Multiple Constant Multiplication Optimization Using Common Subexpression Elimination and Redundant Numbers

Firas Ali Jawad Al-Hasani

A thesis presented for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering
at the
University of Canterbury,
Christchurch, New Zealand.

March 2014

ABSTRACT

The multiple constant multiplication (MCM) operation is a fundamental operation in digital signal processing (DSP) and digital image processing (DIP). Examples of the MCM are in finite impulse response (FIR) and infinite impulse response (IIR) filters, matrix multiplication, and transforms. The aim of this work is minimizing the complexity of the MCM operation using common subexpression elimination (CSE) technique and redundant number representations. The CSE technique searches and eliminates common digit patterns (subexpressions) among MCM coefficients. More common subexpressions can be found by representing the MCM coefficients using redundant number representations.

A CSE algorithm is proposed that works on a type of redundant numbers called the zero-dominant set (ZDS). The ZDS is an extension over the representations of minimum number of non-zero digits called minimum Hamming weight (MHW). Using the ZDS improves CSE algorithms' performance as compared with using the MHW representations. The disadvantage of using the ZDS is it increases the possibility of overlapping patterns (digit collisions). In this case, one or more digits are shared between a number of patterns. Eliminating a pattern results in losing other patterns because of eliminating the common digits. A pattern preservation algorithm (PPA) is developed to resolve the overlapping patterns in the representations.

A tree and graph encoders are proposed to generate a larger space of number representations. The algorithms generate redundant representations of a value for a given digit set, radix, and wordlength. The tree encoder is modified to search for common subexpressions simultaneously with generating of the representation tree. A complexity measure is proposed to compare between the subexpressions at each node. The algorithm terminates generating the rest of the representation tree when it finds subexpressions with maximum sharing. This reduces the search space while minimizes the hardware complexity.

A combinatoric model of the MCM problem is proposed in this work. The model is obtained by enumerating all the possible solutions of the MCM that resemble a graph called the demand graph. Arc routing on this graph gives the solutions of the MCM problem. A similar arc routing is found in the capacitated arc routing such as the winter salting problem. Ant colony optimization (ACO) meta-heuristics is proposed to traverse the demand graph. The ACO is simulated on a PC using Python programming language. This is to verify the model correctness and the work of the ACO. A parallel simulation of the ACO is carried out on a multi-core super computer using C++ boost graph library.

ACKNOWLEDGEMENTS

I want to express my thanks and gratitude to my supervisors, Dr. Andrew Bainbridge-Smith, Professor Rick Millane, and Dr. Michael Hayes for their guidance, efforts, and support. All the knowledge and skills that I have acquired are returned to their grace. My deep thanks to Dr. Lucy Johnston the Dean of Postgraduate Research for her support and care of my study. I would like to thank Canterbury University for supporting me. It is a great institution that puts the students in the first priority. My work on the supercomputer BlueFern was a great opportunity for me. Thank you BlueFern staff, thank you Dr. Céline Cattoën-Gilbert and Dr. Francois Bissey. These two people provided me with all the support and knowledge in parallel processing. When I was stuck in a problem, Dr. Bissey used to answer my inquiry even in after work hours.

Thanks to my wife and son Mohanad for their support and encouragement. To my room colleagues Alex Opie and Peter Raffensperger, I remember your help when I first enrolled and came to a new environment.

CONTENTS

LIST OF FIGURES x

LIST OF TABLES xix

CHAPTER 1 INTRODUCTION 1
- 1.1 Multiplication without Multiplication 4
- 1.2 Multiple Constant Multiplication 9
- 1.3 Multiplierless Multiple Constant Multiplication 12
- 1.4 Common Subexpression Elimination 15
- 1.5 Radix Number System 17
- 1.6 Finding Subexpressions from Representations 17
- 1.7 Coefficient's Cost and Adder Step 19
- 1.8 Lower Bound and Optimality 20
- 1.9 Chapter Summary 21
- 1.10 Contributions in this Work 22
- 1.11 Thesis Overview 23

CHAPTER 2 COMMON SUBEXPRESSION ELIMINATION 27
- 2.1 Graph Dependent Methods 28
  - 2.1.1 The N-Dimensional Reduced Adder Graph Algorithm 34
- 2.2 Common Subexpression Elimination 36
- 2.3 Hartley's Method 37
  - 2.3.1 Horizontal Common Subexpression Elimination 38
  - 2.3.2 Vertical Common Subexpression Elimination 42
  - 2.3.3 Oblique Common Subexpression Elimination 43
  - 2.3.4 Mixed Common Subexpression Elimination 43
- 2.4 Multiplier Block Common Subexpression Elimination 45
  - 2.4.1 Representation Independent Common Subexpression Elimination 45
  - 2.4.2 Multiplier Block Horizontal Common Subexpression Elimination 46
- 2.5 Chapter Summary 49

CHAPTER 3 PATTERN PRESERVATION ALGORITHM 51
- 3.1 Zero-Dominant Set Generation 51
- 3.2 Zero-Dominant Set Versus Canonical Signed Digit 53
- 3.3 Pattern Preservation Algorithm 55
- 3.4 Algorithm Complexity 59
- 3.5 Results 59

3.6 Chapter Summary 62

CHAPTER 4 REDUNDANT NUMBERS 65
4.1 Classification of Radix Number systems 66
4.2 Polynomial Ring 68
4.3 Radix Polynomials 69
4.4 Digit Sets for Radix Representation 70
4.5 Necessary and Sufficient Conditions for Complete Residue System Modulo Radix 72

CHAPTER 5 GENERATING THE REDUNDANT REPRESENTATIONS 75
5.1 Representation Tree Algorithm 75
5.2 Graph Encoder 80
5.3 Representation Tree Size 83
5.4 Investigating the Performance of the Graph and Tree Algorithms 87
5.5 Chapter Summary 94

CHAPTER 6 SUBEXPRESSION TREE ALGORITHM 97
6.1 Subexpression Space 98
6.2 $\mathcal{A}-$operation 99
6.3 Subexpression Tree Algorithm 100
6.3.1 Subexpressions Determination 101
6.3.2 Decomposition Complexity 103
6.3.3 Multiple Iterations 106
6.3.4 Search Space Size 107
6.4 Results 109
6.5 Chapter Summary 119

CHAPTER 7 COMBINATORIAL MODEL OF MULTIPLE CONSTANT MULTIPLICATION 121
7.1 Arc Routing Problems 121
7.1.1 The Chinese Postman Problem 122
7.1.2 The Capacitated Chinese Postman Problem 123
7.1.3 The Capacitated Arc Routing Problem 124
7.1.4 Winter Gritting (Salting) Problem 124
7.1.5 Dynamic Winter Gritting (Salting) Problem 124
7.2 Decomposition, Demand, and Augmented Demand Graphs 125
7.3 Dynamic Winter Gritting Analogy 130
7.4 Ant Colony Optimization 138
7.4.1 Ant System 138
7.4.2 Elitist Ant System 140
7.4.3 Rank-Based Ant System 140
7.4.4 Max-Min Ant System 140
7.5 Chapter Summary 141

CHAPTER 8 ANT COLONY IMPLEMENTATION 143
8.1 Heuristics Parameters of the MCM 143
8.2 Serial Implementation 145
8.3 Parallel Implementation of Ant Colony Algorithm 153
8.4 Chapter Summary 158

CHAPTER 9 CONCLUSIONS AND SUGGESTIONS FOR FUTURE WORK 159
9.1 Pattern Preservation Algorithm 160
9.2 Graph Encoder 160
9.3 Subexpression Tree Algorithm 161
9.4 Ant Colony Optimization Algorithm 162
9.5 Comparing the Proposed Algorithms 163

APPENDIX A GRAPH 165
A.1 Basic Terms 165
A.2 Paths, Circuits, and Cycles 167
A.3 Connected Graph 168
A.4 Eulerian Graph 168

APPENDIX B ADDER 171
B.1 Binary Adder 171
B.2 Carry-Save Adder 172
B.3 Redundant Adder 175

APPENDIX C ABSTRACT ALGEBRA 177
C.1 Congruence 177
C.2 Groups 178
C.3 Ring 179
C.4 Ring Homomorphism 180

References 181

LIST OF FIGURES

1.1 The binary multiplication of w_0 with x_n. w_0 wordlength equals $L = 3$ bits and x_n wordlength equals $L = 4$ bits. The symbol $\oplus$ is the logical exclusive OR operation. 4

1.2 Using a 4×3 bit multiplier to implement the multiplication $w_0 x_n$. 5

1.3 A schematic diagram of the full adder and its truth table. 5

1.4 The multiplication of $w_0 = 101$ with x_n bits. It is assumed w_0 is with three bit wordlength and x_n is with four bit. 6

1.5 Using a 4×3 bit multipliers to implement the multiplication $5x_n$. 7

1.6 Multiplierless realization of $5x_n$. 7

1.7 A 4-bit signal x_n passes through the lower nibble of 8-bit data lines $d_0 - d_7$. 8

1.8 Redirecting the signal x_n to pass through the lines $d_2 - d_5$ which is equivalent to shifting it to the left by two positions (multiplying by 4). 8

1.9 Using a dedicated hardware to implement the multiplierless realization of $5x_n$. 8

1.10 The canonical structure of the digital FIR filter. 9

1.11 The transposed structure of the FIR filter. 10

1.12 An example of symmetric FIR filter. 11

1.13 The transposed structure of the FIR filter shown in Figure 1.12. 11

1.14 Sharing the multiplier between the same value coefficients. 11

1.15 Finding the power of two common factors between the coefficients increases the sharing between them. The symbol $\ll i$ means this path shifts the signal to the left by i positions. 11

1.16 An example of FIR filter with coefficient set $\{5, 37, 47\}$. 13

1.17 The transposed structure of the filter $\{5, 37, 47\}$. 13

1.18 The multiplier block of the filter in Figure 1.17. 13

1.19 A possible multiplierless realization of the coefficient 37. 14

1.20 A possible multiplierless realization of the coefficient 47. 14

1.21 A possible multiplierless realization for the coefficients $5, 37$, and 47. 14

1.22 Sharing the computation between the coefficients 5 and 37. 15

1.23 Sharing the computation between the coefficients $\{5, 37, 47\}$ after removing the multiplication operation. 16

1.24 Another possible subexpression sharing between the coefficients $\{5, 37, 47\}$. 16

1.25 Synthesizing the filter $y_n = 11x_n + 7x_{n-1}$ using CSE and coefficient representations: (a) $7 = 100\bar{1}$ and $11 = 110\bar{1}$. (b) $7 = 100\bar{1}$ and $11 = 10\bar{1}0\bar{1}$. 18

1.26 (a) The synthesis of 103 from its CSD representation $10\bar{1}0100\bar{1}$ costs LO = 3 adders with $\delta = 2$. (b) The synthesis of 281 from its CSD representation $10010\bar{1}001$ costs LO = 3 adders with $\delta = 2$. (c) The synthesis of 103 and 281 commonly by sharing the subexpression $10\bar{1}001$ between them reduces the total cost to LO = 4 adders and increases the logic depth to LD = 3 adders. 19

1.27 An example of the optimal solution for the coefficient set $\{3, 21, 53\}$. Logic depth constraint is relaxed to LD = 3. 20

1.28 An example of the optimal solution for the coefficient set $\{3, 21, 53\}$. Logic depth constraint is tightened to LD = 2. 20

2.1 Directed graph showing the BHA. 28

2.2 Using the GD method to synthesize the coefficient set $\{-42, -40, 44\}$. (a) The canonical structure of the filter. (b) Making the coefficients positive odds. 30

2.3 (a) The transposed structure of the filter in Figure 2.2. (b) Synthesis the filter using the GD method. 31

2.4 Synthesis of the multiplier block (MB) of the FIR filter with coefficient set $\{5, 37, 47\}$. (a) The transposed structure of the filter. (b) The coefficient 5 is synthesized using cost 1 graph topology. (c) The coefficient 37 is found from 5 using one of cost 2 topologies. (d) A cost 3 graph topology is found such that the coefficient 47 is synthesized from 5 and 37. 32

2.5 Examples of different graph topologies. 33

2.6 Synthesis the coefficients $5, 21, 23, 205, 843, 3395$ using RAG-N algorithm results in LO = 6 and LD = 6. 35

2.7 Synthesis the coefficients $5, 21, 23, 205, 843, 3395$ using the CSE method of reference [Yao et al., 2004] results in LO = 10 and LD = 3. 35

2.8 Classification of the CSE Methods. 36

2.9 The three possible patterns that can be found in Hartley's table. 37

2.10 The result of using the HCSE to synthesize the coefficients $\{5, 11, 21\}$. 40

2.11 Finding the vertical common subexpressions (VCSs) among the CSD representations of the coefficients $\{5, 11, 21\}$. 43

2.12 Finding oblique common subexpressions (OCSs) in the CSD representations of the coefficients $\{5, 11, 21\}$. 44

2.13 An example of the MCSE. 45

2.14 An example of a branch and bound tree. 47

2.15 Filter coefficients $\{21, 11, 5\}$ are synthesized using the method proposed by Ho et al. [2008]. 47
2.16 Yao algorithm realization for the filter $\{105, 411, 815, 831\}$ with filter adder step constraint of 3 adders. 49
2.17 The optimal realization for the filter $\{105, 411, 815, 831\}$ with filter adder step constraint of 3 adders. 49

3.1 Venn diagram illustrating the relationship between CSD, MHW, ZDS, and BSD representations. 53
3.2 Horizontal common subexpressions among CSD representations of the coefficients, $\{-149, -135, -41, 53, 11\}$. 54
3.3 Finding the HCSs in one combination of the ZDS representations of the coefficients $\{-149, -135, -41, 53, 11\}$. 55
3.4 Solving the overlapping problem for the far/near end pattern. 56
3.5 An example illustrates the mechanism of the pattern preservation algorithm. 58
3.6 An example illustrates the mechanism of the two pass algorithm (TPA). 58
3.7 A comparison between the number of adders that obtained from using the PPA, DBAG [Gustafsson, 2007a], and Yao [Yao et al., 2004]. 61
3.8 A comparison between LD's that obtained from using DBAG [Gustafsson, 2007a], and PPA. 62
3.9 The run time comparison between the algorithms PPA, DBAG [Gustafsson, 2007a], and Yao [Yao et al., 2004]. 62

4.1 Classification of positional number systems [Parhami, 1990]. 67
4.2 The mapping $||.||$ is a homomorphism from $\mathcal{P}[r]$ to $\mathbb{Z}$. 71

5.1 The congruent relation, $\equiv_{|r|}$, partitions the digit set $\mathbb{D}$ into $|r|$ disjoint sets, where $\mathbb{D}^c$ is non redundant complete residue modulo $|r|$, i.e. $|\mathbb{D}^c| = |r|$. 76
5.2 Visualization of the first depth division of 281 by $r = 2$ with $\mathbb{D} = \{\overline{1}, 0, 1\}$. 77
5.3 An example of the representation tree for the value $v = 281$ using BSD with wordlength $L = 9$. Edge values are the digits of the representations. 79
5.4 An example of the encoding tree for the value $v = 25$ using number system with $r = 3$ and $\mathbb{D} = \{\overline{1}, 0, 1, 2\}$. 80
5.5 An example of the encoding graph for the value $v = 25$ using BSD. 82
5.6 An example of the encoding graph for the value $v = 25$ using the number system with $r = 3$ and $\mathbb{D} = \{\overline{1}, 0, 1, 2\}$. 83
5.7 An example of the encoding tree for the value $v = 25$ using BSD. $L_{\min}$ is the minimum wordlength to reach a zero node. 84
5.8 An example of the encoding graph for the value $v = 25$ using BSD. $L_{\min}$ is the minimum wordlength to traverse a self loop. 85
5.9 BSD state transitions in the recursive region. 86

5.10 A comparison between theoretical and measured average BSD tree size for the number range $[-256, 256]$. 88

5.11 Average tree size, T, for the range $[-256, 256]$. Plots (I) and (II) are respectively the theoretical and measured average tree size which obtained from using number system with $r = 3$, $\mathbb{D} = \{\overline{1}, 0, 1, 2\}$. Plots (III) and (IV) are respectively the theoretical and measured tree size obtained from using number system with $r = 3$, $\mathbb{D} = \{\overline{1}, 0, 1, 2, 3\}$. Plots (V) and (VI) are respectively the theoretical and measured tree size obtained from using number system with $r = 3$, $\mathbb{D} = \{\overline{2}, \overline{1}, 0, 1, 2\}$. 88

5.12 Average tree size, T, for the range $[-256, 256]$. Plots (I) and (II) are respectively the theoretical and measured average tree size which obtained from using number system with $r = 4$, $\mathbb{D} = \{\overline{1}, 0, 1, 2, 3\}$. Plots (III) and (IV) are respectively the theoretical and measured tree size obtained from using number system with $r = 4$, $\mathbb{D} = \{\overline{1}, 0, 1, 2, 3, 4\}$. Plots (V) and (VI) are respectively the theoretical and measured tree size obtained from using number system with $r = 4$, $\mathbb{D} = \{\overline{3}, \overline{2}, \overline{1}, 0, 1, 2, 3\}$. 89

5.13 Average tree size, T, for the range $[-256, 256]$. Plots (I) and (II) are respectively the theoretical and measured average tree size which obtained from using number system with $r = 5$, $\mathbb{D} = \{\overline{1}, 0, 1, 2, 3, 4\}$. Plots (III) and (IV) are respectively the theoretical and measured tree size obtained from using number system with $r = 5$, $\mathbb{D} = \{\overline{1}, 0, 1, 2, 3, 4, 5\}$. Plots (V) and (VI) are respectively the theoretical and measured tree size obtained from using number system with $r = 5$, $\mathbb{D} = \{\overline{4}, \overline{3}, \overline{2}, \overline{1}, 0, 1, 2, 3, 4\}$. 89

5.14 Average tree size, T, for the range $[-256, 256]$. Plots (I) and (II) are respectively the theoretical and measured average tree size which obtained from using number system with $r = 3$, $\mathbb{D} = \{0, 1, 2, 3\}$. Plots (III) and (IV) are respectively the theoretical and measured tree size obtained from using number system with $r = 3$, $\mathbb{D} = \{\overline{1}, 0, 1, 2, 3\}$. Plots (V) and (VI) are respectively the theoretical and measured tree size obtained from using number system with $r = 3$, $\mathbb{D} = \{\overline{2}, \overline{1}, 0, 1, 2\}$. 90

5.15 Average tree size, T, for the range $[-256, 256]$. Plots (I) and (II) are respectively the theoretical and measured average tree size which obtained from using number system with $r = 4$, $\mathbb{D} = \{0, 1, 2, 3, 4\}$. Plots (III) and (IV) are respectively the theoretical and measured tree size obtained from using number system with $r = 4$, $\mathbb{D} = \{\overline{1}, 0, 1, 2, 3, 4\}$. Plots (V) and (VI) are respectively the theoretical and measured tree size obtained from using number system with $r = 4$, $\mathbb{D} = \{\overline{3}, \overline{2}, \overline{1}, 0, 1, 2, 3\}$. 90

5.16 Average tree size, T, for the range $[-256, 256]$. Plots (I) and (II) are respectively the theoretical and measured average tree size which obtained from using number system with $r = 5$, $\mathbb{D} = \{0, 1, 2, 3, 4, 5\}$. Plots (III) and (IV) are respectively the theoretical and measured tree size obtained from using number system with $r = 5$, $\mathbb{D} = \{\overline{1}, 0, 1, 2, 3, 4, 5\}$. Plots (V) and (VI) are respectively the theoretical and measured tree size obtained from using number system with $r = 5$, $\mathbb{D} = \{\overline{4}, \overline{3}, \overline{2}, \overline{1}, 0, 1, 2, 3, 4\}$. 91

5.17 Elapsed time in seconds to generate the BSD representations for bunches of numbers with $L_{\min} = 6, 7, 8,$ and 9 respectively. 92

5.18 Elapsed time in seconds to generate the BSD representations for bunches of numbers with $L_{\min} = 10, 11, 12,$ and 13 respectively. 92

5.19 Number of BSD representations Γ versus wordlength L for the filters L_1 and S_2. 93

5.20 Number of BSD representations Γ versus wordlength L for the filters: L_3 and S_1. 93

5.21 The encoding tree for the value $v = 25$ using BSD. The branches after zero nodes are pruned. 94

5.22 The average number of representations $\hat{\Gamma}$ versus digit set cardinality $|\mathbb{D}|$ for the values in the range $[-256, 256]$ using number systems with $r = 2$, $L = 8$, and alphabets $\{\overline{1}, 0, 1\}$, $\{\overline{1}, 0, 1, 2\}$, $\{\overline{1}, 0, 1, 2, 3\}$, $\{\overline{1}, 0, 1, 2, 3, 4\}$, and $\{\overline{1}, 0, 1, 2, 3, 4, 5\}$. 94

6.1 A branch of the representation tree of 281 results in a non-exhaustive search. 102

6.2 Realization of the filter F_1 in [Farahani et al., 2010] with coefficient set $W = \{25, 103, 281, 655\}$ using the STA. 105

6.3 The synthesis of the filter F_1 with coefficients $\{25, 103, 281, 655\}$ using (a) STA^{da} algorithm with LD = 3 results LO = 6. (b) STA^{da} algorithm with LD = 2 results LO = 7. (c) Yao [Yao et al., 2004] with LD = 3 results LO = 7. (d) DBAG [Gustafsson, 2007a] results LO = 6 and LD = 4. 111

6.4 A comparison between the number of adders that obtained from using the algorithms DBAG [Gustafsson, 2007a], PPA, Yao [Yao et al., 2004], and STA used to synthesize random MCMs of 5 coefficients. 113

6.5 A comparison between the LD's that obtained from using the algorithms DBAG [Gustafsson, 2007a], PPA, Yao [Yao et al., 2004], and STA used to synthesize random MCMs of 5 coefficients. 113

6.6 The run time comparison between the algorithms DBAG [Gustafsson, 2007a], PPA, Yao [Yao et al., 2004], and STA used to synthesize random MCMs of 5 coefficients. 113

6.7 A comparison between the number of adders that obtained from using the algorithms DBAG [Gustafsson, 2007a], Yao [Yao et al., 2004], and STA used to synthesize random MCMs of 10 coefficients. 114

6.8 A comparison between the LD's that obtained from using the algorithms DBAG [Gustafsson, 2007a], and STA used to synthesize random MCMs of 10 coefficients. Yao [Yao et al., 2004] and STA have the same LD. 114

6.9 The run time comparison between the algorithms DBAG [Gustafsson, 2007a], Yao [Yao et al., 2004], and STA used to synthesize random MCMs of 10 coefficients. 114

6.10 Realizing the filter $\{49, 188, 248, 251, 245, 37, 129, 206, 207, 83\}$ using Hcub method. 115

6.11 Realizing the filter $\{49, 188, 248, 251, 245, 37, 129, 206, 207, 83\}$ using STA. 116

6.12 Realizing the filter $\{4105, 831, 621, 815\}$ using Hcub method. 117

6.13 Realizing the filter $\{105, 831, 621, 815\}$ using STA. 117

6.14 Realizing the filter $\{105, 831, 411, 815\}$ using Hcub method. 118

6.15 Realizing the filter $\{105, 831, 411, 815\}$ using STA. 118

7.1 Streets that must be traversed by a mailman. 122

7.2 An example of making a non-unicursal graph G to unicursal one $\hat{G}$. (a) A graph G with the in-degree not equal the out-degree for the vertices $17, 31$, and 33. (b) A graph $\hat{G}$ with in-degree equals out-degree for all the vertices. 123

7.3 Three possible realizations for the coefficients $\{5, 37, 47\}$. 125

7.4 (a) Combining the realizations in Figure 7.3 results in the decomposition graph. (b) Transforming the decomposition graph results in the demand graph. 125

7.5 Transforming the shift information to a subexpression information. (a) Representing the $\mathcal{A}-$operation where the arcs carry shift information. (b) Combining the addition vertex with the output vertex. (c) A new representation for the two part decomposition. 126

7.6 Adding deadheading arcs to the demand graph shown in Figure 7.4 (a). 127

7.7 The demand graph of the coefficients $\{5, 37, 47\}$. 128

7.8 Adding dead heading arcs to the demand graph of the coefficients $\{5, 37, 47\}$. 129

7.9 (a) A possible tour on the augmented demand graph results in the realization shown in Figure 7.3(a). (b) The corresponding winter gritting route. A solid arrow with plus sign indicates the salting action and the label preceding each arrow is the next destination after this road. A dashed arrow means traversing a deadheading road. The roundabouts are labeled by the road number and the road is labeled by the path between two roundabouts. 132

7.10 (a) A possible tour on the augmented demand graph results in the realization shown in Figure 7.3(b). (b) The corresponding winter gritting route. 133

7.11 (a) A possible tour on the augmented demand graph results in the realization shown in Figure 7.3(c). (b) The corresponding winter gritting route. 134

7.12 Two possible tours on the demand graph for the coefficients $\{45, 195\}$. 136

7.13 (a) The tour (depot $\rightarrow$ 17 $\rightarrow$ 45 $\rightarrow$ 17 $\rightarrow$ depot $\rightarrow$ 31 $\rightarrow$ 45 $\rightarrow$ 195 $\rightarrow$ depot $\rightarrow$ 15 $\rightarrow$ 195 $\rightarrow$ depot) on the demand graph for the coefficients $\{45, 195\}$. (b) The corresponding winter gritting route. 137

8.1 An ant cross arc $(s_i, s_j)_h$ with a demand of s_h. 144

8.2 Percentage of ants found the best-so-far tour against the parameters α and β. A set of 100 MCM with $N = 5$ and wordlengths $8 \leq L \leq 12$ was used. The number of generations (iterations) is 10 and each generation used 4 ants. 148

8.3 A comparison between the number of adders obtained from using the algorithms PPA, STA, and the serial implementation of MMAS used to synthesize random MCMs of 5 coefficients. 151

8.4 The run time comparison between the algorithms PPA, STA, and the serial implmentation of MMAS used to synthesize random MCMs of 5 coefficients. 151

8.5 A comparison between the number of adders obtained from using the algorithms DBAG [Gustafsson, 2007a], PPA, Yao [Yao et al., 2004], STA, and the serial implmentation of MMAS used to synthesize random MCMs of 5 coefficients. 152

8.6 A comparison between the run time of the algorithms DBAG [Gustafsson, 2007a], PPA, Yao [Yao et al., 2004], STA, and the serial implmentation of MMAS used to synthesize random MCMs of 5 coefficients. 152

8.7 A comparison between LD for the methods MMAS, PPA, and DBAG [Gustafsson, 2007a]. 152

8.8 Run time comparison of the algorithm MMAS between its parallel implementation MMASp and the serial implementation MMASs. 157

8.9 Run time comparison between the algorithms PPA, STA, and the parallel implmentation of MMAS (MMAS^p) when they used to synthesize random MCMs of 5 coefficients. 157

9.1 Synthesis of the filter F_1 in [Farahani et al., 2010] using the customized graph encoder results in a realization with LO = 6. 161

A.1 An example of undirected graph G. 166

A.2 An example of directed graph G. 166

A.3 An example of multidigraph G. 166

A.4 A connected graph example. 167

A.5 An example of disconnected graph. 168

A.6 An example of unicursal graph. 169
A.7 An example of Eulerian graph. 169

B.1 4-bit carry propagate adder. 172
B.2 Addition of two 4-bit operands. 172
B.3 Addition of four binary operands using carry save addition. 173
B.4 Four operand carry-save adder (CSA). Note the final stage requires a carry propagate adder (CPA). 174
B.5 Adding two redundant 4-digit numbers with $r = 4$, $\mathbb{D} = \{\overline{2}, \overline{1}, 0, 1, 2, 3\}$, and $t_i = \{\overline{1}, 0, 1\}$. 175
B.6 An example of a 4-digit redundant adder. **TW** denotes the stage of decomposition of the position sum $\mathbf{P_i} = \mathbf{w_i} + \mathbf{rt_{i+1}}$. 176

LIST OF TABLES

2.1 Enumerate the BSD representations with wordlength equals $L = 5$. 39

2.2 BSD representations of the coefficients $\{5, 11, 21\}$. 40

2.3 Finding the HCSs among the representations of the coefficients $\{5, 11, 21\}$. 40

2.4 CSD representations of the coefficients $\{200, 206, 655, 281\}$. 41

2.5 The number of bit-wise matches among all pairs of coefficients. 41

2.6 The number of nonzero bits for all pairs of constants. 42

2.7 A best match is found in the pair w_2 and w_3 with a common pattern $000010\bar{1}001$. 42

2.8 The coefficients after eliminating the common pattern $000010\bar{1}001$. 42

3.1 Dominance relations for the representations of the value 115. 52

3.2 The ZDS of the coefficients $\{-149, -135, -41, 53, 11\}$. 54

3.3 A comparison between the PPA performance using the zero-dominant set (ZDS) and the minimum Hamming weight (MHW) representations. 59

3.4 A comparison between the pattern preservation algorithm (PPA), the two pass algorithm (TPA), the single pass algorithm (SPA), and the CSD algorithm (CSA). 59

3.5 A comparison between: PPA with ZDS, N-dimensional reduced adder graph (RAG-N) method, Hartley's method for CSD, and Hartley's method for BSD, with wordlength extended by one digit. 60

4.1 Examples of redundant number system classes. 68

6.1 Subexpressions comprising of two or more digits found in one of the 25 BSD representations, $1\bar{1}11\overline{11}$. s'_1 and s'_2 are the raw subexpressions. s_1 and s_2 are positive co-prime subexpressions. 100

6.2 Priority values for the subexpression tree algorithm. W is the set of coefficients and subexpressions (fundamentals) waiting for synthesis, T is the set of synthesized fundamentals, and S_s is the set of shared (common) subexpressions. A low priority number is more desirable. 104

6.3 Samples of the decomposition complexity values obtained from the representation tree of 655. 105

6.4 A comparison between the STA^{da} algorithm and the algorithms MITM [Farahani et al., 2010], DBAG [Gustafsson, 2007a], and Yao [Yao et al., 2004] to realize the filters F_1 and F_2 in Farahani et al. [2010]. 110

6.5 A comparison between the STA^{da} algorithm and the algorithms Tsao [Tsao and Choi, 2010], BCSE [Smitha and Vinod, 2007], DBAG [Gustafsson, 2007a], Yao [Yao et al., 2004] to realize the filters S_2 and L_1. 110

6.6 A comparison between the algorithms RAG-n [Dempster and Macleod, 1995], C1 [Dempster et al., 2002], Yao [Yao et al., 2004], MILP [Ho et al., 2008], DBAG [Gustafsson, 2007a], and STA^{da} to realize the filter D in [Dempster and Macleod, 1995] with $N = 25$. 110

8.1 Priority values of the demand s_h associated with arc $(s_i, s_j)_h$. W is the set of subexpressions waiting for synthesis, T is the set of synthesized subexpressions, $a > b > 1$ are constants. 145

8.2 Synthesis the filter F_1 [Farahani et al., 2010] using MMAS ant system. 148

8.3 Synthesis the filter $\{21, 53, 171\}$ using MMAS ant system. 149

8.4 A comparison between the MMAS ant system, and the algorithms DBAG Gustafsson [2007a], Yao Yao et al. [2004], and STA^{da} used to realize the filters F_1 Farahani et al. [2010], F_2 Farahani et al. [2010], and S_2 [Samueli, 1989] . 149

8.5 Synthesis the filters S_2 , L_1, and D using the parallel MMAS algorithm. t_g and t_s are the generating and the search time respectively. 156

9.1 A comparison between the PPA, STA, and MMAS algorithms. 164

C.1 The addition table in $\mathbb{Z}_3$. 180

C.2 The multiplication table in $\mathbb{Z}_3$. 180

Chapter 1

INTRODUCTION

There is market demand to manufacture efficient digital signal processing (DSP) and digital image processing (DIP) based devices. An efficient DSP/DIP system should be of minimum cost, minimum power consumption, and maximum processing speed. Examples of DSP/DIP systems are wireless communication, machine vision, and telescopes. The reasons for seeking efficient DSP/DIP implementation are shown by considering three different applications.

