive algorithms, we have introduced aggressive pruning methods, namely the use of nearly-optimal bounding heuristics as well as IGT pruning for the exhaustive search. Due to this, we are able to solve larger problem sizes within a reasonable amount of time. Heuristics are needed for problems that are presently too large to exhaustively search in practice. In SCM, our proposed overlapping digit patterns facilitate a more efficient search, thus often enabling better solutions to be found with less initial SD forms (thereby decreasing the run time). In MCM, we proposed new methods for computing the adder distance in Hcub. Although our proposed heuristic H3 outperforms Hcub, it must be combined with the existing heuristic DiffAG in order to produce the best solutions on average over the *entire* spectrum of MCM problem sizes.

Table 7.1: A summary of the best existing SCM and MCM algorithms as of 2004. This summary was extracted from Table 2 in [11].

Problem	Criteria	Type	Algorithm	Framework
SCM	bit width ≤ 19	optimal	MAG [1, 2]	DAG
	bit width > 19	heuristic	H(k) [33]	CSE
MCM	few constants	heuristic	HHS [11]	CSE
	many constants	heuristic	RAG-n [14]	DAG

Table 7.2: A summary of the current best SCM and MCM algorithms (as of 2009). Note that DiffAG and H3 partition the space of the MCM problem sizes, however other algorithms may be used in special cases.

Problem	Criteria	Type	Algorithm	Framework
SCM	bit width ≤ 32	optimal	BIGE (§5.2)	DAG and CSE
	bit width > 32	heuristic	H(k)+ODP (§5.1 and [44])	enhanced CSE
MCM	bit width ≤ 12 or 2 const. & b.w. ≤ 20	optimal	BDFS (§6.3)	mostly DAG, a little CSE
	adders per target ≤ 2	heuristic	DiffAG [28]	DAG
	adders per target ≥ 2	heuristic	H3 (§6.1.3)	DAG and CSE
	few const. & large b.w.	heuristic	H4 (§6.1.4)	DAG and CSE

By modifying our proposed algorithms, we have addressed two additional problems. In section 5.3, we proposed a SCM algorithm which attempts to minimize the number of single-bit adders. This metric is more accurate than the number of adder-operations, so *less silicon* is needed to *implement* a constant multiplier. In section 6.2, we proposed a depth-constrained MCM algorithm. By reducing the adder depth, the logic circuit can be clocked faster, thus increasing the computational throughput.

In 2004, [11] presented a summary of the best existing SCM and MCM algorithms. In Table 7.1, we have extracted the relevant parts of this summary. In 2004, optimal SCM was limited to 19 bits and no optimal MCM algorithm had been proposed. In Table 7.1, the HHS algorithm is essentially H(k) applied to multiple constants (usually

with $k = 1$). Note that [11] did not provide a quantitative boundary for the MCM problem sizes in which RAG-n performs better than HHS or vice versa. However, both RAG-n and HHS are outperformed by newer algorithms, thus the location of this boundary is of little relevance.

As explained in section 6.1.1, CSE is better at exploiting the redundancy *within* each constant whereas DAGs are better at exploiting the redundancy *between* constants, hence the use of different heuristic algorithms for different problem sizes. This analysis was not provided in [11], as the summary in Table 7.1 was based purely on experimental results.

In Table 7.2, we have updated the summary of the best existing algorithms and we have refined the criteria for selecting an algorithm based on the problem size. Most of these algorithms use *both* the DAG and CSE framework and thus are able to take advantage of the strengths of both frameworks. The number of adders per target can be empirically estimated based on both the number of constants and the bit width, as shown in Figure 6.8 in section 6.1.5.2. DiffAG and H3 partition the space of the MCM problem sizes, however other algorithms may be used in special cases. BDFS can be used for larger problem sizes than that stated in Table 7.2 if one is willing to wait for a longer time. H4 on average outperforms H3 (as shown in section 6.1.5.1) but the run time of H4 prohibits its use on large problem sizes. Like BDFS, the cutoff in terms of a practical problem size depends on how long one is willing to wait. As discussed in section 1.2.2, this is highly application specific.