In wireless communication systems, a new generation appears approximately every 10 years. The 1G system was introduced in 1981 with frequency bands up to 30 kHz, 2G in 1992 and up to 200 kHz, 3G in 2001 and up to 5 MHz, and 4G in 2011 and up to 40 MHz [Wang et al., 2014]. It would not be unreasonable to expect 5G to appear sometime in 2020 with either higher frequency band, lower battery consumption, better coverage, or less price. This has made the problem of finding a low cost, low-power, and high throughput digital signal processing (DSP) system an active research area [Wang and Roy, 2005] [Mathew et al., 2008] [Vinod and Lia, 2005] [Kamp and Bainbridge-Smith, 2007] [Aksoy et al., 2010]. The most computationally intensive part of the wireless communication systems is the finite impulse response (FIR) digital filter. For example, the channelizer in the receiver extracts multiple narrowband channels from a wideband signal using a bank of bandpass FIR filters. The bandpass filter could be of order 200 to 1200 and have a sharp transition [Mathew et al., 2008]. In addition, filter banks should work at a very high speed with low power consumption in order to be useful in battery operation.

Micro aerial vehicles (MAVs) are a developing research area during the last decade. They are bug-size flyers that could be used, for example, in spying, mine detection, environmental monitoring, and search and rescue missions in collapsed buildings (such as searching for survivors in a damaged building after an earthquake) [ScienceDaily, 2009]. They should be made efficient enough for long missions. Building a low-power vision systems helps flying robots to navigate and identify objects [IEEESpectrum, 2013]. Digital image processing (DIP) algorithms like edge detection, image filtering, and feature extraction should run on battery-powered platforms. Since most of these algorithms are two dimensional FIR filters, these filters should work at a high speed with low power consumption.

The Square Kilometer Array (SKA) is a new generation radio telescope that has a

discovery potential 10,000 times greater than the best present-day instruments [SKA, 2013]. It gives astronomers remarkable insights into the formation of the early Universe, including the emergence of the first stars, galaxies, and other structures. The SKA is a signal processing based instrument, with the receiver being the most extreme system for processing requirements. Polyphase digital filter banks and Fast Fourier Transforms (FFT) are being actively developed as the core elements of beamformers, spectrometers, and cross-correlators. For example, the polyphase digital filter bank in the baseband receiver is a 16384-tap FIR filter and a 2048-point Discrete Fourier Transform (DFT) [SKA, 2013]. The FIR filter part is arranged as 2048 distinct 8-tap subfilters. Power demand of the SKA may reach up to 100 MW which is divided equally between the array and the computer center which includes the high performance DSP [Hall, 2011]. Therefore, using a technique to design a low power DSP for the SKA will save a remarkable amount of power.

Two common themes can be extracted in the DSP/DIP systems mentioned above; first they require an efficient DSP/DIP implementation, and second the core computational operation is the multiple constant multiplication (MCM) such as the FIR and IIR filters. The MCM implements the dot-product operation of a signal vector with a vector of fixed coefficients. Since the MCM is a core operation of most DSP/DIP systems, implementing an efficient MCM operation is critical to building efficient DSP/DIP systems. Three objectives for the design engineer when looking for efficient MCM operation are:

1. Minimize the use of resources.
2. Minimize the critical path length.
3. Minimize power consumption.

Minimizing the resources used reduces the static power consumption which is the product of the CMOS device leakage current and the supply voltage [Sarwar, 1997]. On the other hand, the existence of a long critical path in the MCM circuit increases the chance of glitch occurring which increases the dynamic power consumption. Dynamic power consumption of the CMOS device is due to the switching of transistors from one logic state to another and to the charging of external load capacitance [Sarwar, 1997]. Therefore, minimizing critical path(s) length minimizes dynamic power consumption and maximizes the processing speed simultaneously. Where, the critical path in the MCM is the longest path that the signal passes through from the input to the output. Using a digital multiplier to implement the MCM operation contradict with achieving these objectives. This is because the digital multiplier consumes a large area, high power, and has a long delay (critical path) [Aksoy et al., 2011]. In other words, the digital multiplier is a bottleneck element in DSP/DIP systems. Therefore, it is important to remove the multiplication operation from the MCM operation to increase its efficiency when implemented on ASICs and FPGAs [Farahani et al., 2010].

Removing the explicit multiplication operation from the implementation of MCM makes the latter a multiplieless. In this case, the multiplication operation is substituted

with simpler ones of add/subtract and shift. It is required to develop an optimization procedure that works on the multiplierless MCM to maximize the achieving of the above objectives. There are several optimization procedures in the literature that tackle the MCM problem. Among them the common subexpression elimination (CSE) technique is found to be the most reliable one because it can compromise between the three objectives better than other techniques. However, using this technique requires searching a large space of solutions to find the optimum one. Therefore, the aims of this research are:

1. Review and classify CSE methods.
2. Introduce new CSE methods to reduce the complexity of MCM.
3. Introduce use different search spaces for CSE methods.

This chapter is structured as follows: Section 1.1 illustrates removal of the multiplication operation from a single multiplier. The MCM operation is studied in Section 1.2. Making the MCM operation multiplierless is described in Section 1.3. The concept of CSE is described in Section 1.4. Radix number system is introduced in Section 1.5. Finding the subexpressions from number representations is described in Section 1.6. Section 1.7 is dedicated to explain the coefficient adder step concept. A summary of the chapter is given in Section 1.9.

1.1 MULTIPLICATION WITHOUT MULTIPLICATION

In this section we consider a simple case of one multiplier to show the disadvantage of using digital multiplier in DSP/DIP systems. It also illustrates the advantage of removing the explicit multiplication operation even from a single multiplier. Assume an MCM with N coefficients, $w_0, \cdots, w_{N-1}$, and a signal variable x_n multiplies these coefficients, where the subscript n is the discrete time variable. Consider the multiplication of the coefficient w_0 by x_n given by $w_0 x_n$. If the signal wordlength equals $L = 4$ bit (assume binary representation) and the coefficient wordlength equals 3 bit, their binary multiplication is shown in Figure 1.1. The symbol $x_{<n,j>}$ represents the jth bit of the binary representation of x_n, similarly for $w_{<0,j>}$. Each bit multiplication of the form $w_{<0,i>} x_{<n,j>}$ implemented using logical AND operation.

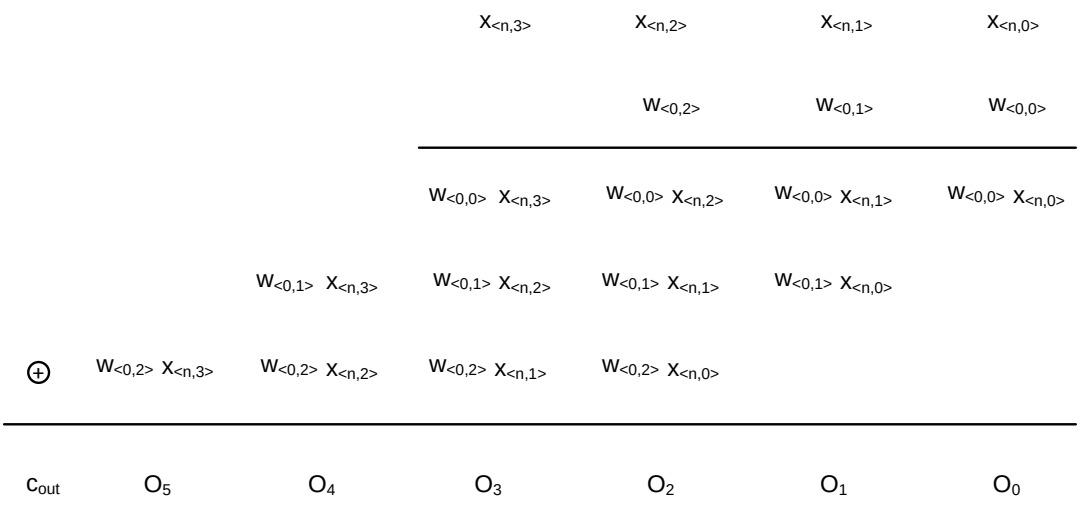


Figure 1.1: The binary multiplication of w_0 with x_n. w_0 wordlength equals $L = 3$ bits and x_n wordlength equals $L = 4$ bits. The symbol $\oplus$ is the logical exclusive OR operation.

The hardware realization of the multiplication shown in Figure 1.1 requires using a 4×3 digital multiplier as shown in Figure 1.2. The digital multiplier in Figure 1.2 consists of an array of nine full adders (FA) arranged to implement three level addition. The block diagram for the full adder is shown in Figure 1.3. It has three input $\mathrm{IN}_1, \mathrm{IN}_2$, and CARRY IN, and has two outputs, the Sum and CARRY OUT. The FA sums the inputs and results outputs according to the truth table shown in Figure 1.3. In Figure 1.2, the carry propagates diagonally and horizontally resulting in a critical path of length five FA as shown in red. For example, the carry $c_{i,j}$ propagates to position i at level j. We conclude that using an $L \times L$ general purpose digital multiplier costs $f(L) = (L-1) \times L$ FA with a critical path of length $2L-1$ FA, while an adder of wordlength L costs L FA. The function $f(L)$ can be written as $f(L) = L^2 - L$. The big-oh notation of $f(L)$ is given by $f(L) = O(L^2)$ [Roberts and Tesman, 2009]. Therefore, an array multiplier with wordlength L occupies an area of $O(L^2)$, whereas that for the adder is $O(L)$ [Bull and Horrocks, 1991].

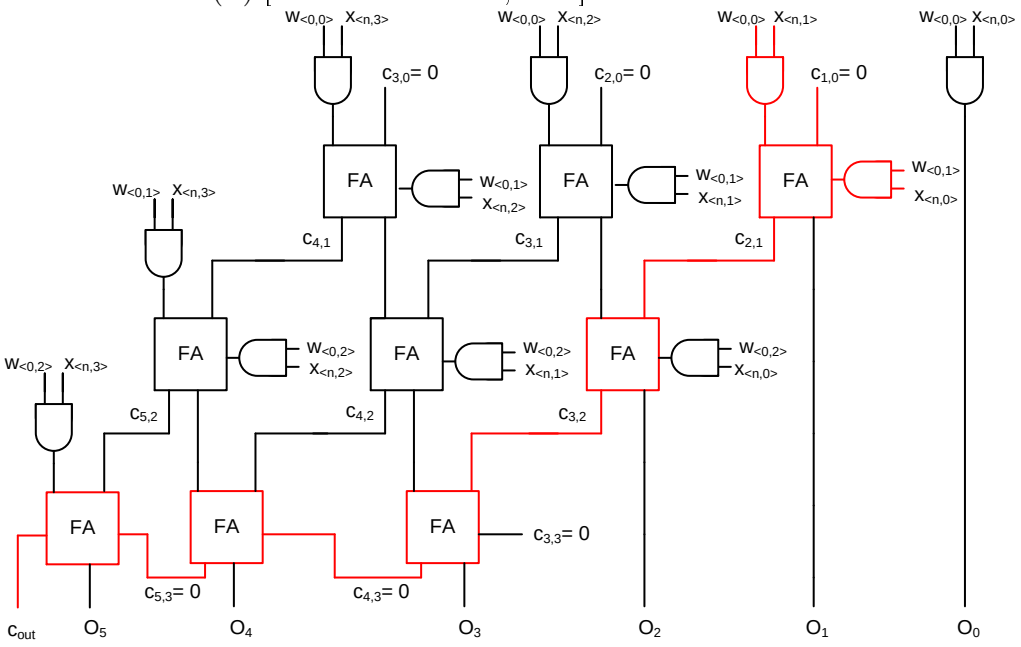
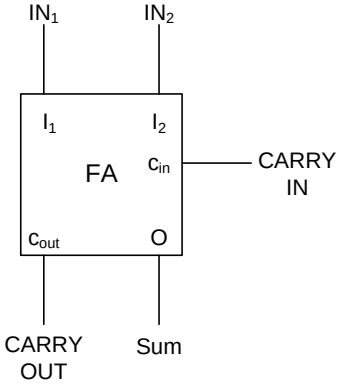


Figure 1.2: Using a 4×3 bit multiplier to implement the multiplication $w_0 x_n$.



I_1	I_2	c_{in}	O	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 1.3: A schematic diagram of the full adder and its truth table.

Assume the coefficient w_0 is of value equals to 5 or 101 in binary, the corresponding bits are $w_{<0,0>} = 1, w_{<0,1>} = 0$, and $w_{<0,2>} = 1$. The multiplication $5x_n$ is shown in Figure 1.4 and its hardware implementation using digital multiplier is shown in Figure 1.5. Alternatively, if the coefficient 5 is decomposed to $5 = 4 + 1$, it can be synthesized as shown in Figure 1.6 which is multiplication free. When the signal x_n passes through the left branch in Figure 1.6, it undergoes a shift to the left by two positions. This shift operation is indicated by the symbol $\ll 2$, where $\ll i$ means shifting the value to the left by i positions which is equivalent to multiplying it by the power of two 2^i. The shift operation can be implemented in hardware for free by redirecting the signal x_n on the data path. Figure 1.7 shows the case when the signal x_n passes through the data path $(d_0 - d_7)$ without undergoing any shift. While Figure 1.8 illustrates the case of shifting x_n to the left by two positions to produce $4x_n$ on the data path. The implementation of the addition $x_n + 4x_n$ is shown in Figure 1.9. The total number of FA in this realization is reduced to five and the critical path reduced to four. This reduction has a significant impact on the cost and power consumption.

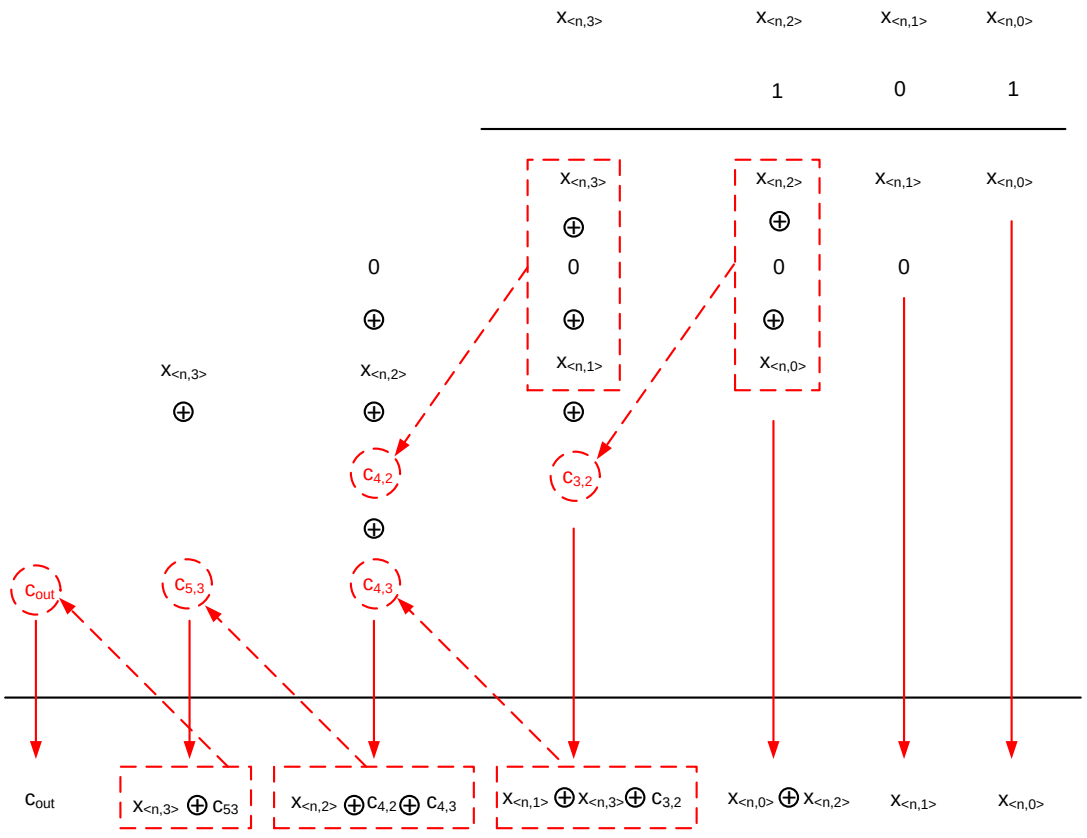


Figure 1.4: The multiplication of $w_0 = 101$ with x_n bits. It is assumed w_0 is with three bit wordlength and x_n is with four bit.

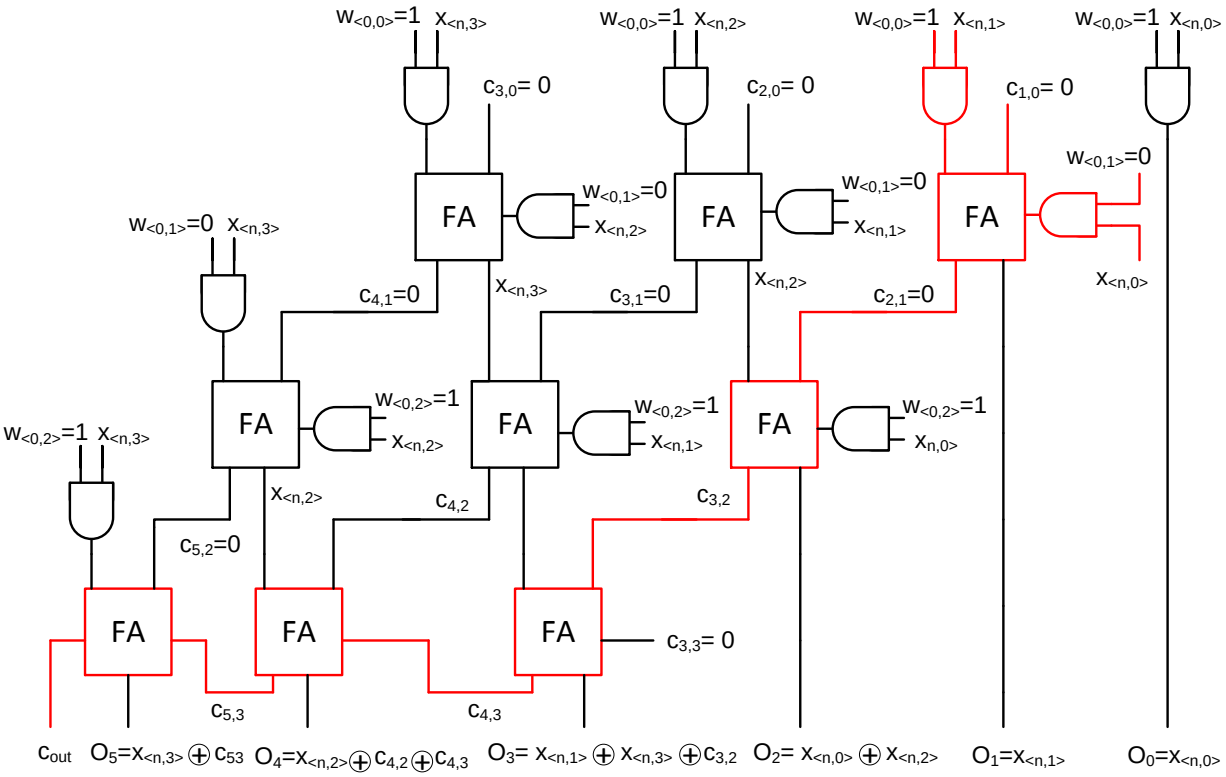


Figure 1.5: Using a 4×3 bit multipliers to implement the multiplication $5x_n$.

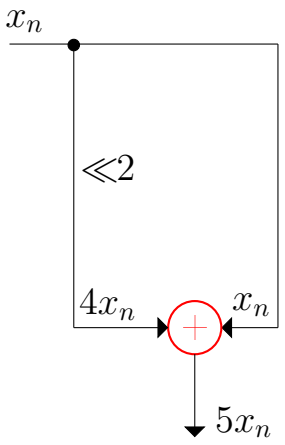


Figure 1.6: Multiplierless realization of $5x_n$.

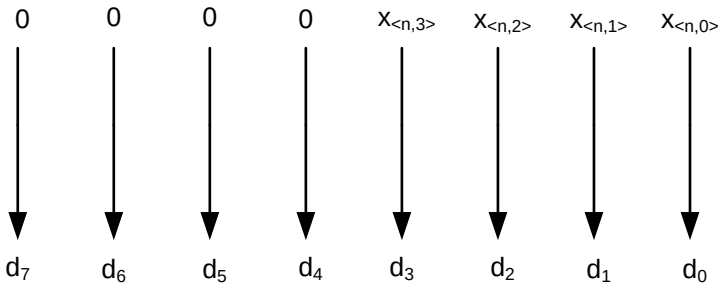


Figure 1.7: A 4-bit signal x_n passes through the lower nibble of 8-bit data lines $d_0 - d_7$.

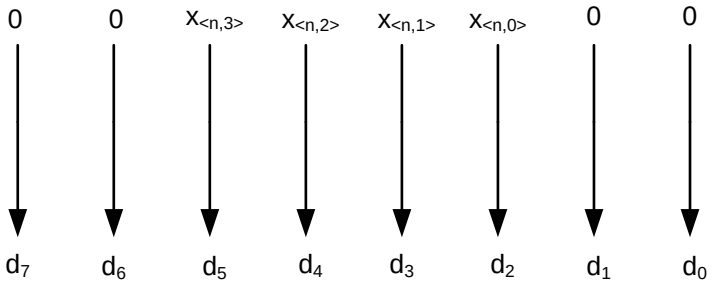


Figure 1.8: Redirecting the signal x_n to pass through the lines $d_2 - d_5$ which is equivalent to shifting it to the left by two positions (multiplying by 4).

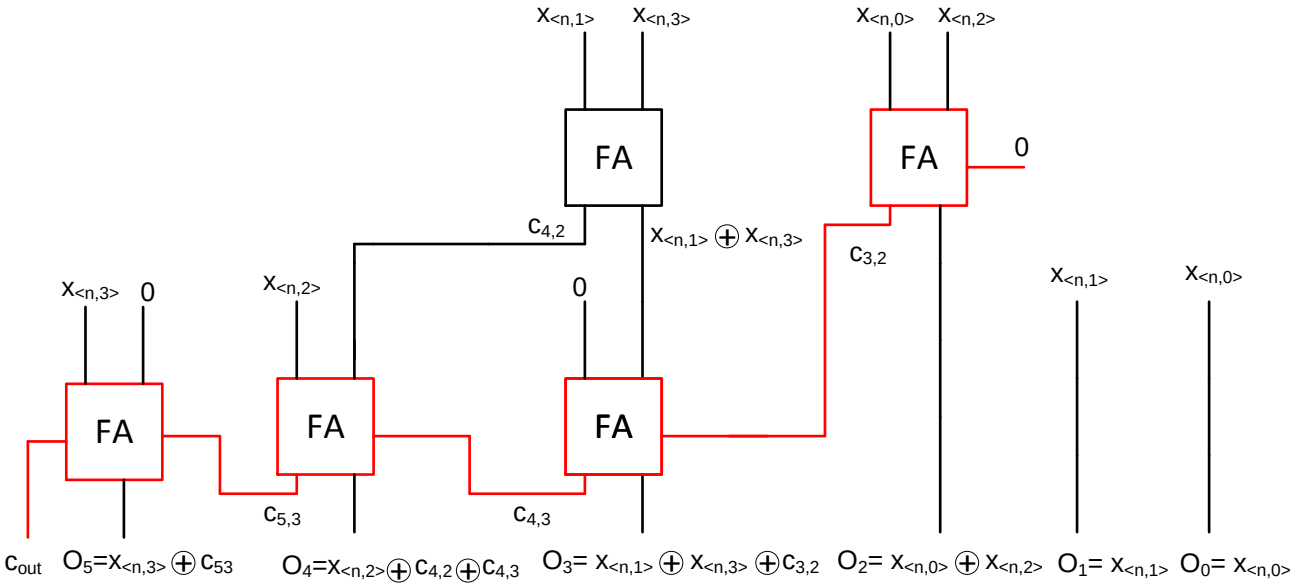


Figure 1.9: Using a dedicated hardware to implement the multiplierless realization of $5x_n$.

1.2 MULTIPLE CONSTANT MULTIPLICATION

The MCM is found in most computationally intensive DSP/DIP such as FIR filters, IIR filters, correlators, DSP transforms, edge detection, etc. [Aksoy et al., 2010]. Therefore implementing an efficient MCM operation is a major concern in the design of low cost, high-speed, and low-power DSP/DIP systems [Aksoy et al., 2011; Vinod and Lia, 2005; Mathew et al., 2008; Aksoy et al., 2012b]. To solve the multiplier problem, the MCM is designed to be a multiplication free (multiplierless) by substituting the explicit multiplication with operations of shift and add/subtract [Yao et al., 2004; Johansson et al., 2011] in a similar way that described in Section 1.1. The FIR filter is considered in this development as an important example of the MCM operation. Figure 1.10 shows the canonical structure of the FIR filter. The structure inside the dashed box contains the multipliers and delay elements. The adders out of the box are called the structure adders. The MCM can be written as:

$$y_n = \sum_{k=0}^{N-1} x_{n-k} w_k, \tag{1.1}$$

where w_k is a set of coefficients, N the filter's order, and x_n the input variable. Equation 1.1 accomplishes the convolution of the signal x_n with the coefficients w_n. Another mathematical form for Equation 1.1 is shown in $y_n = \mathbf{X_n^T W}$ which is the inner product between the coefficient vector, $\mathbf{W} = [w_0 w_1 \cdots w_{N-1}]^T$, and the signal vector $\mathbf{X_n} = [x_n x_{n-1} \cdots x_{n-(N-1)}]^T$, where T means vector transpose. The direct parallel implementation of Figure 1.10 requires using N digital multipliers and $N-1$ adders. Since the digital multiplier is more complex than the adder, filter's complexity is measured by its order N which is the number of multipliers required to realize the filter.

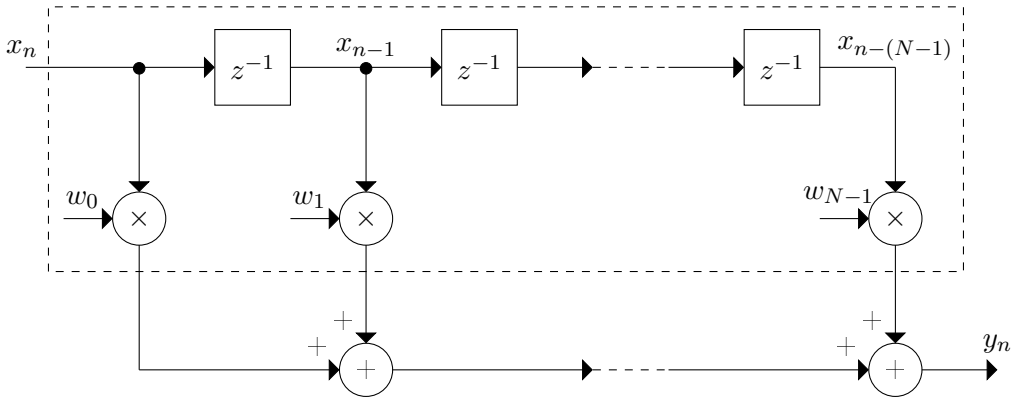


Figure 1.10: The canonical structure of the digital FIR filter.

To tackle the multiplier problem in the FIR filter, an alternate filter structure is used which is the transposed structure that is shown in Figure 1.11. It results from transposing the delay line in Figure 1.10 to be as shown in Figure 1.11. Using the transposed structure helps in reducing the hardware complexity of the filter without affecting its performance [Dempster and Macleod, 1995]. The block inside the dashed rectangle in Figure 1.11 is called the multiplier block (MB). The signal x_n is multiplied by the coefficients inside the MB and the partial products that come out of the block are accumulated. Therefore, the dot product operation of the original canonical structure is transformed to the MCM operation. Since the input signal now multiplies the coefficients simultaneously therefore it is easy to synthesize the coefficients by finding and sharing the common factors among them.

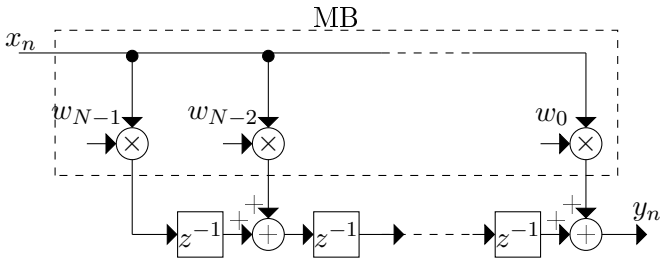


Figure 1.11: The transposed structure of the FIR filter.

An example of coefficient sharing is found in the symmetric FIR filter. The impulse response of this type of FIR filters is designed to be symmetric around the origin to obtain a linear phase operation. In this case, synthesis half the symmetric impulse response synthesizes all the coefficients. For example, consider an FIR filter with coefficient set $W = \{5, 10, 10, 5\}$. The canonical realization of the filter is shown in Figure 1.12 which requires using 4 digital multipliers. If the filter structure is transposed as shown in Figure 1.13, the coefficients $w_0 = 5$ and $w_3 = 5$ can share the same multiplier 5. Similarly the coefficients $w_1 = 10$ and $w_2 = 10$ share the multiplier 10. The new filter structure after sharing the coefficients is shown in Figure 1.14. In this case, two multipliers are saved. More saving is obtained from noticing that the coefficient 10 can be obtained from shifting 5 to the left by one position. This shift is represented arithmetically by $10 = 5 \ll 1$. The power of two relation between the coefficients 5 and 10 reduces the number of multipliers to only one as shown in Figure 1.15. The whole filter operation can be implemented now by just generating the signal $u_n = 5x_n$ as shown in Figure 1.15.

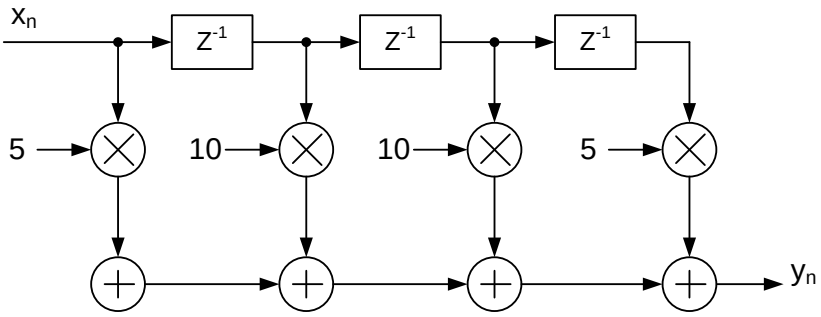


Figure 1.12: An example of symmetric FIR filter.

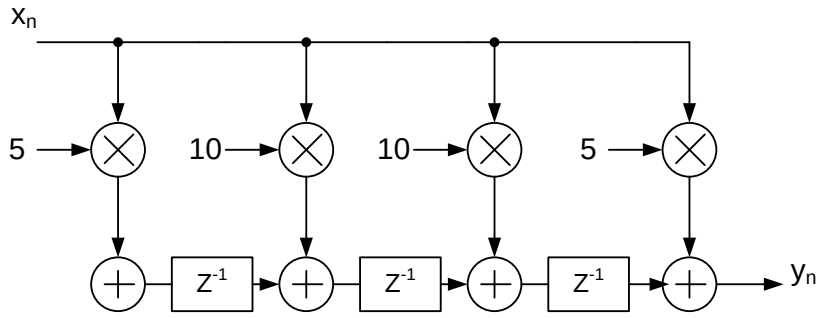


Figure 1.13: The transposed structure of the FIR filter shown in Figure 1.12.

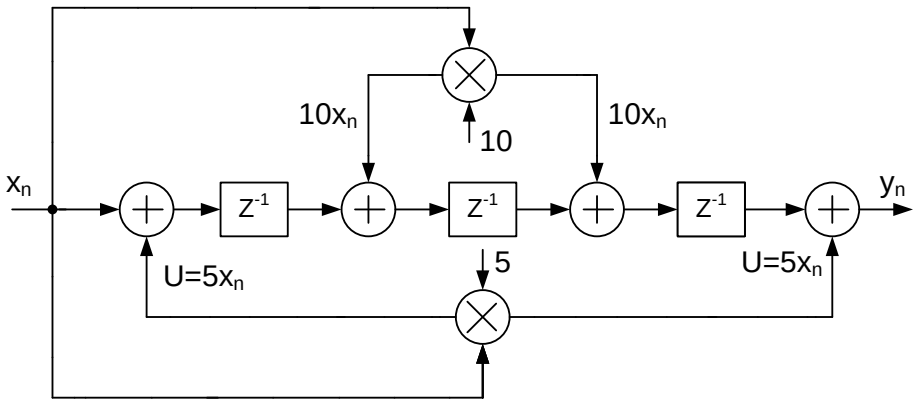


Figure 1.14: Sharing the multiplier between the same value coefficients.

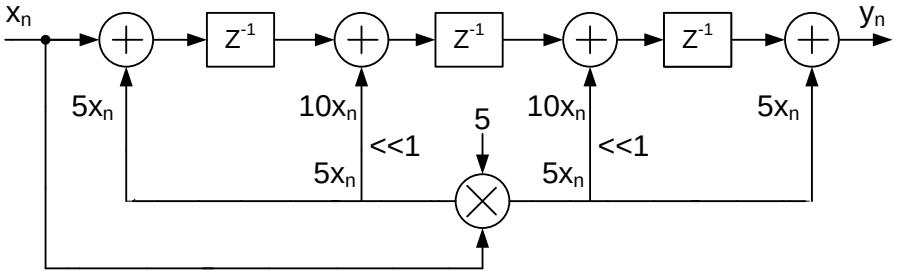


Figure 1.15: Finding the power of two common factors between the coefficients increases the sharing between them. The symbol $\ll i$ means this path shifts the signal to the left by i positions.

1.3 MULTIPLIERLESS MULTIPLE CONSTANT MULTIPLICATION

The approach described in Section 1.1 shows how to remove the explicit multiplication from a single coefficient. The same approach can be applied for multiple coefficients. Removing the multiplication operation from the MCM makes it multiplierless. To illustrate the procedure, consider the following FIR filter

$$y_n = 47x_n + 37x_{n-1} + 5x_{n-2}. \tag{1.2}$$

The canonical structure of the filter is shown in Figure 1.16 and its transposed form is shown in Figure 1.17. The MB in Figure 1.17 is enclosed by a box indicating that this is the most intensive computing part in the MCM and a minimizing procedure is required to reduce its complexity. For this reason the MB circuit is shown separately in Figure 1.18. The multiplierless realization of the coefficient 5 is shown in Figure 1.6. The coefficient 37 can be decomposed into a sum/difference of the power of two factors. A possible decomposition for 37 is $37 = 7 \times 4 + 9 = 7 \ll 2 + 9$ which results in the multiplierless realization shown in Figure 1.19. Similarly the coefficient 47 is decomposed to $47 = 15 \times 2 + 17 = 15 \ll 1 + 17$ with a realization shown in Figure 1.20. The multiplierless realization of the coefficients $5, 37$, and 47 is shown in Figure 1.21. Since the shift operation is carried out for free and the hardware cost of the addition and subtraction is the same [Yao et al., 2004; Ho et al., 2008; Farahani et al., 2010; Thong and Nicolici, 2011], the hardware complexity of the multiplierless MB is measured by two metrics, the number of adders in the block and the longest path of consecutive adders (the critical path). The number of adders is called the logic operators (LO) [Vinod and Lia, 2005], while the adder depth is called the logic depth (LD). Both the LO and LD determine the complexity of the MCM. The MB in Figure 1.21 costs LO = 7 adders and its logic depth (shown in red) LD = 2 adders. An optimization procedure usually follows this step to minimize the complexity of the multiplierles MB. The next section considers one such optimization procedure which is the CSE.

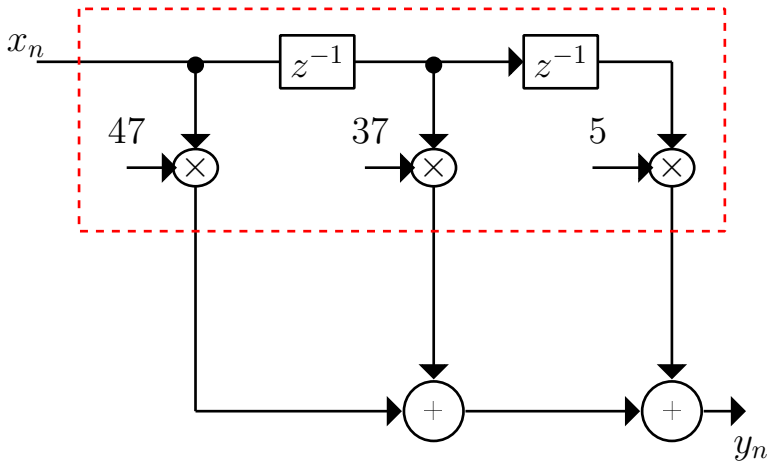


Figure 1.16: An example of FIR filter with coefficient set $\{5, 37, 47\}$.

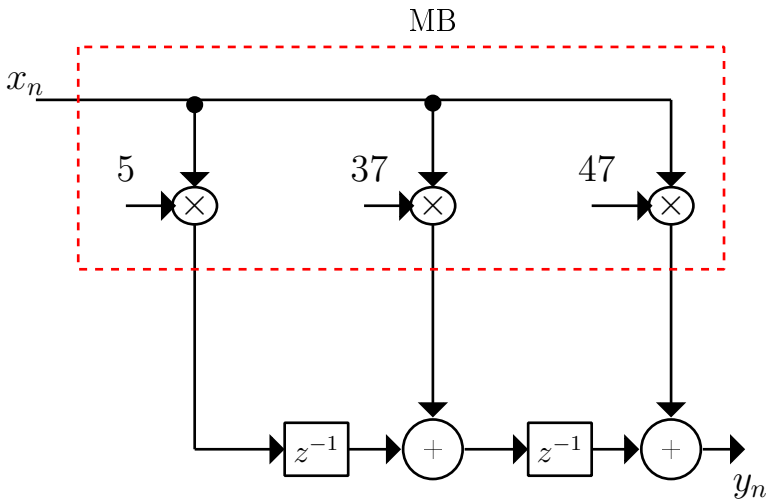


Figure 1.17: The transposed structure of the filter $\{5, 37, 47\}$.

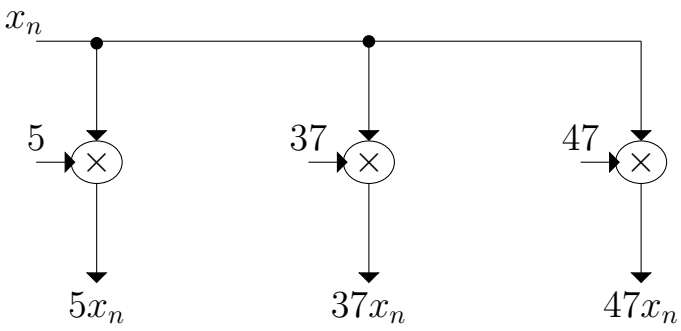


Figure 1.18: The multiplier block of the filter in Figure 1.17.

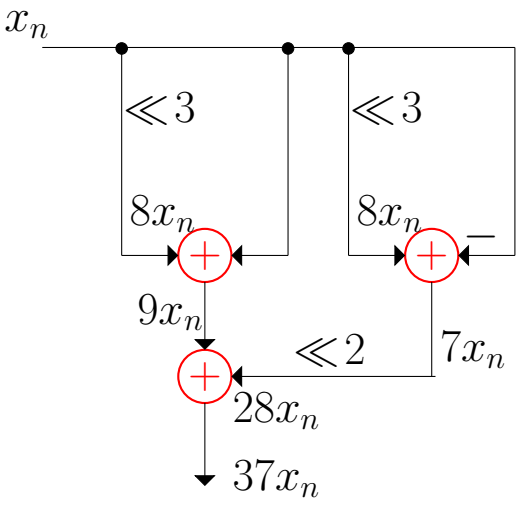


Figure 1.19: A possible multiplierless realization of the coefficient 37.

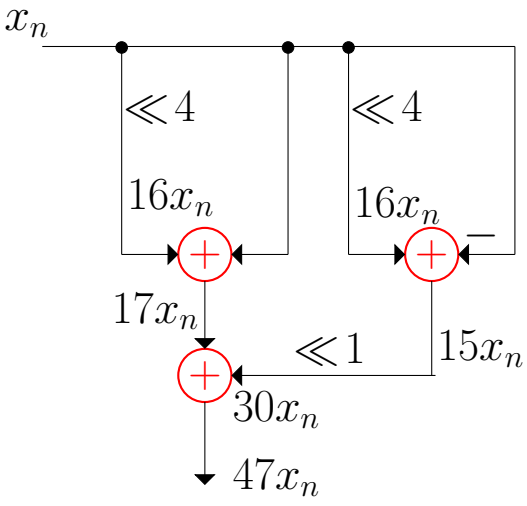


Figure 1.20: A possible multiplierless realization of the coefficient 47.

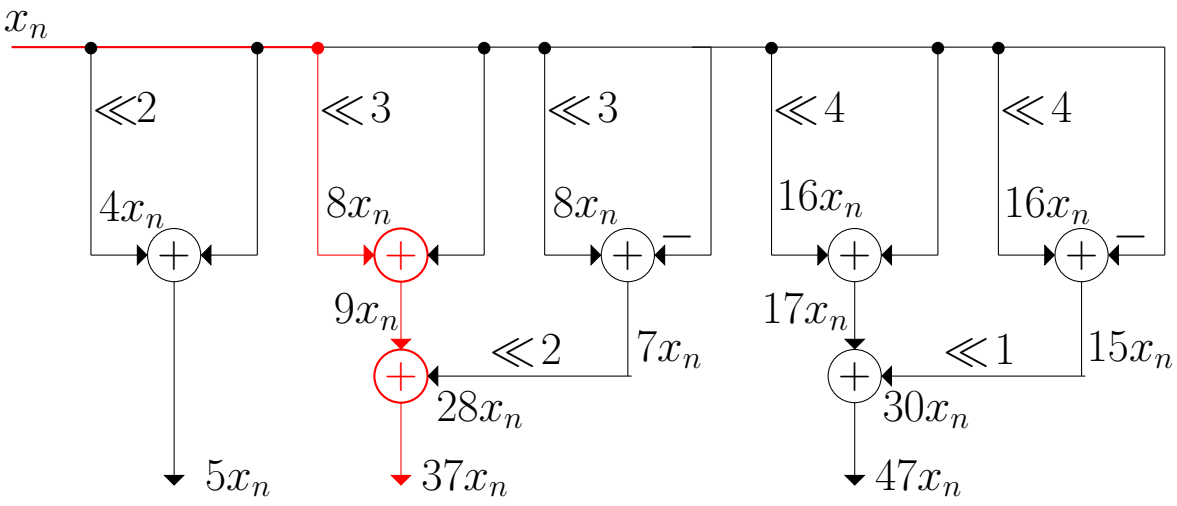


Figure 1.21: A possible multiplierless realization for the coefficients 5, 37, and 47.

1.4 COMMON SUBEXPRESSION ELIMINATION

In computer science, common subexpression elimination (CSE) is a compiler optimization method. It is a transformation that removes re-computations of common subexpressions and replaces them with saved values [Steven, 1997]. The same concept is used to minimize the MCM complexity by identifying and sharing common factors (subexpressions) within, and across, the coefficients. Finding and sharing the common factors among the coefficients requires first removing the multiplication operation from the MB. For example consider the MB shown in Figure 1.17, for which the multiplierless realization of the coefficient 5 is shown in Figure 1.6. Now, the coefficient 37 can be decomposed to $32+5$. The number 32 is just a shift to the left by 5 positions which can be obtained by redirecting the connections. The coefficient 5 is already synthesized, so it can be shared between the two coefficients. The result of sharing the computation between 5 and 37 is shown in Figure 1.22. The value 5 here is called a common subexpression and its sharing between the coefficients is the elimination process. In the same way, the constant 47 can be decomposed to $37+10 = 37+5\times 2 = 37+5 \ll 1$. Figure 1.23 shows the result of sharing the computation between the three coefficients. The total cost of the MB shown in Figure 1.17 is three adders, i.e $\mathrm{LO} = 3$ and its logic depth is $\mathrm{LD} = 3$ adders. However, if the coefficients are synthesized separately then the total cost is $\mathrm{LO} = \mathrm{LO}_5 + \mathrm{LO}_{37} + \mathrm{LO}_{47} = 1 + 2 + 3 = 6$ adders.

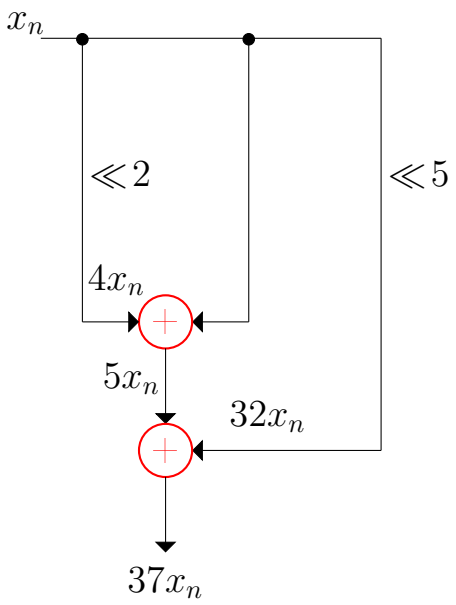


Figure 1.22: Sharing the computation between the coefficients 5 and 37.

Searching for different realizations is possible by selecting alternative decompositions for the coefficients. Consider the following decompositions for 37 and 47

$$37 = 32 + 5 \tag{1.3}$$

$$47 = 7 + 8 \times 5 \tag{1.4}$$

The number 5 in Equation 1.3 and Equation 1.4 is in red to indicate it is a common

subexpression. The result of sharing the computation between the three coefficients is shown in Figure 1.24. Here, the logic depth is reduced to two adders at the expense of increasing the logic operators by one adder.

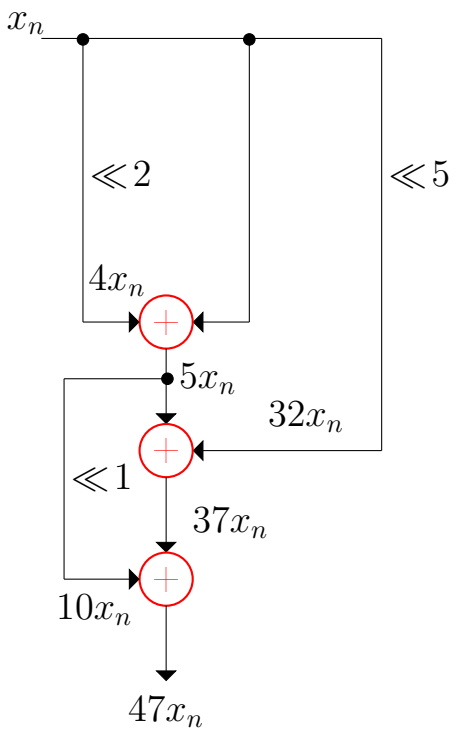


Figure 1.23: Sharing the computation between the coefficients $\{5, 37, 47\}$ after removing the multiplication operation.

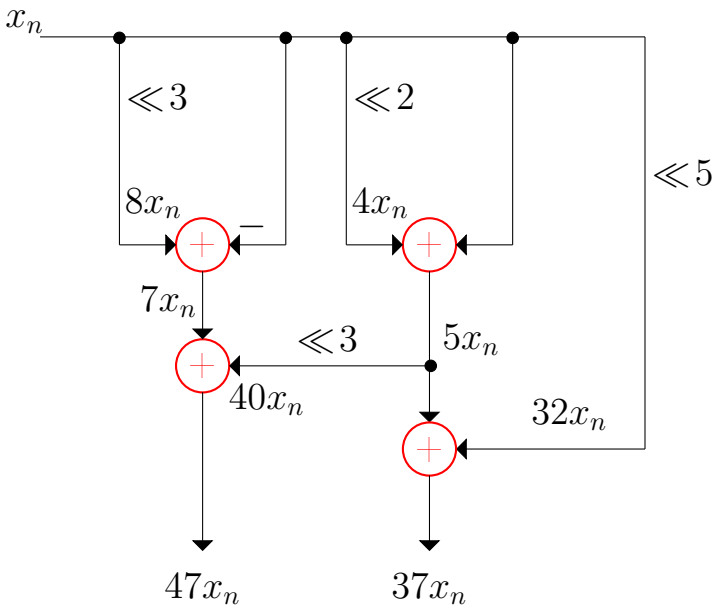


Figure 1.24: Another possible subexpression sharing between the coefficients $\{5, 37, 47\}$.

1.5 RADIX NUMBER SYSTEM

This section introduces the concept of the radix number system and specifically the redundant number system. The reason is the dependency of Chapters 2 and 3 on this concept. However, an algebraic development of redundant number systems is given in Chapter 4. The radix (positional) number system is characterized by the radix, r, and digit set, $\mathbb{D}$. Such number system represents a value using a string of digits picked from the digit set $\mathbb{D}$. Each digit in this string has a weight specified by its position in the string. The value v is evaluated from its representation using the evaluation mapping [Kornerup and Matula, 2010]

$$v = \sum_{i=0}^{L-1} d_i r^i, \tag{1.5}$$

where $d_i \in \mathbb{D}$ is a digit, L is the wordlength, d_0 is the least significant digit (LSD), and d_{L-1} is the most significant digit (MSD).

A number system is redundant if $|\mathbb{D}| > |r|$, i.e. the number of digits in the digit set (its cardinality) is greater than the radix [Kornerup and Matula, 2010]. Redundant number systems provide multi-representations for at least some decimal values. These systems were first introduced by Avizienis [1961] to speed up the addition operation through elimination of the carry propagation [Avizienis, 1961; Jaberipur and Saeid, 2010; Parhami, 1988; Phatak et al., 2001]. On the other hand, the number system is non-redundant if $|\mathbb{D}| = |r|$. In this case, each value has only one representation. For example, consider the traditional binary number system defined by $r = 2$ and $\mathbb{D} = \{0, 1\}$. This number system is non-redundant because $|\mathbb{D}| = |r| = 2$. However, adding the digit $\overline{1}$, which is of value -1, to the digit set makes the number system redundant. The resulting number system is called the binary signed digit (BSD) which is defined by $r = 2$ and $\mathbb{D} = \{\overline{1}, 0, 1\}$. The value 25 for example can be represented in BSD by $1\overline{1}11\overline{1}1$,$1\overline{1}101\overline{1}$, or $10\overline{1}001$. The number of non-zero digits in a representation is called the Hamming weight ($\mathcal{H}$) of the representation. Representations with a minimum number of non-zero digits are called minimum Hamming weight (MHW) representations. A unique non-redundant type of MHW is the canonical signed digit (CSD) representation. It is unique because the product of any two adjacent digits equals zero. For example, the representation $10\overline{1}001$ is CSD with $\mathcal{H} = 3$.

1.6 FINDING SUBEXPRESSIONS FROM REPRESENTATIONS

The number of possible decompositions for a given value is infinite. For example, the number 5 can be found from the decompositions $5 = 4 + 1 = 3 + 2 = 1 + 1 + 1 + 2 = 7 - 2 = 1005 - 1000 = \cdots$. To limit this number, these decompositions are related to the number representation. To explain the idea, consider the following FIR filter example,

$$y_n = 7x_n + 11x_{n-1}. \tag{1.6}$$

Assume that the coefficients are represented using BSD with wordlength $L = 4$. If the representations $(100\overline{1})$ and $(110\overline{1})$ are chosen to represent the numbers 7 and 11

respectively, then substituting these representations in Equation 1.6 yields

$$\begin{aligned} y_n &= (100\bar{1})x_n + (110\bar{1})x_{n-1}, \\ &= x_n \ll 3 - x_n + x_{n-1} \ll 3 + x_{n-1} \ll 2 - x_{n-1}. \end{aligned} \tag{1.7}$$

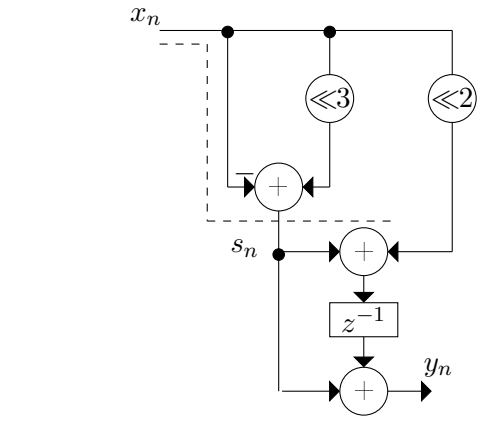
Rearranging Equation 1.7 gives

$$\begin{aligned} y_n &= -x_n + x_n \ll 3 - x_{n-1} + x_{n-1} \ll 3 + x_{n-1} \ll 2, \\ &= (-x_n + x_n \ll 3) + (-x_{n-1} + x_{n-1} \ll 3) + x_{n-1} \ll 2. \end{aligned} \tag{1.8}$$

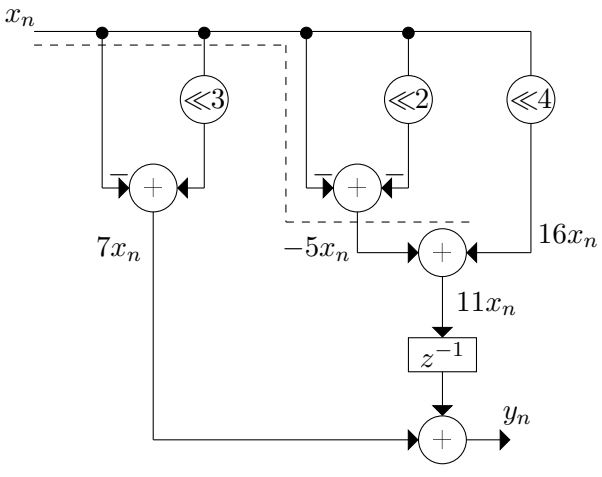
Identifying the term $s_n = -x_n + x_n \ll 3$ as a common subexpression in Equation 1.8 reduces it to

$$y_n = s_n + s_{n-1} + x_{n-1} \ll 2. \tag{1.9}$$

The multiplierless realization for Equation 1.9 is shown in Figure 1.25(a) with MB LO = 2 and LD = 2. Change the representation of 11 to $(10\bar{1}0\bar{1})$ results another realization as shown in Figure 1.25(b) which is with LO = 3 and LD = 2. The increment in the LO is because there is no common subexpressions can be found in the last choice of representations. This example shows the tight relation between the representations and subexpression elimination.



(a)



(b)

Figure 1.25: Synthesizing the filter $y_n = 11x_n + 7x_{n-1}$ using CSE and coefficient representations: (a) $7 = 100\bar{1}$ and $11 = 110\bar{1}$. (b) $7 = 100\bar{1}$ and $11 = 10\bar{1}0\bar{1}$.

1.7 COEFFICIENT'S COST AND ADDER STEP

A coefficient's cost is the number of adders required to synthesize the coefficient from its CSD representation (because it is of MHW). A coefficient is of cost 1 if it is synthesized using only one adder and of cost 2 if its synthesized using two adders and so on. A coefficient's adder step, δ, is the minimum number of adder steps required to synthesize the coefficient from its CSD representation. The coefficient's adder step terminology differs than the logic depth of the MCM because the latter is the longest path (MB critical path) of consecutive adders. Since the coefficients have a range of adder steps, the longest one specifies the minimum LD in the MCM. This minimum LD may be relaxed in some problems to increase the sharing between the coefficients to minimize the LO at the expense of the LD. To illustrate the difference between the coefficient adder step and the MCM logic depth, consider synthesis of the coefficient 103 with a CSD $10\bar{1}0100\bar{1}$ as shown in Figure 1.26(a). The cost of the coefficient 103 is LO = 3 adders or alternatively the coefficient is of cost 3. The adder step of the coefficient 103 equals $\delta = 2$ adders as shown in Figure 1.26(a). Then consider synthesis of the coefficient 281 with CSD $10010\bar{1}001$ as shown in Figure 1.26(b). The coefficient's cost is LO = 3 and its adder step equals $\delta = 2$. If the two coefficients are synthesized separately, the overall cost of the two realizations is LO = 6 adders with LD = 2. However, synthesis the two coefficients commonly results in the realization shown in Figure 1.26(c) with total cost of LO = 4 adders and LD = 3.

Briefly, coefficient cost and adder step are metrics used to measure the complexity of a coefficient. While logic operator and logic depth measure the complexity of MCM.