Table 7.2 mostly consists of our contributions. As explained in section 6.4, our heuristics are expected to be the *last* heuristics to be replaced by an optimal algorithm in the future (as the compute power continues to increase, larger problems

can be solved optimally). Only our H(k)+ODP algorithm has been published, as all of the other algorithms were developed *concurrently*. The optimal algorithms require high performance heuristics. Except for H(k)+ODP, *all* of the algorithms use the inverse graph traversal (IGT) method to perform graph-based pruning. Many computational enhancements have been progressively incorporated into the IGT method, especially for leapfrog graphs, which was illustrated in detail in section 5.2.2.5. Once the developments were finalized, this thesis was written first so that the Masters degree could be completed in a timely manner. In the near future, the contributions will be summarized and submitted for publication in order to facilitate an efficient dissemination of knowledge. This thesis will serve as a reference with additional examples and highly specific details to elaborate the contributions. All of the source code will be made publicly available.

7.2 Future Work

As future work, one may consider exploring how algorithms can be parallelized to take advantage of the emerging multi-core paradigm in desktop computers. In particular, exhaustive searches can easily be split into a set of smaller searches that are *mutually exclusive and collectively exhaustive*. By using a parallel search, optimal exhaustive algorithms can solve larger MCM problems within a reasonable amount of time. Although the BIGE algorithm (optimal SCM) requires an average of less than 10 seconds at 32 bits, the SBAC algorithm (SCM with single-bit adders) is presently too slow to use in practice at 32 bits. Thus, parallel searching may also enable one to use a *more accurate* metric and/or *several* metrics to solve a constant multiplication problem within a reasonable amount of time.

7.2.1 Minimization of Multiple and Less Abstracted Metrics

By using a more accurate metric (less abstraction), we expect to obtain better solutions in terms of minimizing the *absolute* metric, which is the amount of silicon required to realize constant coefficient multiplication in custom hardware. This was demonstrated in section 5.3.3, where we compared the BIGE algorithm (SCM) with the SBAC algorithm (single-bit adder SCM). However, extra computation is needed to encompass a less abstracted metric, hence the observed increase in the run time of SBAC compared to BIGE. Given the same problem sizes, we will be able to solve these with more accuracy within a reasonable amount of time as the compute power continues to increase in the future. As an example, consider minimizing the number of single-bit adders in a MCM problem subject to a user-defined constraint in terms of the critical path due to the carry chain (as opposed to the adder depth).

In order to increase the computational throughput of the logic circuit, one may consider the use of carry-save adders. However, this requires at least 3 operands to be added together, which may not occur in a multiplicative decomposition, for example. Clearly, certain DAG representations of SCM or MCM will be more favorable than others for carry-save adders. The topology of a Wallace or Dadda tree could also be considered in the SCM or MCM algorithm. In addition, one may consider pipelining the logic circuit to meet a given clock frequency constraint. This would involve *joint minimization* of registers and adders (the user could specify how important each of these resources are). To minimize the number of registers, the insertion of registers in the adder tree should occur at narrow locations.

Finally, one may also consider a more accurate model by considering the *routing* of logic resources within the SCM or MCM algorithm. This would enable one to account

for wiring delays in addition to the propagation delay caused by gates, for example. Although shifts incur no cost in custom hardware since they are hardwired, they must still be routed and therefore they will consume some silicon in the logic circuit. To account for this, one could introduce an appropriate metric that is a function of the shifts used in the adder-operation, for instance.

7.2.2 Parallelization of an Exhaustive Search

There are many ways in which an exhaustive search can be partitioned. In SCM, recall from section 5.2.2.1 that in order to determine if the target can be constructed with m adders, we construct every possible R with $m-k$ adders, and then for each of these R, we apply distance k tests. Each distance k test involves computing adder-operations, dividing by C_n, and performing set intersections where one set is sorted. All of these tasks are easy to parallelize due to their deterministic nature. Each distance k test is independent of all of the other distance k tests. The generation of each R set with n adders is independent from all of the other R sets with n adders (each of these is constructed from a R set with $n-1$ adders and this process is easy to parallelize). Clearly, there are many levels at which parallelization can be done.