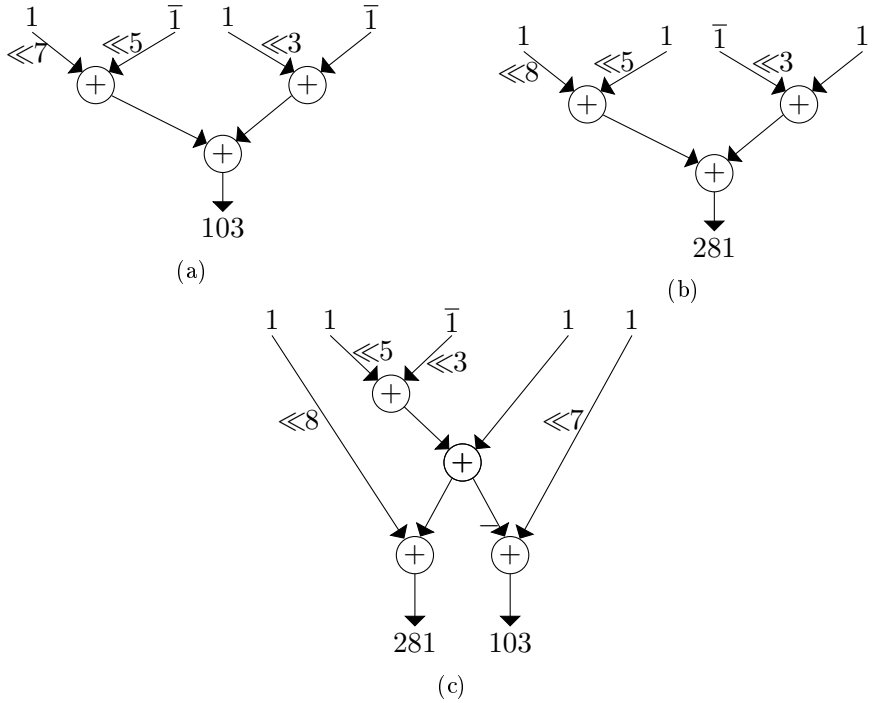


Figure 1.26: (a) The synthesis of 103 from its CSD representation $10\bar{1}0100\bar{1}$ costs LO = 3 adders with $\delta = 2$. (b) The synthesis of 281 from its CSD representation $10010\bar{1}001$ costs LO = 3 adders with $\delta = 2$. (c) The synthesis of 103 and 281 commonly by sharing the subexpression $10\bar{1}001$ between them reduces the total cost to LO = 4 adders and increases the logic depth to LD = 3 adders.

1.8 LOWER BOUND AND OPTIMALITY

The lower bound of the MCM problem is the solution that results in realizations with the same minimum number of LO's under a given LD constraint. These solutions are considered optimal in the sense that there are no other solutions for the problem with a lower number of LO's under the same LD constraint [Gustafsson, 2007b]. The minimum possible lower bound value of an MCM with N coefficients is equal to N adders. In this case, no extra subexpressions are required to synthesize the N coefficients. An example of an optimal solution is shown in Figure 1.27 for the coefficient set $3, 21, 53$ under the constraint LD = 3. The number of logic operator equals LO = N = 3. However, if the logic depth constraint is tightened to LD = 2, an extra subexpression of a value equals to 5 is required to share the coefficient 3 to synthesize the coefficient 53 as shown in Figure 1.28. This is also an optimal solution because there is no lower bound below the value LO = 4 adders that can be found as an optimal solution under the constraint LD = 2.

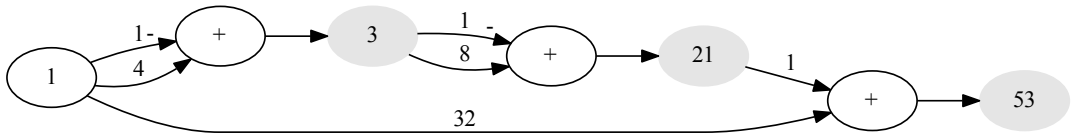


Figure 1.27: An example of the optimal solution for the coefficient set $\{3, 21, 53\}$. Logic depth constraint is relaxed to LD = 3.

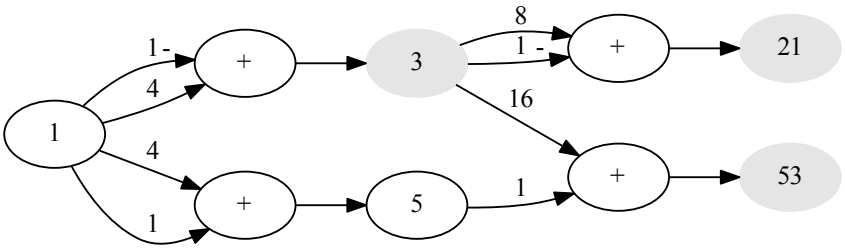


Figure 1.28: An example of the optimal solution for the coefficient set $\{3, 21, 53\}$. Logic depth constraint is tightened to LD = 2.

One of the hard instants of MCM is results when all the coefficients have the same value of adder step δ_m such that $\delta_m > 2$. The minimum number of additional adders required to synthesize a chain of subexpressions from the signal node (node of value 1) to reach the coefficient set is equal to $\delta_m - 1$. Synthesizing all the coefficients using this chain of subexpressions gives a lower bound of LO = $N + \delta_m - 1$ under the minimum logic depth constraint of LD = δ_m. If not all the coefficients have been synthesized using $\delta_m - 1$ extra adders under the LD constraint of δ_m, then the number of extra adders that required to synthesize the coefficients becomes $> \delta_m - 1$. The lower limit of LD is obtained when all the N coefficients are synthesized from the same extra subexpressions as describe above, while the upper limit of LD result from synthesizing all the coefficients as a chain of N adders (as in the graph dependent methods). This means that for MCMs

with large orders (large values of N), the logic depth will be in the order of $N+\delta_m-1$. In this case, the range of logic depth is between $\delta_m \leq LD \leq N+\delta_m-1$.

A coefficient is synthesized optimally if its constituted subexpressions are already synthesized. To minimize the LO and LD, each coefficient is assumed to be composed of two additive parts (subexpressions). In this case, the optimal cost of a coefficient equals one adder. Synthesizing a group of coefficients commonly using a set of subexpressions is optimal if the number of adders that required to synthesize them equals to the number of coefficients. If some of the group subexpressions are not synthesized yet, then the CSE algorithm tries to find a best sharing (maximum sharing) group of subexpressions to synthesize a group of coefficients. The best sharing is achieved when the number of extra adders required to synthesize the subexpressions is minimum.

1.9 CHAPTER SUMMARY

There is a market demand to design efficient DSP/DIP systems of low cost, consume low power, and work at high speed. The most computational intensive part of DSP/DIP systems is the MCM operation. The MCM is a core operation in DSP/DIP systems in which one variable multiplies many constants (coefficients). Therefore, finding an efficient MCM operation means that the underlying DSP/DIP system is efficient. The aim of this work is to find a method(s) that searches the possible MCM realizations for the optimum one.

The inherent multiplication operation in the MCM is a major obstacle to optimize its operation. Therefore, it is essential to remove the multiplication operation and substitute it with simpler ones of add/subtract and shift. It is possible to do that because the coefficients of the MCM are constants. The resulting MCM becomes multiplierless. However, doing this may not be straightforward because DSP/DIP structures are realized with the dot product operation such as in FIR and IIR filters. A transformation is required to convert the dot product operation to a MCM. An example is transforming the canonical structure (dot product) of the FIR filter to the transposed structure (MCM). Because the multiplication operation is substituted with add/subtract and shift, the new operations become metrics for the MCM complexity. The shift operation can be realized in hardware for free, while the cost of the hardware adder equals that of the subtracter. Therefore, the number of adders required to synthesize the MCM is considered as the number of logic operators and the critical path of the consecutive adders is the logic depth. Minimizing these two metrics will optimize the MCM operation, or in other words make it efficient.

The CSE is chosen as a powerful method to minimize the LO and LD. The method can outperform other techniques because it searches a large space of subexpressions. The subexpression space is generated from the redundant number representations. Trimming the search space without losing important information is considered in this work.

1.10 CONTRIBUTIONS IN THIS WORK

The work in this thesis presents a new direction in the field of solving the MCM by using CSE methods that depend on number representations. There are several important contributions have that been made in this field of science which include:

1. A numerous CSE methods are found in the literature. However, these methods can be classified into several types. We found that it is important to the research in this field to have a classification for the CSE methods. Therefore, the work introduced a classification of CSE methods according to the search space and how the method work. The classification shows that there are two main methods which are Hartley's table and multiplier block methods. Hartley's method used to tabulate the representations in $N \times L$ tables, where N is the order of the MCM and L is the wordlength of the MCM coefficients. Then the search starts to find the common patterns between the representations. On the other hand, the multiplier block used to search a space of subexpressions that either extracted from coefficient representations or calculated to achieve some criteria like reducing the search space.

2. Develop the pattern preservation algorithm (PPA) to resolve complicated overlapping patterns occur in Hartley's table method when using number representation sets that are larger than CSD and MHW. A set of redundant representations that larger than the MHW but less than the BSD is used to be the search space of the PPA. The set is called the zero-dominant set (ZDS). The number of representation combinations is smaller than BSD. However, the quality of solution is not too far than that obtained from using the whole BSD. The PPA itself follows a new method of searching Hartley's table which takes care of not eliminating a subexpression at the expense of losing another subexpressions.

3. Developing a tree and graph encoders using the concept of polynomial ring. The encoders are used to generate number representations of a given positional number system. Developing such algorithms is important to the research in this field because they can generate all of redundant representation.

4. A new direction of CSE algorithms is introduced that customize the tree encoder to search for common subexpressions simultaneously with the generation of number representations. This differs than traditional CSE methods that used to generate the search space then search for common subexpressions. Fusing representation generation and CSE in one process is an elegant way to make a non-exhaustive search of subexpressions. This because the CSE method tries finding optimal sharing between the coefficients during the process of representations generation. This part of work also introduced a mathematical development of the size of search space.

5. Developing a combinatorial model of the MCM using the subexpression space concept presented in this work. The model is a graph we called it the demand graph.

It is found that MCM problem is an arc routing problem. Solving the problem becomes a routing over the demand graph to find the routes with minimum complexity. Each route is correspond to a candidate solution for the MCM. However, only solutions of high quality are survived. Ant colony optimization metaheuristic algorithm is introduced to search the demand graph in parallel using computational ant agents. Metaheurstic algorithms like ant colony optimization can replace heuristic methods that is traditionally used to solve the MCM problem such as CSE and graph dependent GD methods. This is because the search space is now is the possible solutions of the MCM and solutions the was obtained from different heuristics can be obtained from routing the demand graph.

1.11 THESIS OVERVIEW

The chapters of this thesis are sorted hierarchically according to the development of our work. The reader will find a strong dependency and connection between the subsequent chapters. A brief overview for each chapter is presented in this section as follows.

Chapter 2 includes a literature survey on the CSE and GD methods. The two methods are the most common heuristics used to tackle the MCM. A historical development for the GD method is given. More space is given for the CSE method because it is the subject of this thesis. A classification for the most common CSE methods is presented in this chapter. Each CSE method is explained with an illustrative example.

A new common subexpression elimination algorithm called the pattern preservation algorithm (PPA) is developed in Chapter 3. The algorithm uses a set of redundant representations called the zero-dominant set to increase the number of identical digit patterns among the coefficient representations. The PPA tabulates the digits of the zero-dominant representations in an $N \times L$ table called Hartley's table, where N is the number of MCM coefficients and L is the wordlength. Each representation occupies a row in the table. The rows are ranked according to the coefficient ranks in the MCM. The columns are indexed by the digit positions in the representation starting from the least significant digit to the most significant digit. After tabulating the representations, the algorithm starts searching the table for horizontal common patterns (subexpressions). The patterns are prevented from being lost due to the collision (overlapping patterns) at the far/near end of the representation by introducing a digit set/reset technique. The overlapping occurs when one or more digits are shared between a number of patterns. Therefore, eliminating one pattern causes losing other patterns because of eliminating some of the common digits. This requires that the PPA uses two passes in searching each representation. In the first pass it searches for CSD patterns and in the second pass searches for clustered patterns. Before starting the second pass, the PPA resets the digit at the inner border of the far/near end stream of clustered non zero digits. The PPA sets the inner border digit to its original value after finishing its second pass. In this way the other patterns are saved from being destroyed so they can be used in the subsequent search. The drawback of using the PPA is the large number of representation combinations [Roberts and Tesman, 2009] which equal to the number of Hartleys' tables

and the difficulty of constraining the logic depth.

Chapter 4 presents an algebraic development for radix number systems. This development is fundamental to develop the tree and graph algorithms for generating redundant number representation as shown in Chapter 5. A polynomial ring is considered to find the sufficient and necessary conditions of a complete digit set. A digit set is considered complete if it can represent all the elements of the corresponding ring (i.e the ring of integer numbers $\mathbb{Z}$). Evaluating each polynomial to a single value partitions the representations to equivalent classes. A redundant number system arises if there is more than one polynomial evaluated to the same value.

Graph and tree algorithms are developed in Chapter 5 depending on the algebraic development shown in chapter 4. The algorithms are designed to generate the number representations (non-redundant and redundant) for any radix and digit set. The graph algorithm enumerates the representations on a graph while the tree algorithm generates the representation tree. The advantage of developing these algorithms is that they can be customized to search for common subexpressions. This customizing is a new direction in the CSE heuristics design. The size of the search space is determined by the size of the representation trees. Markov's transition matrix concept is used to find a closed form for the size of the representation tree.

A new subexpression elimination algorithm called the subexpression tree algorithm (STA) is developed in chapter 6 by customizing the representation tree. The STA searches for the subexpressions that maximize the sharing between the coefficients simultaneously with generating the representation tree. New subexpressions are generated at each born node. These subexpressions are used to find the decompositions of the value to be encoded. The algorithm examines these decompositions at each node to find the one with optimal sharing. The STA is designed to terminate the tree generation when it finds a decomposition with a maximum sharing. Only decompositions of two parts are considered because each decomposition consumes only one adder which is the minimum number of adders that required to compose a value. Decomposing a value into two parts is mathematically described by $\mathcal{A}-$operation. This relation is used to make each part indivisible by the radix which is called co-prime. The STA can find MCM realizations with minimum logic depth where other algorithms miss these realizations.

Chapter 7 addresses the final goal of this research which is deriving a combinatorial model for the MCM problem called the demand graph. The demand graph is a multigraph that is obtained from combining the MCM solutions after transforming the shift operation on the arcs to subexpression information. The solution of the MCM is found by constructing tours on the demand graph with each tour being a solution. A subexpression attached to an arc is considered as a demand on the arc traversal. This is considered as dynamic behavior because each new demand changes the tour plan to synthesize the demand itself. The dynamic behavior of the touring over the demand graph requires using metaheuristic algorithms to construct the tours. Ant colony optimization metaheuristic is proposed as a proper method to search the demand graph.

In Chapter 8, the ant colony optimization algorithm (ACO) is implemented to search the demand graph in parallel. The algorithm uses ant agents that search the

graph independently. Ant tours are compared for the one with minimum cost which is called minimum iteration tour. The minimum iteration tour become as a best-so-far tour if there is no other minimum iteration tour of cost less than it yet. Two implementation for the ACO algorithm are introduced in this chapter to search the demand graph. A serial implementation and a distributed computing implementation that use the C++ parallel boost graph library (PBGL).

Chapter 2

COMMON SUBEXPRESSION ELIMINATION

The MCM problem was defined as the problem of minimizing the logic operators (LO) and logic depth (LD). This problem is classified as NP-hard (unless NP=P) [Cappello and Steiglitz, 1984; Gustafsson, 2008; Voronenko and Püschel, 2007]. In other words, the optimum solution currently cannot be found in polynomial time and the computation time grows exponentially with the problem size. Heuristic techniques are often used to find good solutions. However, it is difficult to guarantee that a heuristic can find the minimum solutions for all possible MCM. Two optimization methods identified as being efficient techniques to tackle the MCM problem in polynomial time are the common subexpression elimination (CSE) technique proposed by Hartley [1991], and the graph dependent (GD) method proposed by Bull and Horrocks [1991].

The CSE technique searches for the most common subexpressions (patterns) in the coefficient representations [Yao et al., 2004; Dempster et al., 2004; Mahesh and Vinod, 2008; Banerjee et al., 2007; Thong and Nicolici, 2009]. Each common subexpression is synthesized once and its value is shared. On the other hand, the GD method represents the coefficients on a graph such that each coefficient is synthesized from that in the graph set [Dempster and Macleod, 1995; Gustafsson, 2008; Voronenko and Püschel, 2007; Gustafsson, 2007a; Gustafsson et al., 2006; Dempster and Macleod, 1994; Aksoy et al., 2010]. A typical GD consists of two parts [Dempster and Macleod, 1995], optimal and heuristic. In the optimal part, a GD method tries synthesizing an N coefficients MCM using only N adders. This may result in a long chain of N consecutive adders and consequently a very long LD [Yao et al., 2004]. Implementing MCM blocks with long logic depths increases the dynamic power consumption [Faust and Chang, 2010] and decreases the processing speed, which is one of the main concerns in mobile communication systems. Since the CSE methods searches a larger space of subexpressions, they have the potential to find solutions with shorter LD. Dempster et al. [2004] showed that it is better to use the CSE method for complex dot product problems while the GD method performs faster for simpler problems. Unifying the GD and the CSE methods requires finding an analogy to the MCM problem. To our knowledge, there is no obvious work in the literature that develops this analogy. The reason may be the nature of the problem which is NP-hard [Ho et al., 2008; Thong and Nicolici, 2011; Gustafsson, 2008; Voronenko and Püschel, 2007].

Usually CSE methods are designed to search small sets of number representations such as binary [Smitha and Vinod, 2007; Chang and Faust, 2010], canonical signed

digit [Farahani et al., 2010; Vinod et al., 2010], or some small subset of the binary signed digit (BSD) such as the minimum Hamming weight (MHW) representations [Park and Kang, 2001; Aksoy et al., 2012a], or MHW with an extension [Ho et al., 2008]. Searching such representations is possible because of their relatively small size, compared to the set of binary signed digit (BSD) representations. For example, searching the MHW representations means searching only those with the minimum number of nonzero digits. The search space is further reduced to a minimum size if the canonical signed digit (CSD) representation is used. Reducing the search space may cause losing of some subexpressions

This chapter is structured as follows: Section 2.1 presents a historical overview of the GD methods and how these methods solve the MCM problem. Section 2.2 introduces the classification of the CSE methods. Hartley's table methods are considered in Section 2.3. Multiplier block CSE methods are considered in Section 2.4. A summary of the chapter is given in Section 2.5.

2.1 GRAPH DEPENDENT METHODS

Bull and Horrocks [1987] proposed the first graph dependent algorithm, this later became known as the Bull-Horrocks algorithm (BHA). The algorithm synthesizes the coefficients regardless of their parity (odd or even). The coefficients are synthesized in ascending order using the partial sums in the graph (its vertices), otherwise the algorithm searches for partial sums with shorter distance from the graph. These partial sums require minimum additional adders to be synthesized from the graph set. Shift operations are realized using adders. To explain the BHA, consider the following FIR filter:

$$y_n = -42x_n + 44x_{n-1} - 40x_{n-2}. \tag{2.1}$$

We need to only consider the positive coefficients resulting the set $\{40, 42, 44\}$, as simple negation can synthesize a negative value. A directed signal flow graph is used to show the multiple product formation as shown in Figure 2.1. Nodes with a power of two values are first added to the graph, these are nodes $2, 4, 8, 16$, and 32. These partial sums are used to synthesize the coefficients. The coefficient 40 is obtained from adding 8 and 32, 42 from adding 40 and 2, and 44 from adding 42 and 2.

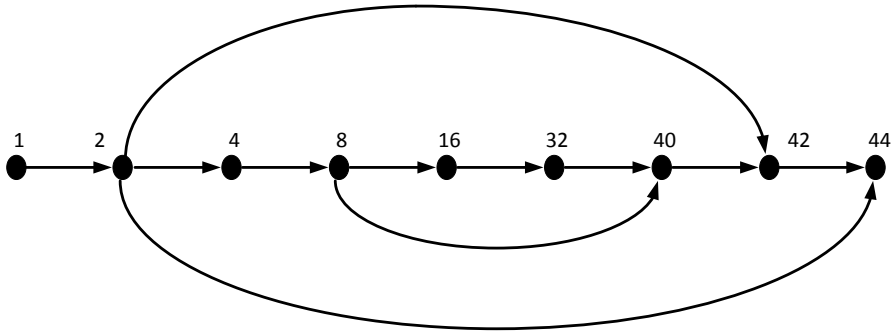


Figure 2.1: Directed graph showing the BHA.

The same authors improved this work in [Bull and Horrocks, 1991] by increasing the partial sums set through forming a power of two subgraph on every vertex in the graph. This was the early work that recognized the shift operation as the logic operator with lowest cost when compared with addition or subtraction. Dempster and Macleod [1994] modified the BHA algorithm by permitting the partial sums to have values greater than the coefficients to be synthesized. Only odd fundamentals (coefficients and partial sums) are used. In the same paper, they proposed an algorithm which considers all the possible graph topologies (shown in Figure 2.5) of an integer to find the one that results in the minimum adder cost.

Logic depth concerns were considered in [Dempster et al., 2002] by testing the effect of each coefficient on the resulting adder depth before synthesizing this coefficient. However, the authors stated that the algorithm is computationally intensive.

Gustafsson [2008] proposed a hypergraph approach to solve the multiple constant multiplication (MCM) problem. In this method, an optimal solution is found by using the minimum spanning tree with the shortest length to connect the whole set of coefficients. If the algorithm does not find directed paths from the root vertex to some coefficients, Steiner vertices [Gustafsson, 2008] are found such that the resulting spanning tree is of minimum cost and depth. To reduce the length of the spanning tree, extra intermediate vertices and edges may be added to the graph. Han and Park [2008] proposed a multiple adder graph algorithm to take into account the future effect of a particular coefficient on the rest of the coefficients. The authors showed that the computational complexity to find all the possible adder graphs is high even for one coefficient. An exact depth-first search algorithm to find the minimum solution is proposed by Aksoy et al. [2010]. A comprehensive study of GD methods can be found in [Voronenko and Püschel, 2007].

In general, the GD method is summarized as follows:

1. Fundamentals are represented on an acyclic directed graph with the root vertex being the signal node and equal 1. Each vertex (except the root) has in-degree 2[1]. There is no limit to the out-degree [Dempster and Macleod, 1995].
2. Negative fundamentals are made positive.
3. Even values are made odd by successive right shifting.
4. Any repetition in the resulting positive odd fundamentals is removed.
5. Try synthesizing the coefficients from that in the graph set using minimum number of additional adders.
6. Each newly synthesized fundamental is moved to the graph set.

[1] In-degree of a vertex is the number of input edges to this vertex

For example consider the filter in Equation 2.1, its canonical structure is shown in Figure 2.2 (a). Making the coefficients positive odd results in

$$-42 \rightarrow 21,\ 44 \rightarrow 11,\ -40 \rightarrow 5.$$

The structure in Figure 2.2 (b) is obtained by transferring the negative signs from the coefficients to the input of the structure adders and assigning the shift operations to the signal paths.

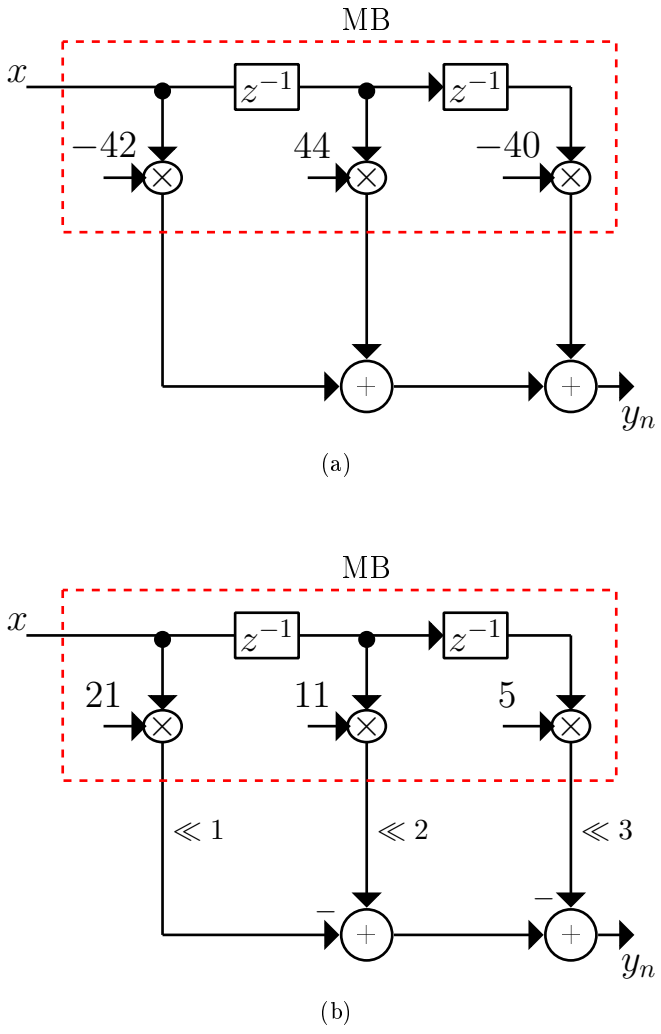
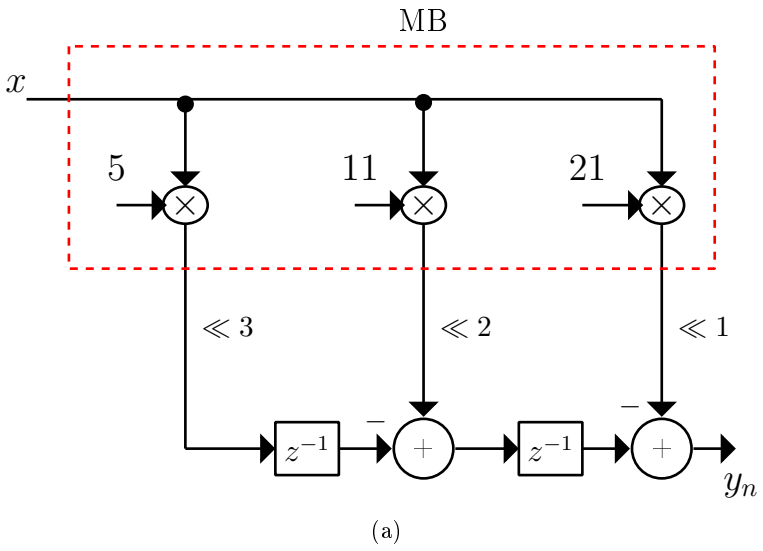


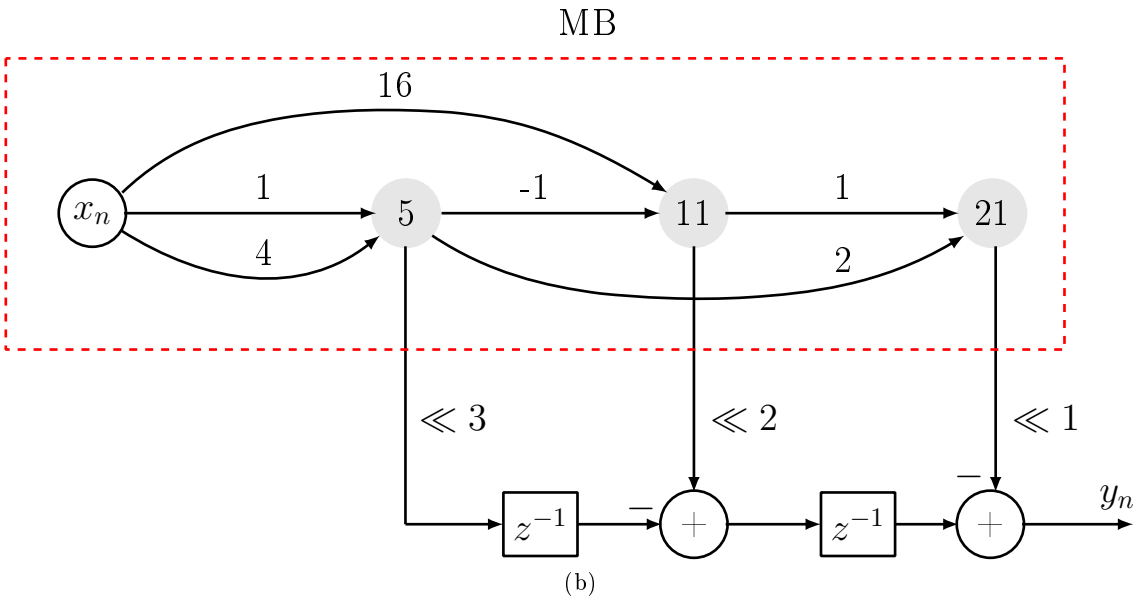
Figure 2.2: Using the GD method to synthesize the coefficient set $\{-42, -40, 44\}$. (a) The canonical structure of the filter. (b) Making the coefficients positive odds.

The transposed structure of the filter is shown in Figure 2.3 (a). The dashed box in Figure 2.3 (a) encloses the multiplier block (MB). The graph method starts by synthesizing all of the cost 1 coefficients. In this example, there is only one coefficient with cost 1, which is 5. After synthesizing 5, the graph set is now $\{1, 5\}$. The coefficient 11 can be synthesized from the graph set because $11 = 1 \ll 4 - 5 = 16 - 5$. The graph set is now $\{1, 5, 11\}$. The last coefficient 21 is synthesized as $21 = 11 + 5 \ll 1 = 11 + 10$. The result of synthesizing the coefficients is shown in Figure 2.3 (b). Here, each vertex inside the MB (except the signal node) is an adder and its output becomes the vertex label. These adders are called multiplier block adders (MBA). The other two adders

outside the MB of Figure 2.3 (b) are the structure adders. The number of logic operators in the MB is LO = 3 and the logic depth is LD = 3. The overall filter cost equals 5 adders (neglecting delay elements cost).



(a)



(b)

Figure 2.3: (a) The transposed structure of the filter in Figure 2.2. (b) Synthesis the filter using the GD method.

Another example is to synthesize the filter shown in Figure 2.4 (a) with coefficient set $\{5, 37, 47\}$. The steps of synthesizing the MB are shown in Figure 2.4 (b)-(d). These graphs represent the different graph topologies [Johansson, 2008]. Examples of other graph topologies are shown in Figure 2.5. Graph methods precompute and store all these topologies then search them for the minimum realization.

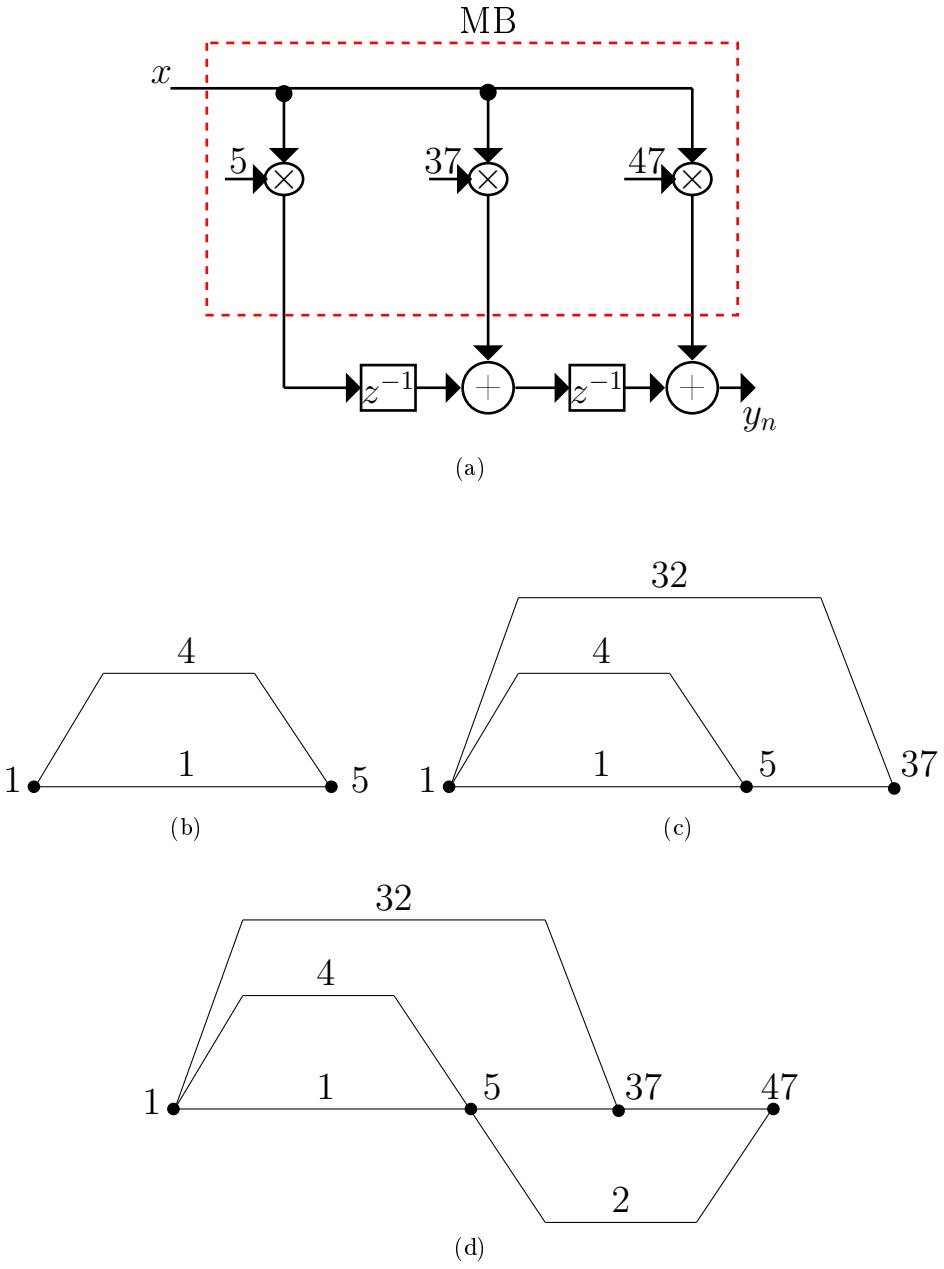


Figure 2.4: Synthesis of the multiplier block (MB) of the FIR filter with coefficient set $\{5, 37, 47\}$. (a) The transposed structure of the filter. (b) The coefficient 5 is synthesized using cost 1 graph topology. (c) The coefficient 37 is found from 5 using one of cost 2 topologies. (d) A cost 3 graph topology is found such that the coefficient 47 is synthesized from 5 and 37.

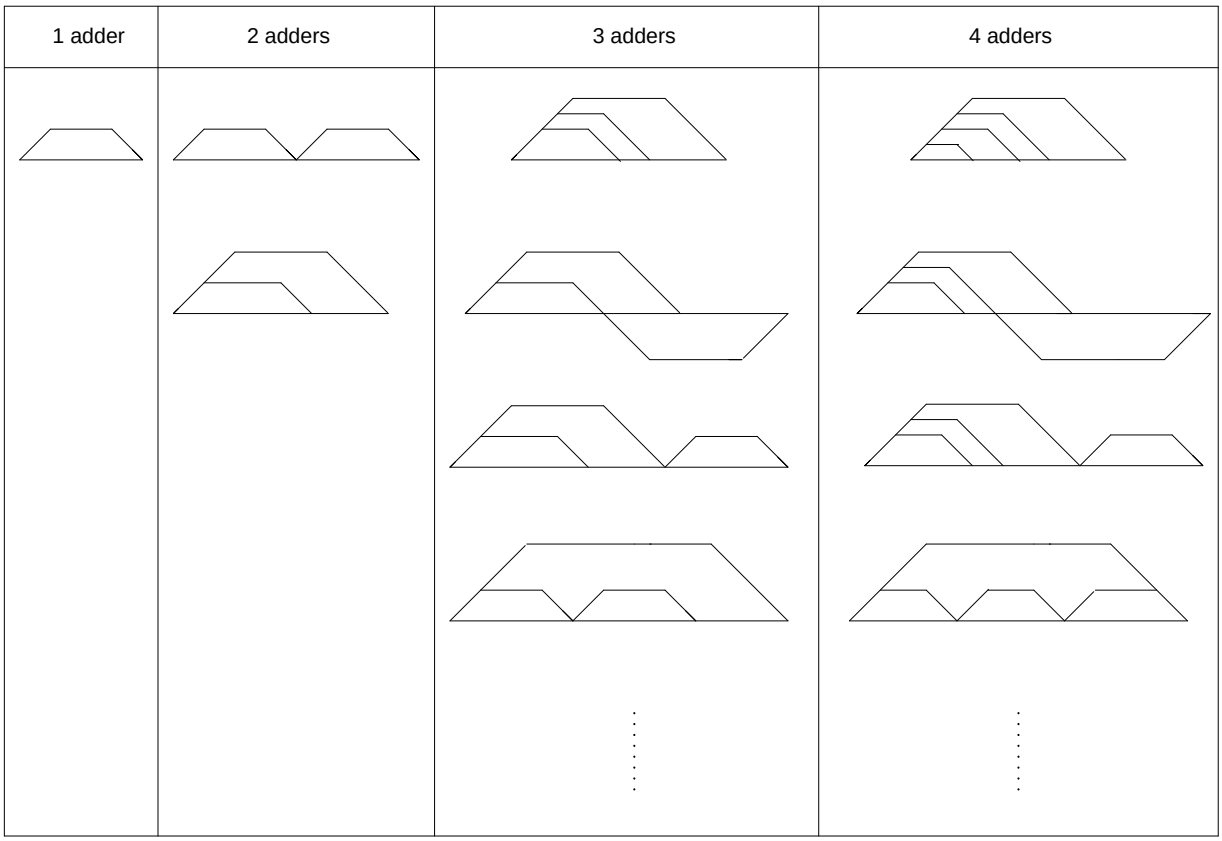


Figure 2.5: Examples of different graph topologies.

2.1.1 The N-Dimensional Reduced Adder Graph Algorithm

The N-Dimensional Reduced Adder Graph (RAG-N) algorithm was proposed by Dempster and Macleod [1995]. The algorithm consists of two parts: Firstly, the optimal part which is designed to find the realization of N, non-repeated odd, coefficients using only N adders. This number of adders is the lower bound of the MCM problem because it is impossible to synthesize N distinct coefficients using M adders for $M < N$ [Dempster and Macleod, 1995]. The algorithm synthesize the fundamentals using those in the graph set (synthesized). Each synthesized fundamental is composed using at most one adder. Secondly, the heuristic part uses lockup tables that shown in Figure 2.5 to store the optimum costs of a single coefficient and the different possibilities of fundamentals that can be used to implement it [Dempster and Macleod, 1995].

The RAG-N algorithm steps are as following:

1. Make all coefficients positive odd fundamentals and remove all duplication and cost-0 fundamental if any is exist (all power of two fundamentals are reduced to 1). Move these fundamentals to incomplete set.

2. Calculate the cost of each coefficient by using the cost lookup table.

3. Move all cost 1 fundamentals if any from the incomplete set to the graph set. These fundamentals are composed by using only one adder. For example, the number 7 is of cost 1 because it can be composed as $7 = 8 - 1$.

4. For all possible fundamental pairs in the graph set, check whether their sum or its multiplications by power of two numbers equals to coefficients in the incomplete set.

5. Remove all the coefficients produced in step 4 from the incomplete set to the graph set. Since these coefficients are synthesized using only one adder, they are of distance 1 adder from the graph set.

6. Repeat step 5 until no more fundamentals are added to the graph set.

   The solution is optimal if all the coefficients are synthesized by the end of step 6 and the incomplete set becomes empty coefficient set if it is to be an optimal cost block.

   The heuristic part of the algorithm continues as follows:

7. Find the difference between each coefficient in the incomplete set and each fundamental in the graph set. If any difference is found of cost 1, this difference is synthesized and the corresponding coefficient is synthesized too by summing the difference coefficient and the examined fundamental in the graph set.

8. Find the difference between each coefficient in the incomplete set and the sums of every pair of fundamentals in the graph set. If any difference is found of value zero, this difference is synthesized and the corresponding coefficient is synthesized too by summing the difference coefficient and the sum of the examined two fundamentals in the graph set.

9. Repeat steps 6, 7, and 8 until no new distance 1 or 2 fundamentals are found.

10. Synthesize residual coefficients in the incomplete set by choosing arbitrary fundamentals of lowest sum. This procedure starts with selecting from the incomplete set the coefficient of minimum cost. While, the selected fundamentals are added to the incomplete set.

11. Repeat steps 6-10 until all coefficients are synthesized.

The RAG-N is a powerful heuristic used to synthesize the MCM. However, its optimal part may accomplished at the expense of LD. For example, if the RAG-N is used to synthesize the coefficients $5, 21, 23, 205, 843, 3395$ it results in the realization shown in Figure 2.6. This realization is optimal because it results from the optimal part of the algorithm with each coefficient is at distance 1 from the graph set. The cost of the realization in Figure 2.6 is LO = 6 and the logic depth is LD = 6 adders. This shows that the optimal part may result in realizations with long logic depth which is a disadvantage in high speed, low power consumption DSP systems. Synthesis the same coefficients using the CSE method that proposed in [Yao et al., 2004] results the realization shown in Figure 2.7. In this example, the CSE method succeed in synthesizing the MCM using a shorter logic depth of 3 adders but at the expense of increasing the number of LO which raised to 10 adders. The method in reference [Yao et al., 2004] is described in details in Subsection 2.4.2.

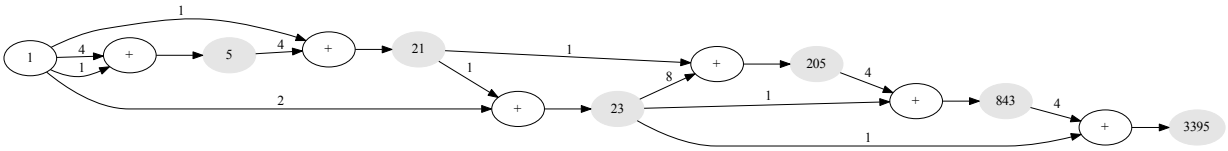


Figure 2.6: Synthesis the coefficients $5, 21, 23, 205, 843, 3395$ using RAG-N algorithm results in LO = 6 and LD = 6.

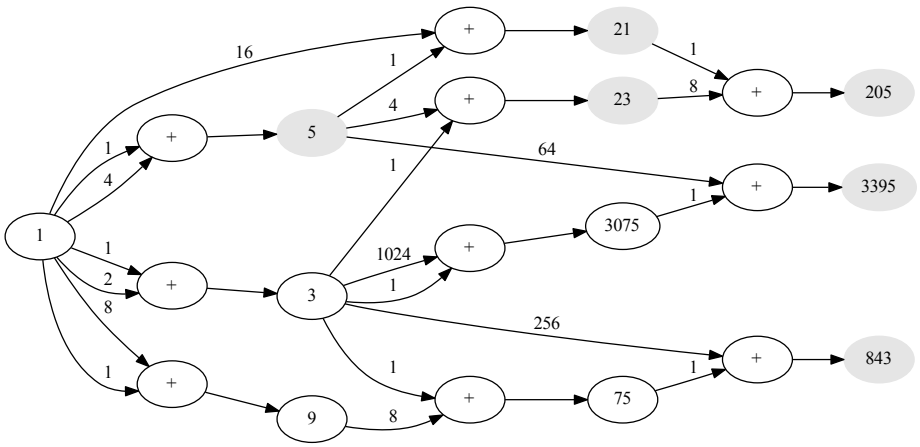


Figure 2.7: Synthesis the coefficients $5, 21, 23, 205, 843, 3395$ using the CSE method of reference [Yao et al., 2004] results in LO = 10 and LD = 3.

2.2 COMMON SUBEXPRESSION ELIMINATION

Common subexpression methods can be classified into two main categories as shown in Figure 2.8. Firstly, Hartley's method [Hartley, 1991, 1996] which works on the FIR filter canonical structure. Hartley's method in turn is classified into four methods, the horizontal common subexpressions (HCSE), the vertical common subexpressions (VCSE), the oblique common subexpressions (OCSE), and the mixed common subexpression elimination (MCSE). Secondly, the multiplier block (MB) method which works only on the transposed structure of the filter. In turn MB methods can be subclassified into an HCSE that works on the multiplier block and denoted by HCSE-M, and representation independent (RI) methods. These classifications are considered in more details in the next sections.

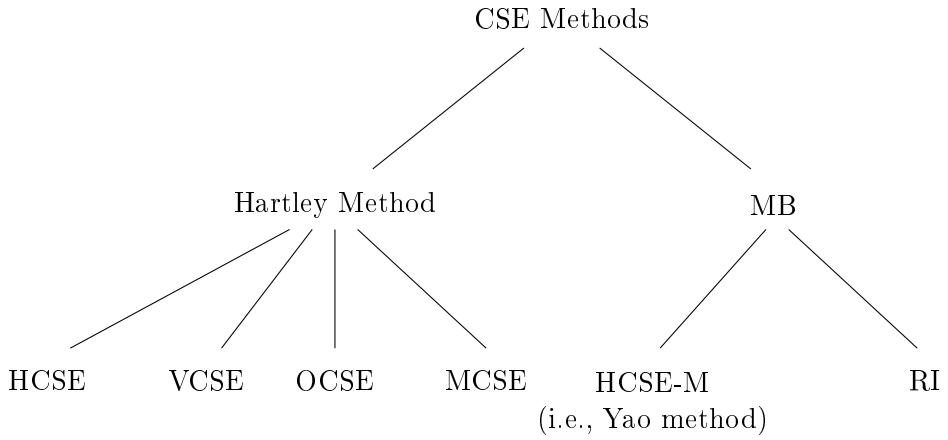


Figure 2.8: Classification of the CSE Methods.

2.3 HARTLEY'S METHOD

Hartley's method arranges the representations in a $N \times L$ table, called Hartley's table, where N is the number of coefficients and L is the wordlength. The table is searched for the common patterns (subexpressions). The number and type of subexpressions are affected directly by the representations. Three types of digit patterns can be identified in the Hartley's table as shown in Figure 2.9, the horizontal common subexpressions (HCSs) shown in red, the vertical common subexpressions (VCSs) shown in green, and the oblique common subexpressions (OCSs) shown in blue. Therefore, CSE algorithms that work on Hartley's table can be classified into HCSE, VCSE, OCSE, and MCSE as shown in Figure 2.8. The MCSE method works on two or more types of common subexpressions, i.e. horizontal and vertical, vertical and oblique, and so on.

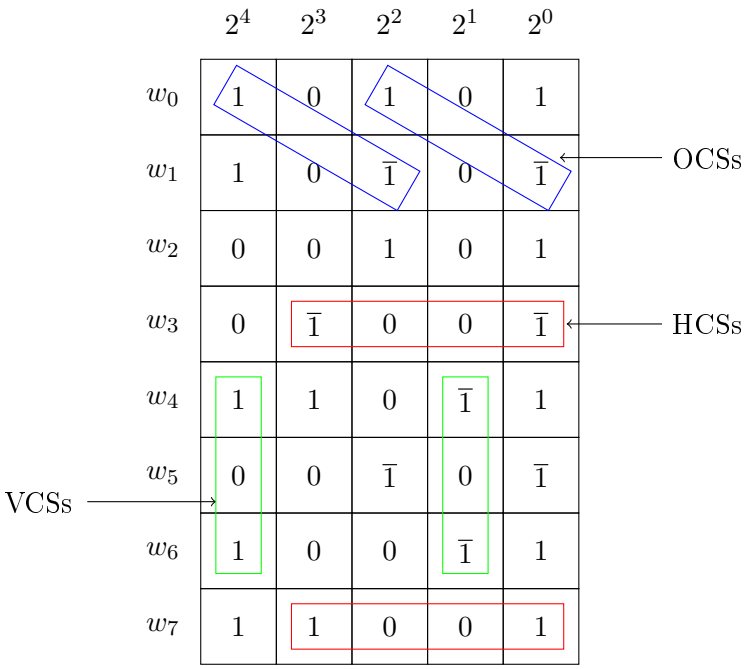


Figure 2.9: The three possible patterns that can be found in Hartley's table.

2.3.1 Horizontal Common Subexpression Elimination

The HCSE method finds and eliminates the HCSs such as $\{101, 10\bar{1}, 1001, \cdots\}$. These subexpressions are found in the CSD or BSD representations. To explain the HCSE method, consider the FIR filter

$$y_n = 21x_n + 11x_{n-1} + 5x_{n-2}. \tag{2.2}$$

The BSD representations are calculated using an exhaustive enumerating of all the possible combinations of the digits $\bar{1}, 0, 1$ with a length of 5 digits as shown in Table 2.1. Then, using the evaluation mapping in Equation 5.2 maps the representations to their corresponding values as shown in the second column of Table 2.1. The other subsets of BSD such as the MHW and CSD can be obtained from Table 2.1 by considering their characteristics (i.e., MHW are representations with minimum number of non-zero digits). In this case, the BSD representations of the coefficients $\{5, 11, 21\}$ are shown in Figure 2.2. Finding the optimal solution requires searching all the possible combinations of the representations shown in Figure 2.2. These combinations are given by:

1. $(1\bar{1}0\bar{1}\bar{1}), (10\bar{1}0\bar{1}), (110\bar{1}\bar{1})$

2. $(1\bar{1}0\bar{1}\bar{1}), (10\bar{1}0\bar{1}), (11\bar{1}1\bar{1})$
   $\vdots$

If the number of coefficients is N and the coefficient w_0 has n_{w_0} representations, the coefficient w_1 has n_{w_1} representations, and so on. The number of combinations C (Hartleys' Tables) is given by [Roberts and Tesman, 2009]:

$$C = n_{w_0} \times n_{w_1} \cdots \times n_{w_{N-1}}. \tag{2.3}$$

For the example in Table 2.2, the number of combinations equals to:

$$C = n_5 \times n_{11} \times n_{21} = 8 \times 8 \times 5 = 320.$$

This means that the HCSE method should search 320 Hartley's table to find the optimal solution. For example, the combination 00101, $10\bar{1}0\bar{1}$, and 10101 is chosen arbitrary corresponding to 5, 11, and 21 respectively. These representations are tabulated in Hartley's table as shown in Table 2.3. This is mathematically equivalent to substitute the representations in Equation 2.2 which results in

$$y_n = (10101)x_n + (10\bar{1}0\bar{1})x_{n-1} + (00101)x_{n-2}. \tag{2.4}$$

The patterns 101 and $\bar{1}0\bar{1}$ shown in Table 2.3 are enclosed by rectangles to indicate that these are calculated to the same value (same subexpression). It can be identified

mathematically by rewriting Equation 2.4 as

$$\begin{aligned} y_n = \quad & (x_n \ll 4 + x_n \ll 2 + x_n) + (x_{n-1} \ll 4 - x_{n-1} \ll 2 - x_{n-1}) + (x_{n-2} \ll 2 + x_{n-2}), \\ = \quad & (x_n \ll 4 + s_n) + (x_{n-1} \ll 4 - s_{n-1}) + s_{n-2}, \end{aligned} \tag{2.5}$$

where

$$s_n = x_n + x_n \ll 2. \tag{2.6}$$

Table 2.1: Enumerate the BSD representations with wordlength equals $L = 5$.

rep.	**value**	**type**
00000	0	-
00001	1	CSD
$0000\bar{1}$	-1	CSD
00010	2	CSD
00011	3	MHW
$0001\bar{1}$	1	BSD
00100	4	CSD
00101	5	CSD
$0010\bar{1}$	3	CSD
00110	6	MHW
00111	7	BSD
$0011\bar{1}$	5	BSD
01000	8	CSD
01001	9	CSD
$0100\bar{1}$	7	CSD
	⋮	

Figure 2.10 shows the realization of the filter after applying the HCSE method. Hartley's methods use the canonical structure of the FIR filter, therefore the number of logic operators represents the total adder cost of the filter. In this case, the adder cost of the filter equals LO = 5 adders. To compare this result with that of other methods (like GD), the number 5 is broken into 2 structure adders and 3 adders that correspond to the MB adders in the GD methods. In this case, the value of 3 adders is equal to that obtained in the GD method described in Section 2.1. The logic depths are the same for the two methods ands equal to LD = 3.

Table 2.2: BSD representations of the coefficients $\{5, 11, 21\}$.

5	**11**	**21**
$(1\bar{1}0\bar{1}\bar{1})$	$\boxed{(10\bar{1}0\bar{1})}$	$(110\bar{1}\bar{1})$
$(010\bar{1}\bar{1})$	$(1\bar{1}10\bar{1})$	$(11\bar{1}1\bar{1})$
$(1\bar{1}\bar{1}1\bar{1})$	$(0110\bar{1})$	$(1011\bar{1})$
$(01\bar{1}1\bar{1})$	$(10\bar{1}\bar{1}1)$	$(11\bar{1}01)$
$(0011\bar{1})$	$(1\bar{1}1\bar{1}1)$	$\boxed{(10101)}$
$(1\bar{1}\bar{1}01)$	$(011\bar{1}1)$	
$(01\bar{1}01)$	$(1\bar{1}011)$	
$\boxed{(00101)}$	(01011)	

Table 2.3: Finding the HCSs among the representations of the coefficients $\{5, 11, 21\}$.

	2^4	2^3	2^2	2^1	2^0
$w_0 = 21$	1	0	$\boxed{1}$	0	$\boxed{1}$
$w_1 = 11$	1	0	$\boxed{\bar{1}}$	0	$\boxed{\bar{1}}$
$w_2 = 5$	0	0	$\boxed{1}$	0	$\boxed{1}$

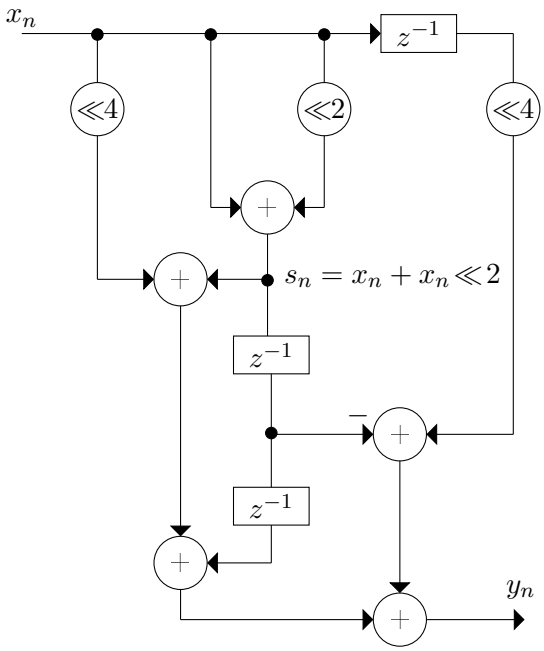


Figure 2.10: The result of using the HCSE to synthesize the coefficients $\{5, 11, 21\}$.

Another variation of the HCSE was proposed by Potkonjak et al. [1996] which is called the iterative matching algorithm (ITM). The algorithm finds the best matched patterns across each coefficient pair. Sign digit (SD) representations are used in this work. The authors used a greedy algorithm to select the best SD representations. A modified iterative matching (MITM) algorithm is proposed by Farahani et al. [2010]. The MITM algorithm calculates the number of nonzero bits for all pairs of coefficients to detect and eliminate more common subexpressions. To explain the basis of both the ITM and MITM, consider the coefficient set $\{200, 206, 655, 281\}$ given in [Farahani et al., 2010]. The CSD representations of the coefficients are shown in Table 2.4. The number of bitwise matches among all pairs of constants is calculated as shown in Table 2.5. The method requires calculating the number of nonzero bits for all pairs of constants as shown in Figure 2.6. Only N pairs are survived from the previous calculations, where N is number of coefficients. The selection is made according to the maximum number of bitwise matches found in Table 2.5 and in the same time have a minimum number of nonzero bits that found in Table 2.6. An evaluating function is used to find the best match in the set of candidate pairs. It is based on both the immediate saving in the LO and the later saving. The results of evaluating the function for the example in Table 2.4 shows that the pair w_2 and w_3 is the best match with a common pattern $000010\bar{1}001$ as shown in Table 2.7. Eliminating the pattern $000010\bar{1}001$ from the CSD representations of the coefficients w_2 and w_3 results in a new CSD representations as shown in Figure 2.8. The coefficients in Figure 2.8 are searched for the common patterns $10\bar{1}001$ and $\bar{1}0100\bar{1}$. These patterns are found in w_0 and w_1 respectively and identified by bold as shown in Figure 2.8. After eliminating these patterns, the process continue until there is no bitwise matches larger than 1.

coefficient	**value**	**csd**
$\boldsymbol{w_0}$	200	$010\bar{1}001000$
$\boldsymbol{w_1}$	206	$010\bar{1}0100\bar{1}0$
$\boldsymbol{w_2}$	655	$1010101\bar{0}01$
$\boldsymbol{w_3}$	281	$010010\bar{1}001$

Table 2.4: CSD representations of the coefficients $\{200, 206, 655, 281\}$.

coefficient	$\boldsymbol{w_0}$	$\boldsymbol{w_1}$	$\boldsymbol{w_2}$	$\boldsymbol{w_3}$
$\boldsymbol{w_0}$	-	2	0	1
$\boldsymbol{w_1}$	2	-	0	1
$\boldsymbol{w_2}$	0	0	-	3
$\boldsymbol{w_3}$	1	1	3	-

Table 2.5: The number of bit-wise matches among all pairs of coefficients.

coefficient	w_0	w_1	w_2	w_3
w_0	-	7	8	7
w_1	7	-	9	8
w_2	8	9	-	9
w_3	7	8	9	-

Table 2.6: The number of nonzero bits for all pairs of constants.

w_2	$101010\bar{1}001$	$000010\bar{1}001$
w_3	$010010\bar{1}001$	

Table 2.7: A best match is found in the pair w_2 and w_3 with a common pattern $000010\bar{1}001$.

coefficient	**value**	**csd**
w_0	200	$0\mathbf{10\bar{1}001}000$
w_1	206	$010\mathbf{\bar{1}0100\bar{1}}0$
w_2	655	1010000000
w_3	281	0100000000

Table 2.8: The coefficients after eliminating the common pattern $000010\bar{1}001$.

The other variation of HCSE proposed by Paško et al. [1999] in which the CSE problem is formulated as a linear transform where a $L \times N$ matrix is used to represent Hartley's table. The coefficients are represented using CSD. Common patterns are searched across the rows starting with the ones of largest Hamming weight. A steepest descent approach was used to select the pattern with the highest frequency. Selecting the pattern with the highest frequency splits the matrix into two matrices, the common subexpression matrix and the remainder. The process is repeated recursively until no common subexpressions can be found in the remainder matrix. Marcos et al. [2002] modified the method proposed in [Paško et al., 1999] by decreasing subexpressions sharing to elevate the recursive use of a common subexpression. However, this could be at the expense of increasing the LO.

2.3.2 Vertical Common Subexpression Elimination

The VCSE method searches and eliminates the VCSs. This method may give better results than the HCSE for some coefficient sets. Vinod et al. [2010] showed such a case, but in the same article the authors mentioned that the VCSE may be not so useful in

realizing symmetric coefficients. In this case, additional adders are required to keep the symmetry which increases the number of LO. To explain the VCSE method, consider the same filter example used in Subsection 2.3.1. The representations are tabulated as shown in Figure 2.11. The vertical pattern $1\bar{1}1$ is chosen to be a common subexpression. The equation for this pattern is given by

$$s_n = x_n - x_{n-1} + x_{n-2}, \tag{2.7}$$

and the output equation is

$$y_n = s_n + s_n \ll 2 + x_n \ll 4 + x_{n-1} \ll 4. \tag{2.8}$$

The number of LO in Equation 2.8 is 5 adders which is the same of that obtained from using the HCSE described in Subsection 2.3.1.