In the MCM problem, the exhaustive search is formulated slightly differently. The optimal part of RAG-n can be used repeatedly in between the construction of each non-target term, but ultimately, the computation involved is similar to the generation of the R sets in SCM. The distance tests are still used for pruning. As before, the search can be parallelized at the adder-operation level, or we can compute each distance test in parallel, or we can assign different branches in the search tree to different processing cores. The higher in the search tree that we decide to split the

search, the less memory needs to be shared between different execution threads, but we may lose parallelism if we split the search too early. For example, given 100 cores, if we split the *entire* search into 100 parts, some execution threads will invariably finish before others. Due to the pruning methods, the amount of time needed to compute a branch in the search tree is impossible to know ahead of time. For instance, some adder distance tests involve division by C_n, and only the elements which divide C_n with zero remainder will continue in the search.

We have not implemented any parallel exhaustive searching algorithms for this thesis. Finding the appropriate degree of parallelism and establishing an efficient scheduling and synchronization of the searches over multiple computing cores is not a trivial task. More importantly, most desktop computers currently have up to four computing cores and thus a search *only up to* four times larger can be completed within the same amount of time. Due to the *exponential* size of the solution space, having only four cores does not help much. However, it is expected that multi-core computers will have tens to hundreds of computing cores in the future, thus facilitating the search of a much larger solution space within a reasonable amount of time.

One could also consider the use of a graphic processing unit (GPU). GPUs offer more parallelism than multi-core computers, but due to bandwidth limitations, highly localized computation is needed to obtain the maximum benefit from the parallelism.

In conclusion, as the compute power continues to increase in the future, optimal algorithms will be able to solve larger problem sizes in practice. However, algorithms will need to be parallelized in order to take advantage of future computing platforms. Designing algorithms that are well suited for multi-core computers and/or GPUs remains an open topic for future research.

Bibliography

[1] A. Dempster and M. Macleod, ''Constant integer multiplication using minimum adders,'' *IEE Proceedings on Circuits, Devices and Systems*, vol. 141, no. 5, pp. 407--413, Oct 1994.

[2] O. Gustafsson, A. Dempster, and L. Wanhammar, ''Extended results for minimum-adder constant integer multipliers,'' in *IEEE International Symposium on Circuits and Systems (ISCAS), 2002*, vol. 1, 2002, pp. I--73--I--76 vol.1.

[3] Altera Corporation. ''NIOS II processor: the world's most versatile embedded processor.'' Feb 2009. [Online] `http://www.altera.com/products/ip/processors/nios2/`.

[4] H. Wu and M. Hasan, ''Closed-form expression for the average weight of signed-digit representations,'' *IEEE Transactions on Computers*, vol. 48, no. 8, pp. 848--851, Aug 1999.

[5] G. W. Reitwiesner, ''Binary arithmetic,'' *Advances in Computers*, vol. 1, pp. 231--308, 1960.

[6] Y. Voronenko and M. Püschel, ''Multiplierless multiple constant multiplication,'' *ACM Transactions Algorithms*, vol. 3, no. 2, p. 11, 2007.

[7] D. Bull and D. Horrocks, ''Primitive operator digital filters,'' *IEE Proceedings on Circuits, Devices and Systems*, vol. 138, no. 3, pp. 401--412, Jun 1991.

[8] P. Cappello and K. Steiglitz, ''Some complexity issues in digital signal processing,'' *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 32, no. 5, pp. 1037--1041, Oct 1984.

[9] M. R. Garey and D. S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[10] A. Matsuura and A. Nagoya, ''Formulation of the addition-shift-sequence problem and its complexity,'' in *ISAAC '97: Proceedings of the 8th International Symposium on Algorithms and Computation*. London, UK: Springer-Verlag, 1997, pp. 42--51.

[11] A. Dempster, M. Macleod, and O. Gustafsson, ''Comparison of graphical and subexpression methods for design of efficient multipliers,'' in *Conference Record of the Thirty-Eighth Asilomar Conference on Signals, Systems and Computers, 2004*, vol. 1, Nov. 2004, pp. 72--76 Vol.1.