	2^4	2^3	2^2	2^1	2^0
$w_0 = 21$	1	0	1	0	1
$w_1 = 11$	1	0	$\bar{1}$	0	$\bar{1}$
$w_2 = 5$	0	0	1	0	1

Figure 2.11: Finding the vertical common subexpressions (VCSs) among the CSD representations of the coefficients $\{5, 11, 21\}$.

2.3.3 Oblique Common Subexpression Elimination

Vinod et al. [2010] showed that OCSE is like VCSE because they may destroy the symmetry of the coefficients in symmetric filters which require using additional adders to keep the symmetry. Figure 2.12 shows the oblique patterns for the same coefficients shown in Section 2.3.1. The equation of the oblique common subexpression is

$$s_n = x_n \ll 2 - x_{n-1}. \tag{2.9}$$

The output equation is reduced to

$$y_n = x_n + s_n + s_n \ll 2 + x_{n-1} \ll 4 + x_{n-2} + x_{n-2} \ll 2. \tag{2.10}$$

Realizing Equation 2.10 costs LO = 6 adders.

2.3.4 Mixed Common Subexpression Elimination

It is possible to find more than one type of patterns in the Hartley's table. For example, HCSs+VCSs, HCSs+OCSs are possible patterns. The CSE method that work on mixed pattern combinations is called mixed common subexpression elimination

	2^4	2^3	2^2	2^1	2^0
$w_0 = 21$	1	0	1	0	1
$w_1 = 11$	1	0	$\bar{1}$	0	$\bar{1}$
$w_2 = 5$	0	0	1	0	1

Figure 2.12: Finding oblique common subexpressions (OCSs) in the CSD representations of the coefficients $\{5, 11, 21\}$.

(MCSE). Samadi and Ahmadi [2007] used an identification graph to contain the information about all the horizontal and vertical subexpressions. After constructing the identification graph, the problem of finding the best common subexpression is represented using a Hamiltonian Walk. A genetic algorithm is proposed to find an optimal Hamiltonian Walk that leads to the maximum number of subexpression elimination. The concept of forming super-subexpressions (SSs) in Hartley's table is proposed by Vinod and Lia [2005]. The SSs are formed from three and four nonzero digits usually found in the CSD representations. A similar technique is used by Mahesh and Vinod [2008] but with the coefficients are represented in binary. The authors stated that the reason for using binary is that most of the non zero bits form subexpressions of two bits. Therefore, it will leave a smaller number of unpaired bits as compared with CSD. Examples of the MCSE methods are found in [Takahashi et al., 2004, 2008; Kato et al., 2009; Vinod et al., 2010]. Vinod et al. [2003] proposed a MCSE method that first searches for the horizontal common subexpressions (HCSs) in the CSD representations, then searches for the vertical common subexpressions (VCSs) by examining the residual digits in the Hartley's table.

To illustrate the MCSE method, consider the example shown in Figure 2.13. The horizontal pattern $10\bar{1}$ and the vertical 11 are identified in this table. The equation for the horizontal common pattern is given by

$$s_n = x_n - x_n \ll 2, \tag{2.11}$$

and for the vertical common pattern is given by

$$t_n = x_n + x_{n-1}. \tag{2.12}$$

The output equation is reduced to

$$y_n = s_n \ll 1 + s_{n-2} \ll 2 + t_n + t_n \ll 4. \tag{2.13}$$

The number of adders is 5. However, it becomes 6 adders when using only one type of pattern.

	2^4	2^3	2^2	2^1	2^0
$w_0 = 11$	1	$\bar{1}$	0	1	1
$w_1 = 17$	1	0	0	0	1
$w_2 = -12$	$\bar{1}$	0	1	0	0

Figure 2.13: An example of the MCSE.

2.4 MULTIPLIER BLOCK COMMON SUBEXPRESSION ELIMINATION

The MB common subexpression elimination methods work only on the transposed structure of the FIR filter. The delay line should be transposed before applying the optimization procedure. These methods have mixed features from the CSE and GD methods because they start with an optimizing phase similar to that in the GD but the CSE methods can also be used to constrain the logic depth. This makes this type of CSE methods the most powerful. There are two distinct types of MB algorithm (Figure 2.8), the representation independent (RI) methods and the horizontal common subexpression elimination works on multiplier block (HCSE-M).

2.4.1 Representation Independent Common Subexpression Elimination

Representation independent (RI) methods solve the MCM problem irrespective of the coefficients' representations. They try to find new coefficients that satisfy the frequency response specifications of the designed filter. Frequency response specifications may have some allowable tolerance that can provide some degree of freedom to find alternate values for the coefficients. In other words, it is possible to find a smaller coefficient set than the original one, without violating the frequency domain specifications. For example, Shi and Yu [2011] proposed an algorithm to find the optimum filter coefficients. The algorithm uses mixed integer linear programming (MILP) to traverse the integer space. The authors mentioned that increasing the wordlength for some filters could result in a runtime of several days. Yao and Sha [2010] proposed an algorithm that constructs the binary tree to find a cost-effective solution. The algorithm uses an expanded subexpression search to find the discrete coefficients. A branch and bound algorithm (B&B) [Walukiewicz, 1991] is used to limit the size of the tree as the tree grows exponentially with the filter length. However, the logic depth is unconstrained in this algorithm. A two stage FIR filter is used in [Shi and Yu, 2010] to reduce the adder depth. The two subfilters are optimized simultaneously using a MILP search. All the possible filters are explored to find a cascade design with minimum number of adders. The method gives better results in terms of the total number of adders, adder depth, and effective wordlength when compared with [Gustafsson and Wanhammar,

2002]. In reference [Yu et al., 2008b], the coefficients are approximated by extrapolating the impulse response of the FIR filter without constraining the adder depth. This reduces the dynamic range of the coefficients and results in coefficients with identical values. Residual compensation is proposed to preserve filter performance by restoring the coefficient values to the original optimum ones.

To illustrate the RI methods, consider the algorithm in [Yu et al., 2008b]. A MILP method is used to minimize the normalized peak ripple of the filter frequency response. The optimization method results in an optimum coefficient set of real values. The algorithm uses a predefined subexpression set (for example $\{0, \pm 1, \pm 3, \pm 5\}$) to approximate the real coefficients to integers. Only combinations of at most k subexpressions are used to compose the coefficients. A B&B algorithm (Figure 2.14) is used to minimize the number of subexpressions required to construct the coefficients. The algorithm starts with the optimum continuous (real) coefficients $P_{0,s}$. The coefficients are selected in succession such that each time one of them is picked to branch on it. For example, in Figure 2.14 the first picked coefficient is x_3. Two subproblems P_1 and P_2 are created by imposing the bounds $x_3 \leq \lfloor x_3 \rfloor$ and $x_3 \geq \lceil x_3 \rceil$, respectively. Let $x_3 = 7.4$ and $k = 2$, then the closest integer to x_3 is obtained from summing the subexpressions 3 and 5, i.e. 8. This integer is greater than x_3, therefore the algorithm considers this integer as the ceiling value of x_3 and searches for the floor integer in the space which is 7. Since the algorithm branches on the floor value, the ceiling node is stored as an unsolved problem. The algorithm tries to find the closest integer to 7 using exhaustive search for the subexpression combinations. In this case, the closest floor is 6. Another continuous optimum solution is found by solving for this floor constraint. When the algorithm reaches a node where all the coefficients are integers, this node is fathomed. A fathomed node is defined as the one that stores the integer solution and there is no profit from searching beyond this node. Fathomed nodes are shown in bold in Figure 2.14. The algorithm backtracks to the previous depths to solve each stored problem. Branching and backtracking continues until fathoming all the nodes. The integer solution with minimum objective function is selected.

2.4.2 Multiplier Block Horizontal Common Subexpression Elimination

Multiplier block HCSE methods search the MB for the HCSs. Hence, they are denoted by HCSE-M. For example, Kuo et al. [2011] proposed a HCSE-M algorithm in which the process of filter coefficients quantization is combined with the CSE. A new complexity-aware allocation of non-zero terms is proposed to take into account the CSE. Comparison results showed that the proposed algorithm results in more adders than RAG-N [Dempster and Macleod, 1995] but the former has a smaller area complexity. Ho et al. [2008] used a 0-1 MILP method to tackle the MCM. The method extends the MHW space with shifted sums or differences (SSD) of the coefficients. Then it tries to synthesize the coefficients using a minimum number of subexpressions. Adder depth is used as a constraint at the top of the problem. The drawback of this technique is the computational complexity which increases exponentially with the wordlength and the

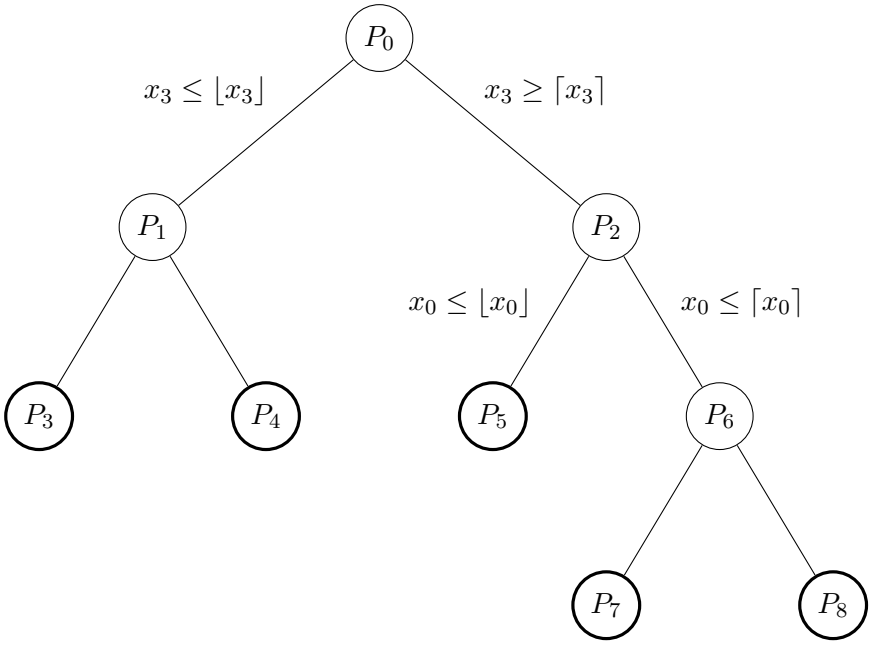


Figure 2.14: An example of a branch and bound tree.

number of variables. To explain the concept of SSD, consider the filter in Equation 2.1 with coefficient representations shown in Table 2.2. The representations that are enclosed by rectangles are chosen to illustrate the concept. The value 5 is synthesized from its representation as

$$00101 = 1 \ll 2 + 1 = 5.$$

The SSD of the subexpressions 1 and 5 synthesize the coefficients 11 and 21 from their representations as

$$10\bar{1}0\bar{1} = 1 \ll 4 - 5 = 11,$$
$$10101 = 5 \ll 4 + 1 = 21.$$

This simple example results the realization directly as shown in Figure 2.15 without need for using the MILP.

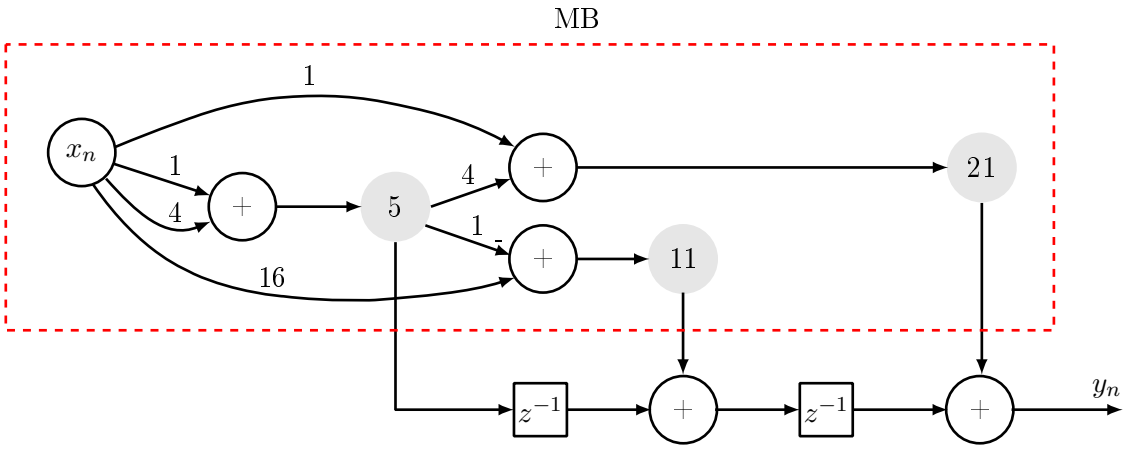


Figure 2.15: Filter coefficients $\{21, 11, 5\}$ are synthesized using the method proposed by Ho et al. [2008].

Another example of the HCSE-M methods is the algorithm proposed by Yao et al. [2004]. In this algorithm, filter coefficients are represented using CSD representation. A subexpression set is constructed from the coefficients' CSD representations. The coefficients and subexpressions are classified according to their adder step. The filter adder step is used as a constraint in the algorithm. The algorithm contains three loops that iterate over the coefficients from the highest adder step coefficients to the lowest. Each coefficient is synthesized by using no more than two subexpressions to minimize the adder step. In the first loop, the algorithm tries to compose each coefficient from the synthesized subexpressions and common subexpressions (zero cost subexpressions). In the second loop, residual coefficients are synthesized as follow: One part is from the zero cost subexpressions, while the other part is from the CSD subexpression set. Subexpressions that require minimum additional adders are preferred. Then a minimum set of the most common subexpressions is chosen. In the third loop, the two parts of each coefficient are taken from the CSD subexpression set. These candidates subexpressions are searched for the minimum set. The algorithm is summarized as follows

1. Construct the subexpression set N_h from the CSD representations of the coefficients that exist in the set H_n;
2. Classify all numbers in H_n and N_h according to their minimum adder step MAS;
3. Let $H_p = \emptyset$ be the set of synthesized coefficients.
4. Loop while $H_n \neq \emptyset$:
   (a) Loop1: Iterate over H_n's partitions starting with the one with the highest MAS;
      i. For each coefficient h_n in H_n;
      ii. Try to decompose h_n using only two parts, $h_n = \pm 2^i p_1 \pm 2^j p_2$, where $p_1, p_2 \in H_p \bigcup H_n \bigcup \{1\}$.
   (b) Loop2: Iterate over H_n's partitions starting with the one with highest MAS;
      i. For each coefficient h_n in H_n;
      ii. Try to decompose h_n using only two parts, $h_n = \pm 2^i p_1 \pm 2^j p_2$, where $p_1 \in H_p \bigcup H_n \bigcup \{1\}$ and $p_2 \in N_h$;
      iii. Find $S_{\min}$ = the smallest set of the most common p_2's that result in minimum additional circuitry.
   (c) Loop3: Iterate over H_n's partitions starting with one with the highest MAS;
      i. For each coefficient h_n in H_n;
      ii. Try to decompose h_n using only two parts, $h_n = \pm 2^i p_1 \pm 2^j p_2$, where $p_1, p_2 \in N_h$;
      iii. Find $S_{\min}$ = the smallest set of the most common p_1's and p_2's that result in minimum additional circuitry.

The method in [Yao et al., 2004] may lack the optimal solution if some of the subexpressions are not in the CSD representations of the coefficients. For example,

the algorithm fails in finding the optimal solution for the filter $\{105, 411, 815, 831\}$ when the required filter adder depth is 3 as shown in Figure 2.16. In this case, the subexpression 129 requires one extra adder to be synthesized making the filter cost 7 adders as compared with 6 adders in the optimal realization shown in Figure 2.17.

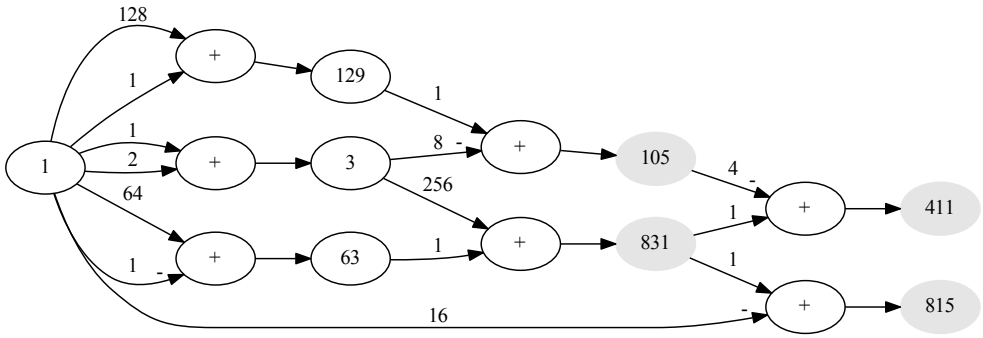


Figure 2.16: Yao algorithm realization for the filter $\{105, 411, 815, 831\}$ with filter adder step constraint of 3 adders.

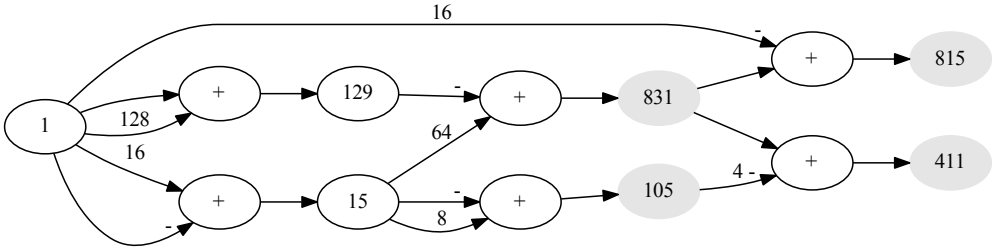


Figure 2.17: The optimal realization for the filter $\{105, 411, 815, 831\}$ with filter adder step constraint of 3 adders.

2.5 CHAPTER SUMMARY

A literature review of the GD and CSE methods is presented in this chapter. The GD method uses an acyclic directed graph to optimize the MCM. Optimal realization is obtained when the graph set is the coefficient set and there is no need to compute extra partial products. In other words, the method tries with a small space of fundamentals and carefully adds more partial products with a minimum distance from the graph. This limits the search space and the method does not explicitly optimize the logic depth.

A classification of the CSE methods is introduced in this chapter. Two main categories of the CSE methods are identified in this classification which are the Hartley's table method and the MB method. Hartley's method tabulates coefficient representations in $N \times L$ table, where N is the number of coefficients and L is the wordlength. Four types of the subexpression patterns are found in Hartley's table which are the HCSE, VCSE, OCSE, and MCSE. The HCSE is preferred since the other methods could destroy the symmetry of the FIR filter coefficients. The method is explored in the next chapter to examine the effect of using larger set of the redundant representations than the MHW or CSD.

Multiplier block methods work on the transposed structure of the FIR filter. Therefore, these methods of CSE combine the advantages of the GD and Hartley's methods. Optimization algorithms such as branch and bound (B&B) and integer programming can be used to search for best subexpressions sharing. The MB algorithms are able to constrain the logic depth of the block by putting it as a constraint at the top of the MCM problem. Therefore, these methods can search for solutions with minimum logic operators simultaneously with minimizing the logic depth.

Chapter 3

PATTERN PRESERVATION ALGORITHM

We showed in the previous chapter that the performance of the common subexpression elimination (CSE) methods is effected by the number representations. Therefore it is difficult to separate the CSE algorithm from number representations. Furthermore, making the algorithm calculates all the possible patterns and clusters of digits would boost its ability to detect and resolve hard overlapping patterns. Doing this may be infeasible when the search space is the whole binary signed digit (BSD) representations and an exhaustive search is required to find a multiple constant multiplication (MCM) with minimum complexity. The search space should be reduced to develop a polynomial time algorithm. On the other hand, reducing the search space should not be at the expense of losing a valuable subexpressions that could result in optimal solution equals to the lower bound as described in Section 1.8, otherwise the lower bound would be increased resulting in a sub-optimal solution. Therefore, choosing a proper search space of number representations could improve the CSE algorithm performance and find the solution in polynomial time. One of the redundant representations that could satisfy this compromise is the zero-dominant set (ZDS). This set is proposed by Kamp [2010] for hardware optimization in DSP systems. It is slightly larger than the MHW but this difference is of significant advantage for CSE because it is related to the relative position of the zero digit among the representations. This increases the number of common patterns that can be found in the representations. The effect of using the ZDS in the CSE is investigated in this chapter.

The chapter is structured as follows: Section 3.1 considers generating the ZDS. A comparison between the ZDS and canonical signed digit (CSD) representation is made in Section 3.2. Pattern preservation algorithm (PPA) is proposed in Section 3.3 to search the ZDS. The results of comparing the PPA with other works is shown in Section 3.5. A chapter summary is presented in Section 3.6.

3.1 ZERO-DOMINANT SET GENERATION

The ZDS was proposed by Kamp [2010] to provide near optimum packing in the partial product packing problem. Partial product packing is used to speed up the accumulation of the partial products in symmetric FIR filters. It is different from the CSE which is used to minimize the LO and LD by eliminating the common subexpressions among coefficients' representations. The ZDS algorithm uses the concept of Pareto-

optimality [Kamp, 2010]. It gives a score of 1 for each non-zero digit position in the representation and score of 0 if the digit is zero. The algorithm is based on the hypothesis that there is no other representation where the score of a position can be reduced (improved) without increasing the score of other positions [Kamp, 2010]. In this case, a representation **A** is said to be zero-dominate (Pareto-improve) representation **B** when both:

1. **A** has a zero digit in every position that **B** has a zero digit and;
2. **A** has a lower cumulative score than **B**.

This implies that each zero-dominant representation has at least one zero digit in a position that any other zero-dominant representation has a non-zero digit. An example of dominant relation is shown in Table 3.1 for the value 115. To explain the dominance relation between two representations, consider the first row in Table 3.1. Representation **A** satisfies condition 1 above because it has zero digit at every position **B** has zero digit. These digits are shown in red color in the first row of Table 3.1. Condition 2 is satisfied too because representation **A** has at least one zero digit (shown in blue) in a place that **B** does not. The cumulative score of **A** equals 4, while **B** score equals 5 which result in making **A** dominates **B**. A same conclusion is obtained for the rest of Table 3.1. Therefore, the ZDS of 115 is the five representations of **A** which are listed in the first column of Table 3.1.

Table 3.1: Dominance relations for the representations of the value 115.

A	dominant relation	**B**
$(1,0,0,0,\bar{1},\bar{1},\boxed{0},\bar{1})$	dominates	$(1,0,0,0,\bar{1},\bar{1},\bar{1},1)$
$(1,0,0,\bar{1},0,1,0,\bar{1})$	dominates	$(1,0,0,\bar{1},1,\bar{1},0,\bar{1})$
$(1,0,0,\bar{1},0,0,1,1)$	dominates	$(1,0,\bar{1},1,0,0,1,1)$
$(0,1,1,1,0,1,0,\bar{1})$	dominates	$(0,1,1,1,0,1,\bar{1},1)$
$(0,1,1,1,0,0,1,1)$	dominates	$(1,\bar{1},1,1,0,0,1,1)$

A branch and bound technique is used to generate the ZDS by checking for dominance while generating the representation tree (shown in the next chapter). Nodes which lead to non-dominant representations can be removed before traversing all the branches. This prevents generating and searching all of the representations for dominance, except those with unique zero-digit positions and with minimized Hamming weight [Kamp, 2010]. Figure 3.1 depicts the relation between CSD, MHW, ZDS, and BSD representations using Venn diagram.

Putting importance on the zero positions increase the possibility of obtaining representations with a higher frequency of common patterns and digits than that found in the CSD and MHW. Therefore, using the ZDS could improve the performance of the CSE algorithm. It also could prevent generating and searching all of the BSD representations. For example, the number of ZDS representations for the value 115 equals to

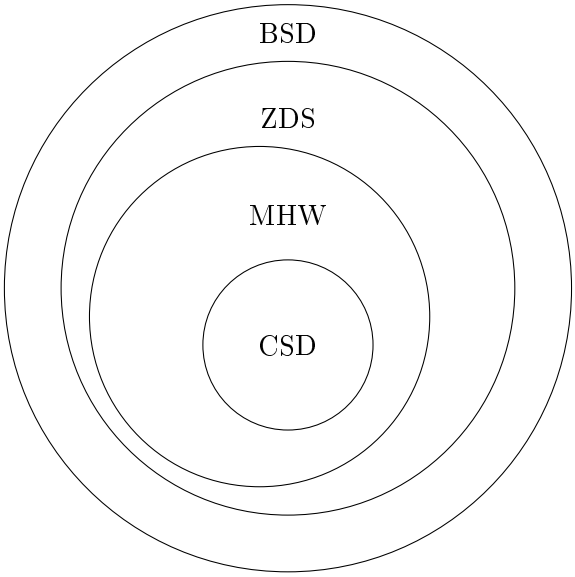


Figure 3.1: Venn diagram illustrating the relationship between CSD, MHW, ZDS, and BSD representations.

5 representations as shown in Table 3.1. While the number of BSD representations for the same value and using the same wordlength (in this case the wordlength is $L = 8$) equals to 22 representations.

In general, the number of BSD representations with a digit set $\mathbb{D}$ and wordlength L equals $|\mathbb{D}|^L$, where $|\mathbb{D}|$ is the digit set cardinality (the number of elements in the set). However, the average number of representations of a value can be estimated as,

$$\Gamma = \lceil (|\mathbb{D}|/r)^L \rceil \ , \tag{3.1}$$

where r is the radix [Kamp, 2010]. For example, if $|\mathbb{D}| = 3$, $L = 16$, and $r = 2$ then $\Gamma = 657$.

3.2 ZERO-DOMINANT SET VERSUS CANONICAL SIGNED DIGIT

In this section we compare the HCSE-H method using ZDS and CSD. A 5th order FIR filter is considered in Equation 3.2 for comparison.

$$y_n = -149x_n - 135x_{n-1} - 41x_{n-2} + 53x_{n-3} + 113x_{n-4}. \tag{3.2}$$

The result of using the CSD to find horizontal common subexpressions (HCSs) is shown in Figure 3.2.

The corresponding filter output y_n of Hartley's table in Figure 3.2 is given by:

$$\begin{aligned} y_n = &- s_n - x_n \ll 4 - x_n \ll 7 + t_{n-1} - x_{n-1} \ll 7 - x_{n-2} \\ &- s_{n-2} \ll 3 + s_{n-3} - x_{n-3} \ll 4 + x_{n-3} \ll 6 \\ &+ x_{n-4} - t_{n-4} \ll 4, \end{aligned} \tag{3.3}$$

where $$s_n = x_n + x_n \ll 2, \text{ and } t_n = x_n - x_n \ll 3. \tag{3.4}$$

In this case, the number of LO and LD are 13 and 3 respectively.

If the ZDS is used to represent the coefficients in Equation 3.2 it results a multi representations for each coefficient as shown in Table 3.2. In this case the number of ZDS combinations that should be searched to find the minimum realization is calculated according to Equation 2.3 and is found equals to $C = n_{-149} \times n_{-135} \times n_{-41} \times n_{53} \times n_{11} = 7 \times 3 \times 4 \times 5 \times 3 = 1260$, where n_{-149} is the number of representations of the coefficient -149 and similarly for the other coefficients.

Coefficient	**# of ZDS representations**
-149	7
-135	3
-41	4
53	5
11	3

Table 3.2: The ZDS of the coefficients $\{-149, -135, -41, 53, 11\}$.

Picking one of the ZDS combinations and tabulating the representations in Hartley's table results in the table shown in Figure 3.3. The output for Figure 3.3 is given by:

$$\begin{aligned} y_n = &t_n + t_n \ll 2 + s_n \ll 5 + s_{n-1} - x_{n-1} \ll 7 \\ &+ t_{n-2} + s_{n-2} \ll 3 + s_{n-3} + t_{n-3} \ll 2 \\ &+ t_{n-4} + x_{n-4} \ll 7, \end{aligned} \tag{3.5}$$

	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
h_0		-1			-1		$\boxed{-1}$		$\boxed{-1}$
h_1		-1				$\boxed{-1}$			$\boxed{1}$
h_2				$\boxed{-1}$		$\boxed{-1}$			-1
h_3			1		-1		$\boxed{1}$		$\boxed{1}$
h_4		$\boxed{1}$			$\boxed{-1}$				1

Figure 3.2: Horizontal common subexpressions among CSD representations of the coefficients, $\{-149, -135, -41, 53, 11\}$.

where

$$s_n = x_n - x_n \ll 3, \text{ and } t_n = -x_n + x_n \ll 4. \tag{3.6}$$

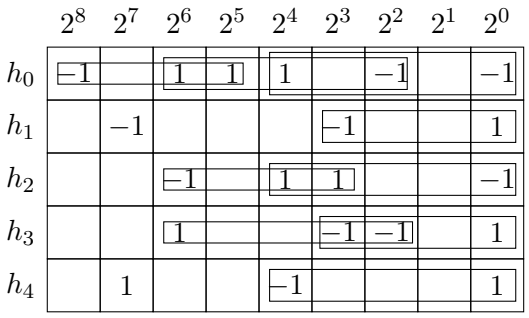


Figure 3.3: Finding the HCSs in one combination of the ZDS representations of the coefficients $\{-149, -135, -41, 53, 11\}$.

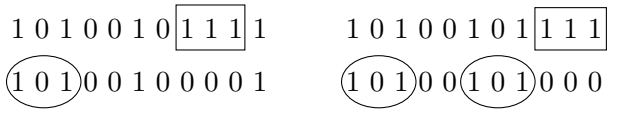
Implementing Equation 3.5 results in LO = 12 and LD = 3. Therefore, using the ZDS reduced the LO by one adder as compared with the CSD while there is no change in the LD value.