[12] O. Gustafsson, A. Dempster, M. Macleod, K. Johansson, and L. Wanhammar, ''Simplified design of constant coefficient multipliers,'' *Circuits, systems, and signal processing*, vol. 25, no. 2, pp. 225--251, 2006.

[13] A. Avizienis, ''Signed-digit number representation for fast parallel arithmetic,'' *IRE Transactions on Electronic Computers*, vol. 10, pp. 389--400, Sept 1961.

[14] A. Dempster and M. Macleod, ''Use of minimum-adder multiplier blocks in FIR digital filters,'' *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 42, no. 9, pp. 569--577, Sep 1995.

[15] L. Aksoy, E. Gunes, and P. Flores, ''An exact breadth-first search algorithm for the multiple constant multiplications problem,'' in *NORCHIP, 2008*, Nov. 2008, pp. 41--46.

[16] O. Gustafsson and L. Wanhammar, ''ILP modelling of the common subexpression sharing problem,'' in *9th International Conference on Electronics, Circuits and Systems, 2002*, vol. 3, 2002, pp. 1171--1174 vol.3.

[17] P. Flores, J. Monteiro, and E. Costa, ''An exact algorithm for the maximal sharing of partial terms in multiple constant multiplications,'' in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2005*, Nov. 2005, pp. 13--16.

[18] L. Aksoy, E. Costa, P. Flores, and J. Monteiro, ''Minimum number of operations under a general number representation for digital filter synthesis,'' in *18th European Conference on Circuit Theory and Design (ECCRD), 2007*, Aug. 2007, pp. 252--255.

[19] L. Aksoy, E. O. Gunes, E. Costa, P. Flores, and J. Monteiro, ''Effect of number representation on the achievable minimum number of operations in multiple constant multiplications,'' in *IEEE Workshop on Signal Processing Systems, 2007*, Oct. 2007, pp. 424--429.

[20] L. Aksoy, E. da Costa, P. Flores, and J. 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*, vol. 27, no. 6, pp. 1013--1026, June 2008.

[21] Y.-H. Ho, C.-U. Lei, H.-K. Kwan, and N. Wong, ''Global optimization of common subexpressions for multiplierless synthesis of multiple constant multiplications,'' in *Asia and South Pacific Design Automation Conference (ASPDAC), 2008*, March 2008, pp. 119--124.

[22] R. Bernstein, ''Multiplication by integer constants,'' *Software -- Practice & Experience*, vol. 16, no. 7, pp. 641--652, 1986.

[23] A. Dempster and M. Macleod, ''General algorithms for reduced-adder integer multiplier design,'' *Electronics Letters*, vol. 31, no. 21, pp. 1800--1802, Oct 1995.

[24] O. Gustafsson and L. Wanhammar, ''A novel approach to multiple constant multiplication using minimum spanning trees,'' in *The 2002 45th Midwest Symposium on Circuits and Systems (MWSCAS)*, vol. 3, Aug. 2002, pp. III--652--III--655 vol.3.

[25] H. Ohlsson, O. Gustafsson, and L. Wanhammar, ''Implementation of low complexity FIR filters using a minimum spanning tree,'' in *Proceedings of the 12th IEEE Mediterranean Electrotechnical Conference (MELECON), 2004*, vol. 1, May 2004, pp. 261--264 Vol.1.

[26] O. Gustafsson, H. Ohlsson, and L. Wanhammar, ''Improved multiple constant multiplication using a minimum spanning tree,'' in *Conference Record of the*

Thirty-Eighth Asilomar Conference on Signals, Systems and Computers, 2004, vol. 1, Nov. 2004, pp. 63--66 Vol.1.

[27] H. Choo, K. Muhammad, and K. Roy, ''Complexity reduction of digital filters using shift inclusive differential coefficients,'' *IEEE Transactions on Signal Processing*, vol. 52, no. 6, pp. 1760--1772, June 2004.

[28] O. Gustafsson, ''A difference based adder graph heuristic for multiple constant multiplication problems,'' in *IEEE International Symposium on Circuits and Systems (ISCAS), 2007*, May 2007, pp. 1097--1100.