The number of BSD combinations in the example of Equation 3.2 is $C = 50 \times 29 \times 40 \times 44 \times 30 = 76,560,000$ as compared with only 1260 for the ZDS. This comparison between the ZDS and BSD is for small filter example. For large filters, a huge space is required to be searched in the case of BSD.

3.3 PATTERN PRESERVATION ALGORITHM

An algorithm for CSE elimination is proposed in this section The algorithm is called the pattern preservation algorithm (PPA). The PPA uses Hartley's table method that described in Subsection 2.3.1 to search for the Horizontal common subexpressions (HCSs). The PPA scans each representation in Hartley's table twice to search for the common subexpressions. In the first pass of the search phase, it searches for the CSD patterns and leave other non CSD patterns for the next pass. Statistics are made about the number of occurrence of each CSD subexpression. In the second pass of the search phase, the algorithm searches for the other forms of subexpression patterns and update the statistics. Once the algorithm is complete the search phase it enter the elimination phase that use the statistics which is built in the search phase. The algorithm use the statics to eliminate the most common subexpression from all the representations. Then it repeats the process for the next most common subexpression until there is no more subexpressions can be eliminated. This elimination phase is implemented using two pass similar to that used in the search phase but with additional feature or mechanism which is a digit set/reset. The algorithm determine if there is a pattern of all non-zero digits clustered at the far end from the search starting point. If this pattern is of value equal the searched subexpression then the digit at the inner border of the pattern is reset. This digit is set again once the second pass of elimination a subexpression is completed.

The PPA is proposed to have two pass because of using the ZDS instead of the CSD introduces two problems. Firstly, the ZDS provides representations that may contain multi patterns for the same value. This requires the algorithm uses a particular search method to find these patterns. Secondly, the ZDS may result in overlapping patterns and eliminating a pattern may result in losing other useful patterns. For example, consider Figure 3.4(a) in which the upper representation is for the coefficient 1327 before starting the search and elimination process for the subexpression 111 (7). Assume that the search goes from the left to the right and it encountered a pattern with a value of 7 (shown by the three ones that enclosed by a square). The algorithm eliminates this pattern as shown in the lower representation of Figure 3.4(a). Suppose that the next subexpression to be searched is 5. The number of occurrences of the value 5 in the lower representation of Figure 3.4(a) equals to 1 which corresponds to the pattern 101 (enclosed by an ellipse). However, there were two patterns of value 5 in the original representation. Loosing one of the patterns 101 is due to the elimination of the pattern 111. To save the second pattern, consider Figure 3.4(b) in which the value 7 is eliminated by zeroing the three ones (enclosed by the square) that clustered at the far end of the representation. In this case, the number of 101 patterns is two as shown by the two enclosed ellipses in the lower representation of Figure 3.4(b).



(a) A case of pattern overlapping results in destroying the pattern 101.

(b) Preserving the pattern 101 using pattern preservation algorithm.

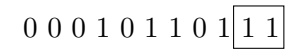
Figure 3.4: Solving the overlapping problem for the far/near end pattern.

The pattern preservation idea shown in Figure 3.4(b) is developed to a pattern preservation algorithm (PPA) described in Figure 3.5. The PPA uses two pass with a digit set-reset mechanism to resolve the overlapping patterns. The first pass searches for the CSD common subexpressions. For example, assume that the algorithm searches for the subexpression 3, then the first pass will search for the patterns $10\overline{1}$ and $\overline{1}01$ as shown in Figure 3.5(a). The result of eliminating the subexpression 3 is shown in Figure 3.5(b). The first pass is finished at this point because no more CSD patterns can be found for 3. In the second pass, the algorithm searches for clusters of non-zero digits as shown in Figure 3.5(c). This cluster is at the far end of the representations. To make the elimination from right to left, which is in reverse of the scan direction, the algorithm calculates the value of the non-zero digits at the far end. The digit at the inner border of the far end pattern is excluded from this calculation to not destroy other possible patterns to the left of this digit. This step is illustrated in Figure 3.5(c) by enclosing the calculated digits by a square. Because the value of these digits equals to the value of the common subexpression 3, the digit at the inner border is reset as shown in Figure 3.5(d). The second pass can start by searching for the patterns {11, $\overline{1}\,\overline{1}$} as shown in Figure 3.5(e). Therefore the common subexpression 3 is eliminated safely as shown in Figure 3.5(f). The value of the inner border digit is recovered at the end of the second pass as shown in Figure 3.5(g). It also shows that the PPA succeed in saving the pattern 100001 = 33 instead of 10000001 = 129. The subexpression 33 is favored over 129 because it results in a shorter adder wordlength than that of 129. The subexpression 129 is obtained when removing the digit set-rest mechanism from the PPA. The resulting algorithm is called the two pass algorithm (TPA) as shown in Figure 3.6. The algorithm chose to eliminate the pattern 11 as shown in Figure 3.6(c) leaving the pattern 10000001 instead of 100001 for the next search step.

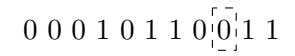
Other forms of the PPA can be obtained when the algorithm is modified to search for only subexpressions that are not CSD like the pattern 11 in the last example. This means the search is limited by the second pass of the PPA, which are the steps shown in Figure 3.6(c) to Figure 3.6(d). In this case, the algorithm is called the single pass algorithm (SPA). A final form of the PPA, is that algorithm which constrains the search for CSD numbers or canonical pattern search algorithm (CSA), implemented in steps Figure 3.6(a) and Figure 3.6(b) only.

(a) Pass1 searches for the subexpression 3 as $(10\overline{1})_2$, $(\overline{1}01)_2$ patterns.

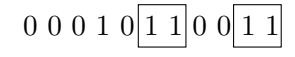
(b) Pass1 eliminates the subexpression 3.



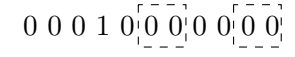
(c) Pass2 calculates the value of the nonzero digits at the far end.



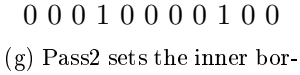
(d) Pass2 resets the inner border digit.



(e) Pass2 searches for the expression 3 as $(11)_2$, $(\overline{11})_2$ patterns.



(f) Pass2 eliminates the subexpression 3.

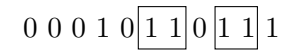


(g) Pass2 sets the inner border digitl before return.

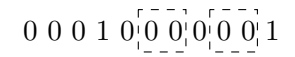
Figure 3.5: An example illustrates the mechanism of the pattern preservation algorithm.

(a) Pass1 searches for the subexpression 3 as $(10\overline{1})_2$, $(\overline{1}01)_2$ patterns.

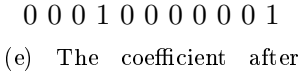
(b) Pass1 eliminates the subexpression 3.



(c) Pass2 searches for the expression 3 as $(11)_2$, $(\overline{11})_2$ patterns.



(d) Pass2 eliminates the subexpression 3.



(e) The coefficient after eliminating the subexpression 3. The subexpression 129 is left instead of 33 that obtained when using the PPA.

Figure 3.6: An example illustrates the mechanism of the two pass algorithm (TPA).

3.4 ALGORITHM COMPLEXITY

The time complexity of the PPA can be estimated as:

$$T_{PPA} = C \times N \times L^2, \tag{3.7}$$

where C is the number of ZDS combinations as defined in Equation 2.3, N is filter order, and L is the wordlength. This estimation is found from the work of the PPA algorithm which used to search C Hartleys' tables each of dimensions $L \times N$. Where N is the number of coefficients of the MCM and L is the wordlength. Squaring the wordlength because the algorithm uses double scan for each row to calculate the subexpressions.

3.5 RESULTS

A comparison is made between the performance of the PPA when it uses minimum Hamming weight representations and when it uses the ZDS as shown in Table 3.3. Three FIR filter examples each with 5 coefficients were used. It was found that using the ZDS results in realizations with smaller LO than using the MHW.

Filter Coefficients	**LO (ZDS)**	**LO (MHW)**
$-149, -135, -41, 53, 113$	12	13
$-1071, -123, 43, 333, 773$	12	13
$-1033, -895, 227, 363, 445$	13	14

Table 3.3: A comparison between the PPA performance using the zero-dominant set (ZDS) and the minimum Hamming weight (MHW) representations.

A second comparison was made between the PPA, TPA, and SPA using the ZDS. The result of the second comparison is shown in Table 3.4. The result shows that the PPA has a better detecting ability than the other algorithms. This is because the algorithm outperforms the TPA and SPA in the case of existing overlapping patterns. In addition, it outperforms the SPA because of the two pass search. The CSA is found to be lagging behind all the other algorithms because its search is limited to the CSD patterns only.

Filter Coefficients	**PPA**	**TPA**	**SPA**	**CSA**
$-149, -135, -41, 53, 113$	12	12	12	13
$-1071, -123, 43, 333, 773$	12	13	13	13
$-1033, -895, 227, 363, 445$	13	13	13	13

Table 3.4: A comparison between the pattern preservation algorithm (PPA), the two pass algorithm (TPA), the single pass algorithm (SPA), and the CSD algorithm (CSA).

A third comparison was made between the PPA using ZDS, N-dimensional reduced graph adder (RAG-N) algorithm [Dempster and Macleod, 1995], Hartley's method us-

ing the CSD [Hartley, 1996], and Hartley's method using the BSD with an extended wordlength by one position [Dempster et al., 2004]. The original comparison was presented in [Dempster et al., 2004]. We added to this comparison the results of our proposed algorithm as shown in Table 3.5.

Two metrics are used in Table 3.5 for comparison which are the adder cost and the filter cost shown in the form adder cost/filter cost. In the case of the graph dependent method, the adder cost means the number of adders required to realize the multiplier block (MB) of the MCM [Dempster and Macleod, 1995]. In Hartley's CSD method [Hartley, 1996] and Hartley's BSD, the adder cost represents the number of terms in Hartley's table. The filter cost in all the cases equals the overall adders required to realize the filter. This is because the number of adders required to compose the common subexpressions in Hartley's CSD and Hartley's BSD tables was not specified in [Dempster et al., 2004]. We calculated this number in the case of Hartley's CSD method from tabulating the CSD representations and search for HCSs. While it is estimated in the case of Hartley's method that work on BSD. The estimation is by substituting the number of filter adders with a range of numbers. The minimum number in this range represents the number of adders that obtained from adding the number of terms in Hartley's table to the number of adders required to construct the common subexpressions in the pattern preservation algorithm. While the maximum range equals to the number of overall adders obtained from applying the algorithm on the ZDS.

Coefficients	**RAG-N**	**Hartley on CSD**	**Hartley on all SD**	**PPA on ZDS**
195, 117, 5	5/7	5/7	5/7	5/7
52, 172, 215	5/7	6/8	4/6-8	5/7
3892, 947, 2486	6/8	9/11	7/9-11	8/10
3832, 3756, 1680	5/7	7/9	5/7-9	7/8
15567, 3787, 9943	9/11	10/12	8/10-12	10/12
7962, 14603, 12486	9/11	10/12	8/10-11	9/11
7479, 303, 13458, 7286	10/13	11/14	9/12-14	11/14
10083, 12975, 15103, 12095	11/14	10/15	9/11-14	12/14
814, 63, 3095, 1823, 3871	8/12	11/15	10/13-15	12/15
3122, 1870, 76, 3364, 1822	9/13	12/15	9/11-12	12/14
569, 831, 814, 2473, 1115	10/14	13/16	$10/\leq 13$	13/16
3892, 947, 2486, 1991, 3651	10/14	12/17	$10/\leq 14$	13/17

Table 3.5: A comparison between: PPA with ZDS, N-dimensional reduced adder graph (RAG-N) method, Hartley's method for CSD, and Hartley's method for BSD, with wordlength extended by one digit.

The result of the comparison shown in Table 3.5 indicates that the PPA has a comparable performance to that of the RAG-N algorithm when using small coefficient sets. But as stated in Chapter 3.2, graph dependent methods may produce realizations with a long critical path [Yao et al., 2004]. The PPA has a better performance than the Hartley's CSD method while this performance lags behind the Hartley's BSD method. This is because the latter searches the whole BSD space while the PPA searches a subset of these representations which is the ZDS.

The algorithm was compared with the difference based adder graph (DBAG) algorithm shown in [Gustafsson, 2007a], and Yao method that given reference [Yao et al., 2004]. Coefficient sets are compared for the wordlenght range from 6 to 15. Each set is of length 5 taps. For each wordlength a hundred filter sets are used for comparison. The result of comparing the average number of adders for the three methods is shown in Figure 3.7. The DBAG shows a superior adder saving than the PPA and Yao methods which is expected form a graph method that try search for minimum realization at the expense of LD. This scarification in the LD is shown in Figure 3.8 where the PPA and Yao have a same minimum LD curve as compared with the DBAG method. The last comparisons between the three methods is in run time as shown in Figure 3.9. The PPA shows worst time complexity than the other two methods because of the number of ZDS representations still large. For example, it is found in the last experiment with wordlength $L = 15$ that the average number of combinations is 25754. This mean searching this number of tables for the optimal subexpression elimination.

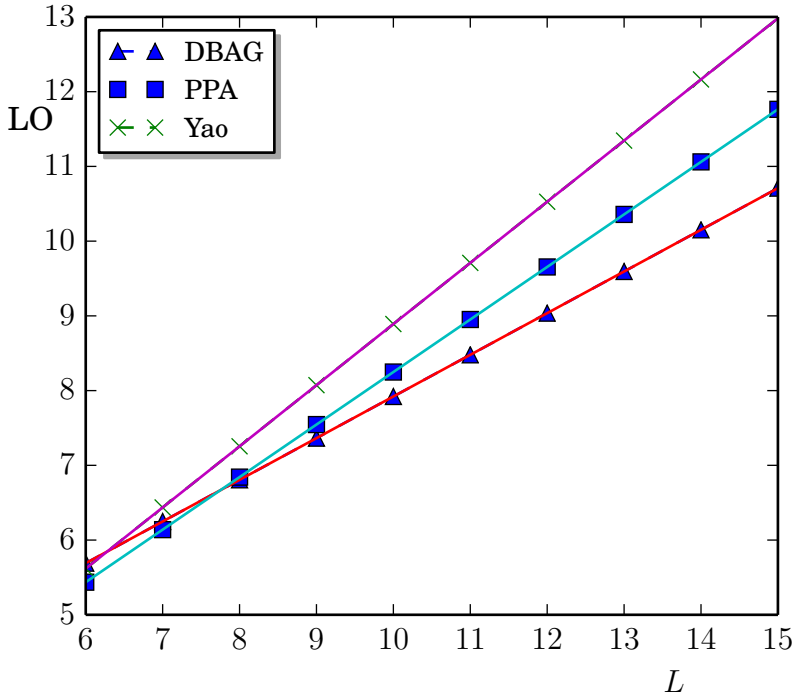


Figure 3.7: A comparison between the number of adders that obtained from using the PPA, DBAG [Gustafsson, 2007a], and Yao [Yao et al., 2004].

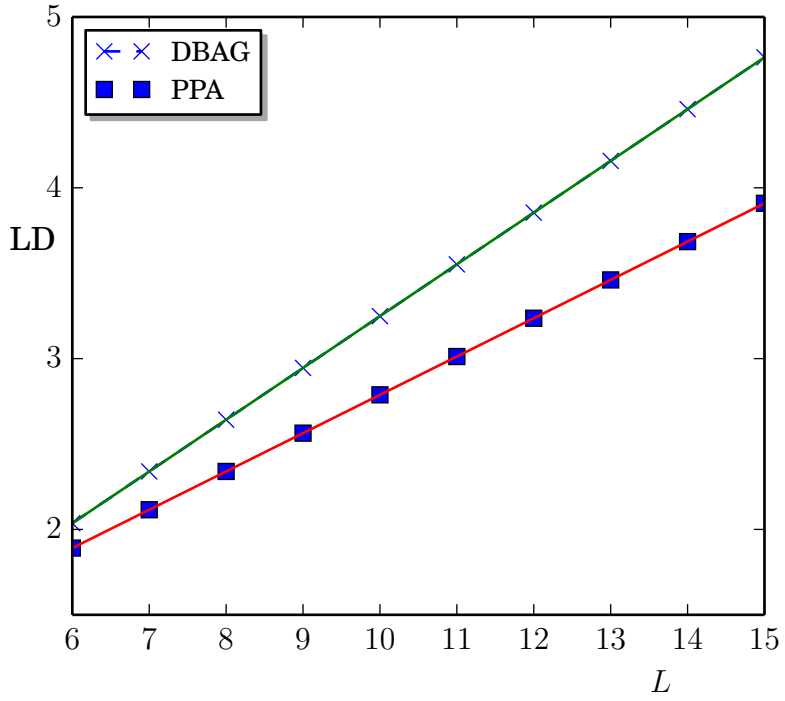


Figure 3.8: A comparison between LD's that obtained from using DBAG [Gustafsson, 2007a], and PPA.

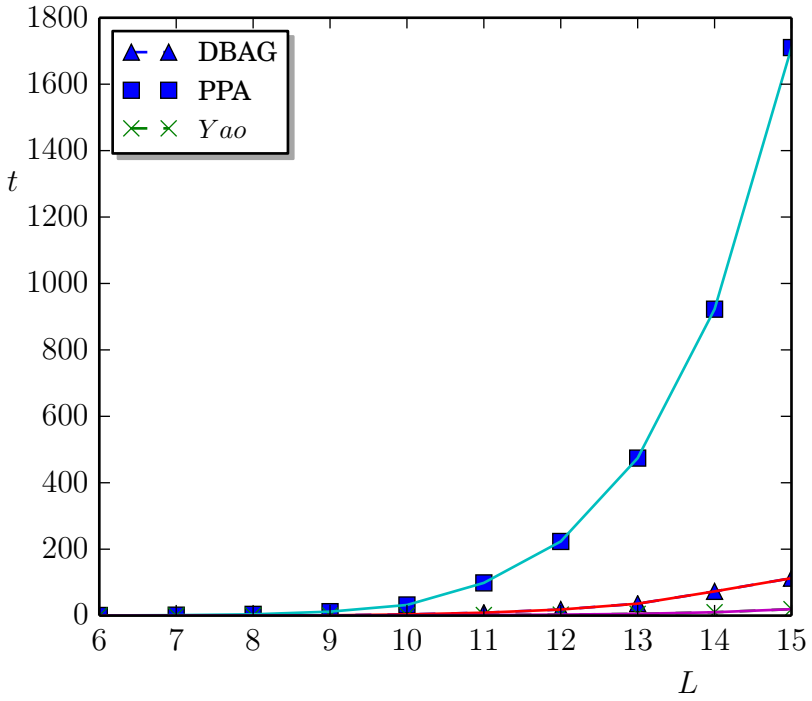


Figure 3.9: The run time comparison between the algorithms PPA, DBAG [Gustafsson, 2007a], and Yao [Yao et al., 2004].

3.6 CHAPTER SUMMARY

Finding minimum realizations for the MCM requires searching representations that are larger than the MHW. A class of redundant representations called the zero-dominant

set is proposed to be used in the CSE method. The set is considered as a compromising between using the MHW and the BSD. Representations from the ZDS are tabulated in Hartley's table to find the horizontal common subexpression between them. Using the ZDS produces a pattern overlapping problem. To prevent losing patterns when searching for other patterns a new algorithm named the pattern preservation algorithm (PPA) is proposed. The proposed algorithm preserves the patterns from being lost by using two pass search with digit set-reset mechanism. Comparison results show that using the ZDS results in fewer adders than using the MHW or CSD. The ZDS can provide a compromise between using the BSD representations and the MHW or CSD but this is true for some extend. In other words, reducing the BSD space still causes the loss of minimum solutions for the MCM. Though using the ZDS reduces the search space to be less than the BSD, the search space is still large.

Chapter 4

REDUNDANT NUMBERS

This chapter covers a number of important properties about numbers and their representations, which will be used in the subsequent chapters. Number and numeral systems are not a new invention. The earliest human methods of counting and enumerating were by using similar objects such as their fingers, pebbles in piles or rows, notches on wooden sticks or bones. The early positional number system was used to group a number of the same objects and substitute them with a single different object. In other words these objects have different weight according to their position in the number system [Kornerup and Matula, 2010].

Additive systems and hybrid systems were developed by the earliest human civilizations [Ifrah, 2001]. Examples of such number systems are the Egyptian hieroglyphic system, the Roman numerals, and the Greek alphabetic number system. In the additive systems, a combination of symbols represent the number and the value is found by summing all the symbols irrespective of their position. Hybrid numeral systems differ from the additive systems in the use of the fundamentals of addition and multiplication. An example of a hybrid number system is the Chinese number system. Both the additive number systems and the hybrid systems have drawbacks which make them difficult to use because they are static (unsuitable for arithmetic), require a lot of space to represent big numbers, and have no clear notation for the numbers. All these drawbacks have been solved by using positional number systems.

In positional number systems, the weight of each symbol depends on the position of that symbol in the number representation. Neighboring position weights need to not be related by a constant ratio. If the ratio between adjacent symbol position is constant then the corresponding positional number system is called a radix system.The oldest example of such system is that used by the Babylonians who used radix 60 for astronomical calculations [Kornerup and Matula, 2010]. The decimal numbers with a fixed radix point was used in India about 600 CE. After this, many radix positional number systems have been developed.

This chapter considers radix positional number systems as they are used in computer arithmetic. A radix number system is defined by two things, the radix and the digit set. When the number of digits in the digit set is equal to the radix, each numerical value has only one representation and the corresponding number system is called nonredundant. If the number of digits in the digit set is allowed to be greater than

the radix value, the corresponding number system is called a redundant number system (redundant non-positional number system is also possible). In redundant number systems, each numerical value has one or more representation. This redundancy is useful in implementing carry free addition. It also provides a higher degree of freedom than nonredundant systems in providing representations that are used to simplify the arithmetic circuit.

This chapter is structured as follows: A classification of positional number systems is presented in Section 4.1. In Section 4.2 the ring of polynomials is developed. Radix polynomials are presented in Section 4.3. Making the radix polynomial to be over a digit set is given in Section 4.4. The necessary and sufficient conditions for a digit set to be a complete residue system modulo radix are given in Section 4.5.

4.1 CLASSIFICATION OF RADIX NUMBER SYSTEMS

Different number systems can be obtained by changing the digit set and/or radix. Choosing the number system that optimizes the algorithm and the technology is not a trivial task. For example, if the critical path of the MCM is determined by the carry propagation, using redundant number arithmetic can provide a carry propagation free addition. In addition, redundancy may reveal some hidden relationships between the subexpressions which are difficult to discover using nonredundant number systems.

Parhami [1990] presented a classification of positional number systems as shown in Figure 4.1. He introduced the generalized signed-digit (GSD) representation which is a more general class of redundant number systems. This class includes number systems with symmetric and asymmetric digit sets. The symmetric digit set has both the digits d and $-d$. An example of the symmetric digit set is the ordinary signed digit (OSD) number system [Avizienis, 1961] and BSD. An example of asymmetric digit set is the binary carry-save with $r = 2$ and digit set $\mathbb{D} = \{0, 1, 2\}$. However, only regular digit sets are considered in the classification in [Parhami, 1990]. Regular digit sets have consecutively valued digits (ignoring the digits that are multiple of the radix [Kamp, 2010]). For example the digit set $\mathbb{D} = \{\overline{1}, 0, 3\}$ radix 2 is regular. Kamp [2010] introduced an algorithm to generate redundant representations for irregular number systems. An example of an irregular number system is the one with $\mathbb{D} = \{\overline{1}, 0, 5\}$ and radix 2. The irregular digit set class is added here to the classification of Parhami [1990] as shown in Figure 4.1.

The redundancy in number systems is measured relative to the non-redundant case. For example, consider the number system with radix r and regular digit set $\mathbb{D} = \{d_n, d_n + 1, \cdots, -1, 0, 1, \cdots, d_p - 1, d_p\}$. If $d_n = -\alpha$ is the most negative digit and $d_p = \beta$ is the most positive, then the redundancy index ρ of the positional number system is given by,

$$\rho = \alpha + \beta + 1 - r. \tag{4.1}$$

For example, a non-redundant conventional number system is one with $\alpha = 0$ and $\beta = r - 1$, i.e., $|\mathbb{D}| = r$. On the other hand, a redundant number system is with $\alpha + \beta + 1 > r$. An example of the redundant number systems is the OSD number

system [Avizienis, 1961; Parhami, 1990]. The redundancy range of the OSD number system is given by,

$$\text{Minimum redundancy} : \alpha = \beta = \lfloor \frac{1}{2} r \rfloor + 1,\ 2 \leq \rho < r,$$

$$\text{Maximum redundancy} : \alpha = \beta = r - 1,\ \rho = r - 1.$$

More examples of redundant number systems are shown in Table 4.1.

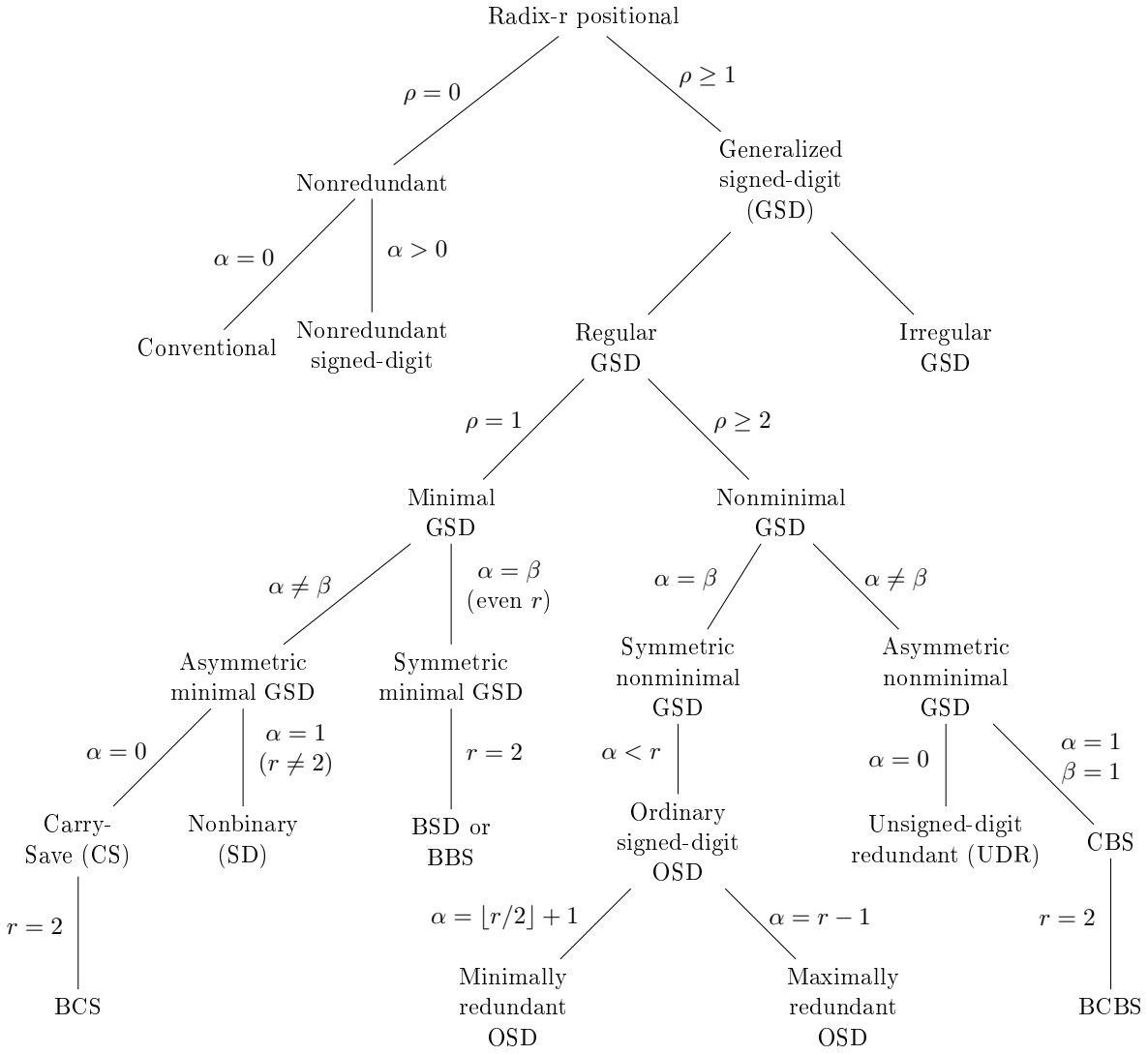


Figure 4.1: Classification of positional number systems [Parhami, 1990].

Number System	Parameters	Example
Binary Carry-Save (BCS)	$r = 2, A = \{0, 1, 2\}, \rho = 1$	$(122102)_2$
Binary Carry-or-Borrow-Save (BCBS)	$r = 2, A = \{\overline{1}, 0, 1, 2\}, \rho = 2$	$(\overline{1}2\overline{1}102)_2$
BSD or Binary Borrow-Save (BBS)	$r = 2, A = \{\overline{1}, 0, 1\}, \rho = 1$	$(10\overline{1}11\overline{1})_2$
Radix-r Carry-Save (CS)	$A = \{0, 1, \cdots, r\}, \rho = 1$	$(130232)_3$
Radix-r Borrow-Save (BS)	$A = \{\overline{1}, 0, 1, \cdots, r-1\}, \rho = 1$	$(122\overline{1}0\overline{1})_3$
Radix-r Carry-or-Borrow-Save (CBS)	$A = \{\overline{1}, 0, 1, \cdots, r\}, \rho = 2$	$(132\overline{1}0\overline{1})_3$
Minimally Redundant ODS	$r = 5, A = \{\overline{3}, \overline{2}, \overline{1}, 0, 1, 2, 3\}, \rho = 2$	$(\overline{21}0\overline{3}21)_5$
Maximally Redundant ODS	$r = 5, A = \{\overline{4}, \overline{3}, \overline{2}, \overline{1}, 0, 1, 2, 3, 4\}, \rho = 4$	$(\overline{21}0\overline{4}21)_5$
Irregular Redundant Number Systems	$r = 2, A = \{\overline{1}, 0, 1, 5\}, \rho = 2$	$(\overline{1}1050\overline{1})_2$

Table 4.1: Examples of redundant number system classes.