[29] R. Hartley, ''Optimization of canonic signed digit multipliers for filter design,'' in *IEEE International Sympoisum on Circuits and Systems, 1991*, Jun 1991, pp. 1992--1995 vol.4.

[30] ------, ''Subexpression sharing in filters using canonic signed digit multipliers,'' *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 43, no. 10, pp. 677--688, Oct 1996.

[31] R. Pasko, P. Schaumont, V. Derudder, S. Vernalde, and D. Durackova, ''A new algorithm for elimination of common subexpressions,'' *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 1, pp. 58--68, Jan 1999.

[32] V. Lefèvre, ''Multiplication by an integer constant,'' in *INRIA*. RR-4192, May 2001, pp. 1--17.

[33] A. Dempster and M. Macleod, ''Using all signed-digit representations to design single integer multipliers using subexpression elimination,'' in *Proceedings of the*

2004 International Symposium on Circuits and Systems (ISCAS), 2004, vol. 3, May 2004, pp. III--165--8 Vol.3.

[34] I.-C. Park and H.-J. Kang, ''Digital filter synthesis based on an algorithm to generate all minimal signed digit representations,'' *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1525--1529, Dec 2002.

[35] A. Dempster and M. Macleod, ''Generation of signed-digit representations for integer multiplication,'' *IEEE Signal Processing Letters*, vol. 11, no. 8, pp. 663--665, Aug. 2004.

[36] L. Aksoy and E. O. Gunes, ''An approximate algorithm for the multiple constant multiplications problem,'' in *SBCCI '08: Proceedings of the 21st Annual Symposium on Integrated Circuits and System Design.* New York, NY, USA: ACM, 2008, pp. 58--63.

[37] C.-Y. Yao, H.-H. Chen, T.-F. Lin, C.-J. Chien, and C.-T. Hsu, ''A novel common-subexpression-elimination method for synthesizing fixed-point FIR filters,'' *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 51, no. 11, pp. 2215--2221, Nov. 2004.

[38] H.-J. Kang and I.-C. Park, ''FIR filter synthesis algorithms for minimizing the delay and the number of adders,'' *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 48, no. 8, pp. 770--777, Aug 2001.

[39] K. Johansson, O. Gustafsson, and L. Wanhammar, ''A detailed complexity model for multiple constant multiplication and an algorithm to minimize the

complexity,'' in *Proceedings of the 2005 European Conference on Circuit Theory and Design, 2005*, vol. 3, Aug.-2 Sept. 2005, pp. III/465--III/468 vol. 3.

[40] ------, ''Bit-level optimization of shift-and-add based FIR filters,'' in *14th IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2007*, Dec. 2007, pp. 713--716.

[41] L. Aksoy, E. Costa, P. Flores, and J. Monteiro, ''Optimization of area in digital FIR filters using gate-level metrics,'' in *DAC '07. 44th ACM/IEEE Design Automation Conference, 2007*, June 2007, pp. 420--423.

[42] O. Gustafsson, ''Lower bounds for constant multiplication problems,'' *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, no. 11, pp. 974--978, Nov. 2007.

[43] V. Lefèvre, ''Multiplication by an integer constant: Lower bounds on the code length,'' in *Proceedings of the 5th Conference on Real Numbers and Computers*, École Normale Suprieure de Lyon, France, Sep. 2003, pp. 131--146.

[44] J. Thong and N. Nicolici, ''Time-efficient single constant multiplication based on overlapping digit patterns,'' *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 9, pp. 1353--1357, Sept. 2009.

[45] V. Lefèvre. ''Multiplication by integer constants.'' June 2009. [Online] `http://www.vinc17.org/research/mulbyconst/index.en.html`.

[46] Spiral Website. ''Spiral multiplier block generator.'' Jan. 2009. [Online] `http://spiral.ece.cmu.edu/mcm/gen.html`.

[47] A. Dempster, S. Dimirsoy, and I. Kale, ‘‘Designing multiplier blocks with low logic depth,’’ in *IEEE International Symposium on Circuits and Systems (ISCAS), 2002*, vol. 5, 2002, pp. V--773--V--776 vol.5.

Index