4.2 POLYNOMIAL RING

Positional number systems can be defined in terms of the theory of polynomial rings. A polynomial ring over a ring $\mathcal{R}$ (described in Section C.3) is denoted by $\mathcal{R}[x]$ and its elements have the form

$$P(x) = d_m x^m + d_{m-1} x^{m-1} + \cdots + d_0, \tag{4.2}$$

where $P(x)$ is a polynomial in x, $d_i \in \mathcal{R}$ are the coefficients of the polynomial, and x is an indeterminate. The ring $\mathcal{R}$ is a subring (Section C.3) of the ring $\mathcal{R}[x]$ [Hungerford, 1997]. If the polynomials are of the form

$$P(x) = d_m x^m + d_{m-1} x^{m-1} + \cdots + d_i x^i + \cdots + d_l x^l, \tag{4.3}$$

where $-\infty < l \leq i \leq m < \infty$, then $P(x)$ is an extended polynomial. The set that contains the extended polynomials is defined by $\mathcal{R}^*[x]$ which forms an extended polynomial ring.

Adding or multiplying two extended polynomials produces another extended polynomial. This can be shown by considering the following polynomials

$$P(x) = \quad d_m x^m + d_{m-1} x^{m-1} + \cdots + d_l x^l, \tag{4.4}$$

$$Q(x) = \quad b_n x^n + b_{n-1} x^{n-1} + \cdots + b_k x^k. \tag{4.5}$$

Adding the two polynomials results

$$P(x) + Q(x) = (d_s + b_s) x^s + (d_{s-1} + b_{s-1}) x^{s-1} + \cdots + (d_t + b_t) x^t, \tag{4.6}$$

where $s = \max(m, n)$ and $t = \min(l, k)$. Their multiplication is

$$P(x) \times Q(x) = c_{n+m} x^{n+m} + c_{n+m-1} x^{n+m-1} + \cdots + b_{l+k} x^{l+k}, \tag{4.7}$$

where

$$c_j = d_l b_{j-l} + d_{l+1} b_{j-l-1} + \cdots + d_{j-k} b_k. \tag{4.8}$$

Since the coefficients in Equation 4.6 and Equation 4.7 are in $\mathcal{R}$ then the set of extended polynomials is closed under the two operations of addition ($+$) and multiplication ($\times$). It is easy now to prove that if $\mathcal{R}$ is an integral domain (Section C.3), then so is $\mathcal{R}^*[x]$. A ring $\mathcal{R}$ is considered an integral domain if and only if it has the following cancellation property:

$$\forall d \in \mathcal{R}, d \neq 0_R, \text{and } db = dc \in \mathcal{R}, \text{then } b = c.$$

4.3 RADIX POLYNOMIALS

The focus of this thesis is on the ring of integer numbers $\mathbb{Z}$. The ring of extended polynomials in this case is denoted by $\mathbb{Z}^*[x]$. If the indeterminate variable x is fixed to some integer value r, where $|r| > 1$, the resulting extended polynomial may be used as a representation of real numbers. The fixed value r is called the base or radix and the coefficients of the polynomial are the symbols or the digits. Each symbol in the representation has a weight equal to some power of the radix. The resulting polynomial is called the extended radix polynomial and is given by

$$P([r]) = d_m [r]^m + d_{m-1} [r]^{m-1} + \cdots + d_l [r]^l, \tag{4.9}$$

where the notation $[r]$ means this is undetermined, $d_m, d_{m-1}, \cdots, d_l \in \mathbb{Z}$ (the set of integer numbers) and $-\infty < l \leq m < \infty$. The set of radix-$r$ polynomials is given by

$$\mathcal{P}[r] = \{P \;\Big|\; P = P([r])\}, \tag{4.10}$$

where the set $\mathcal{P}[r]$ includes the zero element of the ring $\mathbb{Z}^*[x]$.

The expression in Equation 4.9 is an unevaluated expression. Evaluating the polynomial at particular value of r determines the real value that the polynomial represents. The evaluation mapping is given by:

$$||P([r])|| = P(x)|_{x=r} = P(r) = \sum_{i=l}^{m} d_i r^i. \tag{4.11}$$

Since r and d_i are integers, the value $v = \sum_{i=l}^{m} d_i r^i$ calculated by Equation 4.11 is a rational number [Kornerup and Matula, 2010] and belongs to

$$\mathbb{Q}_r = \{ir^j \;\Big|\; i, j \in \mathbb{Z}\}, \tag{4.12}$$

where $\mathbb{Q}_r \in \mathbb{Q}$ is the set of radix-r numbers and $\mathbb{Q}$ is the set of rational numbers. Examples of these numbers are the set of the binary numbers $\mathbb{Q}_2$, the ternary numbers $\mathbb{Q}_3$, the octal numbers $\mathbb{Q}_8$, and the hexadecimal numbers $\mathbb{Q}_{16}$. Thus, the arithmetic in the ring $\mathbb{Q}_r$ corresponds to the arithmetic in the ring $\mathcal{P}[r]$.

The evaluation mapping ||.|| in Equation 4.11 is a homomorphism (Section C.4) from $\mathcal{P}[r]$ to $\mathbb{Q}_r$. This partitions $\mathcal{P}[r]$ into residue classes. The elements of each class are polynomials evaluated to the same value. The residue class whose elements are evaluated to v is given by the definition:

$$\mathcal{V}_r(v) = \{P(x) \in \mathbb{Z}^*[x] \mid P(x)|_{x=r} = v\}. \tag{4.13}$$

To prove that any polynomial in Equation 4.13 is evaluated to the same value v for $x = r$, let $P(x)$ and $Q(x)$ be polynomials in the same class $\mathcal{V}_r(v)$. Since the two polynomials are in the same residue class, they are congruent to each other according to the relation $P(x) \equiv Q(x) \bmod (x - r)$ [Kornerup and Matula, 2010]. This can also be written as

$$P(x) = Q(x) + (x - r). \tag{4.14}$$

Evaluating Equation 4.14 at $x = r$ results in

$$||P(r)|| = ||Q(r)||, \tag{4.15}$$

which proves that the polynomials in Equation 4.13 are evaluated to the same value for $x = r$. The set $\mathcal{V}_r(v)$ is also defined by the set of redundant representations of v. For example consider the following unevaluated polynomials:

$$\begin{aligned} P[r] &= [r]^4 + [r] + 1, \\ Q[r] &= [r]^4 + [r]^2 - 1, \\ F[r] &= [r]^4 + [r]^3 - [r]^2 - 1. \end{aligned}$$

Evaluating the polynomials at $r = 2$ yields:

$$\begin{aligned} P(2) &= 2^4 + 2 + 1 = 19, \\ Q(2) &= 2^4 + 2^2 - 1 = 19, \\ F(2) &= 2^4 + 2^3 - 2^2 - 1 = 19. \end{aligned}$$

That is, $P[2]$, $Q[2]$, and $F[2]$ are all in the residue class $\mathcal{V}_2(19)$.

4.4 DIGIT SETS FOR RADIX REPRESENTATION

A digit set (denoted by $\mathbb{D}$) is a finite subset of the ring $\mathbb{Z}$ which always contains the zero of the ring. If the coefficients of a radix polynomial are taken from $\mathbb{D}$, the radix polynomial will be over $\mathbb{D}$. In this case, the elements of the digit set (alphabet) are called digits (symbols). Both the radix r and digit set $\mathbb{D}$ characterize the number system. Radix polynomials over a digit set are the same as those given by Equation 4.9 except the digits are now taken from the set $\mathbb{D}$, i.e., $d_m, d_{m-1}, \cdots, d_l \in \mathbb{D}$. The set of radix-$r$ polynomials over $\mathbb{D}$ is given by

$$\mathcal{P}[r, \mathbb{D}] = \{P \mid P = P([r])\}, \tag{4.16}$$

If the numbers to be represented are integers with zero fractional part, then the radix polynomial over $\mathbb{D}$ is given by

$$P([r]) = d_m[r]^m + d_{m-1}[r]^{m-1} + \cdots + d_0[r]^0. \tag{4.17}$$

The condition that a radix polynomial over $\mathbb{D}$ represents all the elements of the ring $\mathbb{Z}$ is given by

$$\forall v \in \mathbb{Z} : \exists P([r]) \in \mathcal{P}[r, \mathbb{D}] \;\Big|\; ||P([r])|| = v. \tag{4.18}$$

The condition in Equation 4.18 is known as the completeness condition and the digit set $\mathbb{D}$ that satisfies this condition is called complete radix r. Thus, it is also complete for $\mathbb{Q}_r$ numbers and the condition in Equation 4.18 becomes

$$\forall v \in \mathbb{Q}_r : \exists P([r]) \in \mathcal{P}[r, \mathbb{D}] \;\Big|\; ||P([r])|| = v. \tag{4.19}$$

For the ring $\mathbb{Z}$, it can be shown that the evaluation mapping $||.||$ in Equation 4.11 is a homomorphism from $\mathcal{P}[r]$ to $\mathbb{Z}$, which partitions $\mathcal{P}[r, \mathbb{D}]$ into residue classes as shown in Figure 4.2. The elements of each class are polynomials evaluated to the same value.

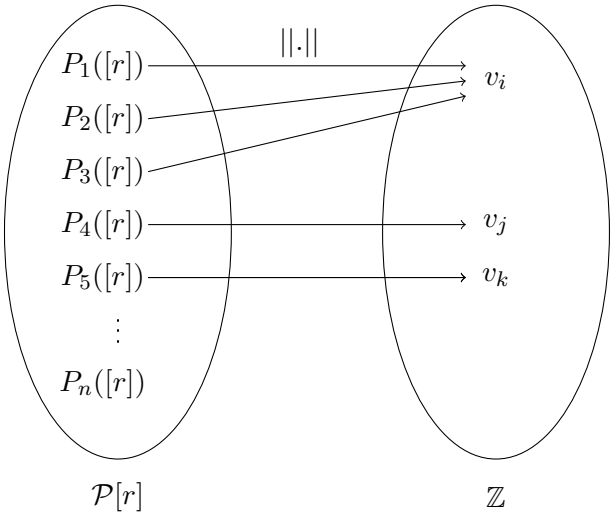


Figure 4.2: The mapping $||.||$ is a homomorphism from $\mathcal{P}[r]$ to $\mathbb{Z}$.

Similar to Equation 4.13, the residue class $\mathcal{V}_r(v)$ with elements evaluated to v is defined by

$$\mathcal{V}_r(v) = \{P([r]) \in \mathcal{P}[r, \mathbb{D}] \;\Big|\; ||P([r])|| = P(r) = v\} \tag{4.20}$$

Equation 4.20 implies that if $P([r]), Q([r]) \in \mathcal{V}_r(v)$, then $P([r]) \equiv Q([r])(\bmod ([r]-r))$. The set $\mathcal{V}_r(v)$ is also defined by the set of redundant representations of v. If the radix polynomials over $\mathbb{D}$ can represent all the elements of the ring $\mathbb{Z}$, then this radix polynomial will satisfy the completeness condition given by

$$\forall v \in \mathbb{Z} : \exists P([r]) \in \mathcal{P}[r, \mathbb{D}] \;\Big|\; ||P([r])|| = v \tag{4.21}$$

The digit set, $\mathbb{D}$, in Equation 4.21 is complete radix r because it contains a complete residue system modulo $|r|$.

4.5 NECESSARY AND SUFFICIENT CONDITIONS FOR COMPLETE RESIDUE SYSTEM MODULO RADIX

The concepts of ideal, cosets, and quotient group are introduced to find the necessary and sufficient conditions for $\mathbb{D}$ to be a complete residue system modulo $|r|$ [Nielsen, 1997]. Assume that the set $\mathcal{J}$ is a subgroup of the ring $\mathbb{Z}$. The subgroup $\mathcal{J}$ is called an ideal in $\mathbb{Z}$ if it has the property $dr \in \mathcal{J}$ and $rd \in \mathcal{J}$ for all $r \in \mathcal{J}$ and $d \in \mathbb{Z}$. If $b \in \mathbb{Z}$, then the set $b + \mathcal{J}$ is termed the left-coset of $\mathcal{J}$ in $\mathbb{Z}$. Similarly, $\mathcal{J} + b$ is called the right coset of $\mathcal{J}$ in $\mathbb{Z}$. Since the additive group of the ring $\mathbb{Z}$ is abelian (described in Section C.2), then $d + \mathcal{J} = \mathcal{J} + d$ [Ayres and Jaisingh, 2004]. The quotient group $\mathbb{Z}/\mathcal{J} = \{b + \mathcal{J} : b \in \mathbb{Z}\}$ is the set of all distinct cosets (residue classes) of $\mathcal{J}$ in $\mathbb{Z}$. The quotient group $\mathbb{Z}/\mathcal{J}$ is a ring with respect to addition and multiplication of cosets (residual classes). Now, two elements $d, b \in \mathbb{Z}$ are considered congruent modulo $\mathcal{J}$ if and only if $d-b \in \mathcal{J}$ or alternatively $d \equiv b \bmod \mathcal{J}$. A set $\mathbb{D}$ is a complete residue system modulo $\mathcal{J}$ if it has exactly one element from each distinct co-set in $\mathbb{Z}/\mathcal{J}$ [Nielsen, 1997].

As an illustrative example, consider the ring $\mathbb{Z}$. The ideal generated by $r \in \mathbb{Z}$, is denoted by $\mathcal{J}_r$ and given by:

$$\begin{aligned} \mathcal{J}_r &= \{dr \;\Big|\; d \in \mathbb{Z}\}, \\ &= \{\cdots, -2r, -r, 0, r, 2r, \cdots\}. \end{aligned}$$

That is, if $c \in \mathcal{J}_r$, then $c \bmod r = 0$. The distinct left cosets (similarly for right cosets) that can be formed are given by

$$0 + \mathcal{J}_r = \{\cdots, -2r, -r, 0, r, 2r, \cdots\}, \tag{4.22}$$

$$1 + \mathcal{J}_r = \{\cdots, 1-2r, 1-r, 1, 1+r, 1+2r, \cdots\}, \tag{4.23}$$

$$\vdots \tag{4.24}$$

$$|r|-1 + \mathcal{J}_r = \{\cdots, |r|-1-2r, |r|-1-r, |r|-1, |r|-1+r, |r|-1+2r, \cdots\}, \tag{4.25}$$

which is a complete residue system modulo $|r|$ and the number of distinct left (right)

cosets in this system is given by $|r|$. For example consider the following left coset

$$3 + \mathcal{J}_r = \{\cdots, 3-2r, 3-r, 3, 3+r, 3+2r, \cdots\}.$$

For the particular value $r = 5$,

$$3 + \mathcal{J}_5 = \{\cdots, 3-10, 3-5, 3, 3+5, 3+10, \cdots\}, \tag{4.26}$$

$$3 + \mathcal{J}_5 = \{\cdots, -7, -2, 3, 8, 13, \cdots\}. \tag{4.27}$$

The relation between the elements of the coset $3 + \mathcal{J}_5$ can be determined by choosing two arbitrary elements, say 13 and 3, then subtracting them which results in:

$$13 - 3 = 10 \in \mathcal{J}_5$$

i.e., $13 \equiv 3 \bmod 5$. The particular coset $3 + \mathcal{J}_5$ is denoted by [3] [Nielsen, 1997]. The distinct cosets modulo 5 are denoted by $[0], [1], \cdots, [4]$. Generally, the distinct cosets modulo $|r|$ are denoted by $[0], [1], \cdots, [|r| - 1]$. Taking one element from each distinct coset forms the set $\mathbb{D} = \{0, 1, \cdots, |r| - 1\}$, which is a complete residue system modulo $|r|$. That is, the cardinality (number of elements) of the set $\mathbb{Z}/\mathcal{J}_r$ denoted by $|\mathbb{Z}/\mathcal{J}_r|$ is given by

$$|\mathbb{Z}/\mathcal{J}_r| = |r|. \tag{4.28}$$

The following theorems and definitions give the necessary and sufficient conditions for completeness, non-redundancy, and redundancy of $\mathbb{D}$ [Nielsen, 1997; Kornerup and Matula, 2010].

Theorem 4.5.1. If a digit set $\mathbb{D}$ is complete radix r for the ring $\mathbb{Z}$, then $\mathbb{D}$ contains a complete residue system modulo $|r|$, hence $|\mathbb{D}| \geq |\mathbb{Z}/\mathcal{J}_r|$.

Proof. Let $L \in \mathbb{Z}$, since $\mathbb{D}$ is complete there exists a polynomial $P \in \mathcal{P}[r, \mathbb{D}]$ with

$$P([r]) = d_L[r]^L + d_{L-1}[r]^{L-1} + \cdots + d_0[r]^0, \ d_i \in \mathbb{D},$$

such that $||P|| = b$ for $b \in \mathbb{Z}$, so $||P|| \equiv d_0 \equiv b \bmod |r|$ and the element b is represented by the residue class $d_0 + \mathcal{J}_r$ where $d_0 \in \mathbb{D}$. Since $0 \in \mathbb{D}$, $\mathbb{D}$ contains a complete residue system modulo $|r|$ and $|\mathbb{D}| \geq |r|$. □

The converse statement does not hold. For example the digit set $\mathbb{D} = \{0, 1\}$ is a complete residue system modulo 2. However, no negative integers can be represented using $\mathbb{D}$, so it is not complete radix 2 for $\mathbb{Z}$.

Theorem 4.5.2. A digit set $\mathbb{D}$ is considered redundant radix r if it is complete radix r and $|\mathbb{D}| > |\mathbb{Z}/\mathcal{J}_r|$.

Proof. Let $P \in \mathcal{P}[r, \mathbb{D}]$ such that

$$P([r]) = d_{L-1}[r]^{L-1} + d_{L-2}[r]^{L-2} + \cdots + d_0[r]^0. \tag{4.29}$$

If $\hat{d} = \max\{|d| \;\big|\; d \in \mathbb{D}\}$, then for all $P \in \mathcal{P}[r, \mathbb{D}]$,

$$\begin{aligned} |\,||P||\,| \leq \;& \hat{d}\,(|r|^{L-1} + |r|^{L-2} + \cdots + 1), && (4.30)\\ = \;& \hat{d}\,\tfrac{|r|^L-1}{|r|-1}. && (4.31) \end{aligned}$$

For symmetric digit set, the range of represented integers on number line is given by

$$\Big[-\hat{d}\,\frac{|r|^L-1}{|r|-1}, \hat{d}\,\frac{|r|^L-1}{|r|-1}\Big]. \tag{4.32}$$

The number of the represented integers (number line length) is

$$2\hat{d}\,\frac{|r|^L-1}{|r|-1}. \tag{4.33}$$

But the number of all the possible representations that can be obtained from using the digit set $\mathbb{D}$ is given by

$$\Gamma = \prod_{l=0}^{L-1} |\mathbb{D}| = |\mathbb{D}|^L \geq ||r|+1|^L, \tag{4.34}$$

so

$$\lim_{L\to\infty} \frac{2\hat{d}\,\frac{|r|^L-1}{|r|-1}}{||r|+1|^L} = 0, \tag{4.35}$$

i.e., the number of polynomials over $\mathbb{D}$ is larger than values to represent, thus $\mathbb{D}$ is redundant radix r [Nielsen, 1997] [Kornerup and Matula, 2010]. □

As an example of the number of representations, consider the number system with $r = 2$, $\mathbb{D} = \{\bar{1}, 0, 1\}$, and $L = 8$. The range of represented integers is

$$[-2^8 + 1, 2^8 - 1],$$

and the number of values to represent is shown to be

$$2\frac{2^8-1}{1} = 510.$$

The number of all possible representations is given by,

$$3^8 = 6561,$$

Therefore the average number of representations for a value is given by

$$\frac{6561}{510} \approx 13.$$

Theorem 4.5.1 and Theorem 4.5.2 are essentials to develop Lemma 5.1.1 in the next chapter. This lemma is the algebraic basis for tree and graph algorithms presented in Chapter 5 to generate redundant numbers.

Chapter 5

GENERATING THE REDUNDANT REPRESENTATIONS

Number representations (encoding) of a value can be obtained by generating a directed graph. The generating starts from the root which is the value to be encoded and ends at leaf nodes with value zero. New child nodes spawn from its parent according to the congruent relation (shown later) which partition the digit set into equivalent classes. This relation can be described on a graph by branching from a parent node to its children. If the parent-child relation is restricted such that each child has only one parent, the generated graph is expanded to a representation tree. On the other hand, removing this restriction collapses the representation tree to a graph. Therefore, graph and tree algorithms (encoders) are presented in this chapter to generate the representations of a value for a given digit set and radix. The two algorithms are important to solve the multiple constant multiplication (MCM) problem because each one can be customized differently to search for the optimal subexpression sharing. The tree algorithm is selected to study the size of the representation tree because it is more illustrative than the graph encoder. The size of representation tree is derived using a Markov transition matrix and verified through simulation. The development considered the binary singed digit (BSD) representations and a study of generalizing it to include other number systems is presented.

The chapter is structured as follow: The tree encoder is described in Section 5.1. The graph encoder is described in Section 5.2. In Section 5.3, a closed-form is derived for the size of representation tree. The tree and graph encoders performance is investigated in Section 5.4. A chapter summary is given in Section 5.5.

5.1 REPRESENTATION TREE ALGORITHM

The major contribution in this chapter is providing a new interpretation of Lemma 5.1.1 in [Kornerup and Matula, 2010] to develop the representation tree algorithm as follows. Lemma 5.1.1 is derived from Equation 4.11 and Equation 4.21 to find a representation from the polynomial, $P([r])$, as

Lemma 5.1.1. Let $\mathbb{D}$ be a digit set which is complete residue modulo $|r|$, $v \in \mathbb{Z}$, and $d_0 \in \mathbb{D}$ such that $v \equiv d_0 \bmod |r|$. Then there exists a radix polynomial $P \in \mathcal{P}[r, \mathbb{D}]$ such that $||P|| = v$ if and only if there exists $P' \in \mathcal{P}[r, \mathbb{D}]$ with $||P'|| = \frac{v-d_0}{r}$.

Proof. For the if part let $P \in \mathcal{P}[r, \mathbb{D}]$ such that $||P|| = v$. Then

$$P = \sum_{i=0}^{m} d_i r^i, \tag{5.1}$$

so

$$||P|| = v = \sum_{i=0}^{m} d_i r^i \equiv d_0 \bmod |r|, \tag{5.2}$$

where $d_0 \in \mathbb{D}$, and $v \equiv d_0 \bmod |r|$ is called v congruent to d_0 modulo r. Therefore,

$$\frac{v - d_0}{r} = \sum_{i=1}^{m} d_i r^{i-1}. \tag{5.3}$$

Consequently $P' = \sum_{i=1}^{m} d_i r^{i-1} \in \mathcal{P}[r, \mathbb{D}]$ with

$$||P'|| = v' = \frac{v - d_0}{r}. \tag{5.4}$$

For the only if part assume $P' \in \mathcal{P}[r, \mathbb{D}]$ with $||P'|| = \frac{v-d_0}{r}$. Now, define $P(r) = P'(r) + d_0$, then $P \in \mathcal{P}[r, \mathbb{D}]$ and

$$||P|| = ||P'||r + d_0 = \frac{v - d_0}{r} r + d_0 = v. \tag{5.5}$$

□

Applying Equation 5.4 recursively until reaching a zero polynomial, generates a string of digits that represent the value v. Here, the digit d_0 is the least significant digit (LSD) and the digit d_m is in the position of the most significant digit (MSD).

From Equation 5.2, we conclude that the congruent relation, $||P|| \equiv d_0 \bmod |r|$, partitions the digit set, $\mathbb{D}$, into $|r|$ residue classes (disjoint sets). This partitioning is shown in Figure 5.1.

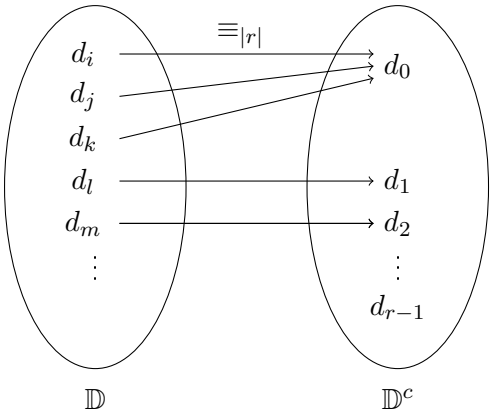


Figure 5.1: The congruent relation, $\equiv_{|r|}$, partitions the digit set $\mathbb{D}$ into $|r|$ disjoint sets, where $\mathbb{D}^c$ is non redundant complete residue modulo $|r|$, i.e. $|\mathbb{D}^c| = |r|$.

The tree encoder is developed by rewriting Equation 5.2 as

$$v = \sum_{i=0}^{L-1} d_i r^i = d_0 + d_1 r + \cdots + d_{L-1} r^{L-1}, \tag{5.6}$$

or

$$v - d_0 = d_1 r + \cdots + d_{L-1} r^{L-1}. \tag{5.7}$$

We observe that the difference $v - d_0$ is divisible by the radix r, or alternatively v is congruent to d_0 modulo r as shown in Equation 5.2.

If the digit set is redundant, then there could be more than one d_0 satisfying Equation 5.2. For example, consider using the BSD to encode $v = 281$. Both of the two digits $\{\overline{1}, 1\}$ satisfy Equation 5.2 and thus

$$(281 - 1)/2 = 140 \text{ and } (281 - (-1))/2 = 141. \tag{5.8}$$

Equation 5.8 can be visualized on a tree as shown in Figure 5.2. In this case, the value 281 is the root node while 140 and 141 are the left and right children respectively. The edge values are the digits $\overline{1}, 1$ computed from Equation 5.8.

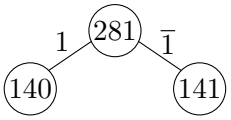


Figure 5.2: Visualization of the first depth division of 281 by $r = 2$ with $\mathbb{D} = \{\overline{1}, 0, 1\}$.

Proceeding in the same way with the children 140 and 141 results in growing the representation tree of 281 shown in Figure 5.3. Tree growth can be expressed by formulating Equation 5.8 as a recursion given by

$$v_c = \frac{v_p - d}{r}, \tag{5.9}$$

where v_p is the parent node value, v_c is the child node value, and d is the edge value between them. Edge attributes are obtained by applying Equation 5.2. These are digits released from their sets when the modulo operation is applied at a node. The modulo operation partitions the digit set into r disjoint sets as

$$\mathbb{D} = \bigcup_{\gamma=0}^{|r|-1} \mathbb{D}_\gamma, \tag{5.10}$$

where the disjoint set γ is given by

$$\mathbb{D}_\gamma = \{d_i : d_i \bmod r = \gamma\}, \tag{5.11}$$

and $d_i \in \mathbb{D}$. For example, in the BSD number system the digit set is partitioned into two disjoint sets, $\mathbb{D}_0 = \{0\}$ and $\mathbb{D}_1 = \{\overline{1}, 1\}$.

The tree encoder algorithm is shown in Listing 1. The inputs to the algorithm are the value to be encoded v, digit set $\mathbb{D}$, radix r, and wordlength L. The representation tree list is initialized with the root node of value v as shown in step 2 of Listing 1. A breadth first search is used to generate the tree as shown in step 3. In this case, the wordlength value limits the tree from growing to infinity. An empty child's list is initialized in step 4. The loop in step 5 causes each node at depth l to spawn a list of children at depth $l+1$. The branching from a node at depth l to another one at depth $l+1$ is according to the digit set partitioning in steps 6 and 7. The value of the new born child is calculated in step 8. The child node is born at step 9 and appended to child list C as shown in step 10. Only children at depth $l+1$ are stored in the tree list as shown in step 11. This is because each node stores the string of released digits (starting from the LSD) in its data structure.

```
1:  procedure TREE(v, D, r, L)
2:    T ← {NewNode(v)}                          ▷ Tree with only root node
3:    for l ← 0 to L do                         ▷ Breadth-first traversing
4:      C ← {}                                  ▷ Empty child list
5:      for each node ∈ T do
6:        v_p = node.v                          ▷ Parent node value
7:        γ ← mod(v_p, r)                       ▷ Partition index
8:        for each d ∈ D do                     ▷ Iterate over digits
9:          if mod(d, r) = γ then               ▷ Partition digit set according to Equation 5.10
10:           v_c ← (v_p − d)/r                 ▷ Calculating child node value v_c as in Equation 5.9
11:           child ← NewNode(v_c)              ▷ New child is born
12:           C ← C + {child}                   ▷ Concatenate child list
13:     T ← C                                   ▷ Next generation
14:   return T
```

Listing 1: Tree algorithm to generate an encoding tree, T, for a value v, with wordlength L, digit set $\mathbb{D}$, and radix r.

Figure 5.3 shows the result of applying the algorithm in Listing 1 to find the BSD representation tree of the value $v = 281$. The growth of the tree shown in Figure 5.3 starts from the root node of value 281. The modulo devision $281 \bmod 2 = 1$ results in releasing the digits in the set $\mathbb{D}_1 = \{\overline{1}, 1\}$. So two edges are emitted from the root node with attributes $\overline{1}$ and 1 respectively. Each digit is in the position of the least significant digit (LSD). The left and right child of 281 are 140 and 141, respectively, which are generated according to Equation 5.9. The process of generating nodes continues until reaching nodes of zero value (leaf nodes). Valid representations are obtained when reaching leaf nodes. Thus, leaf edges are in the position of the most significant digit (MSD). A representation is obtained from tracking edge attributes from the MSD to the LSD. For example, the representation 100011001 is obtained from tracking the red path in Figure 5.3.

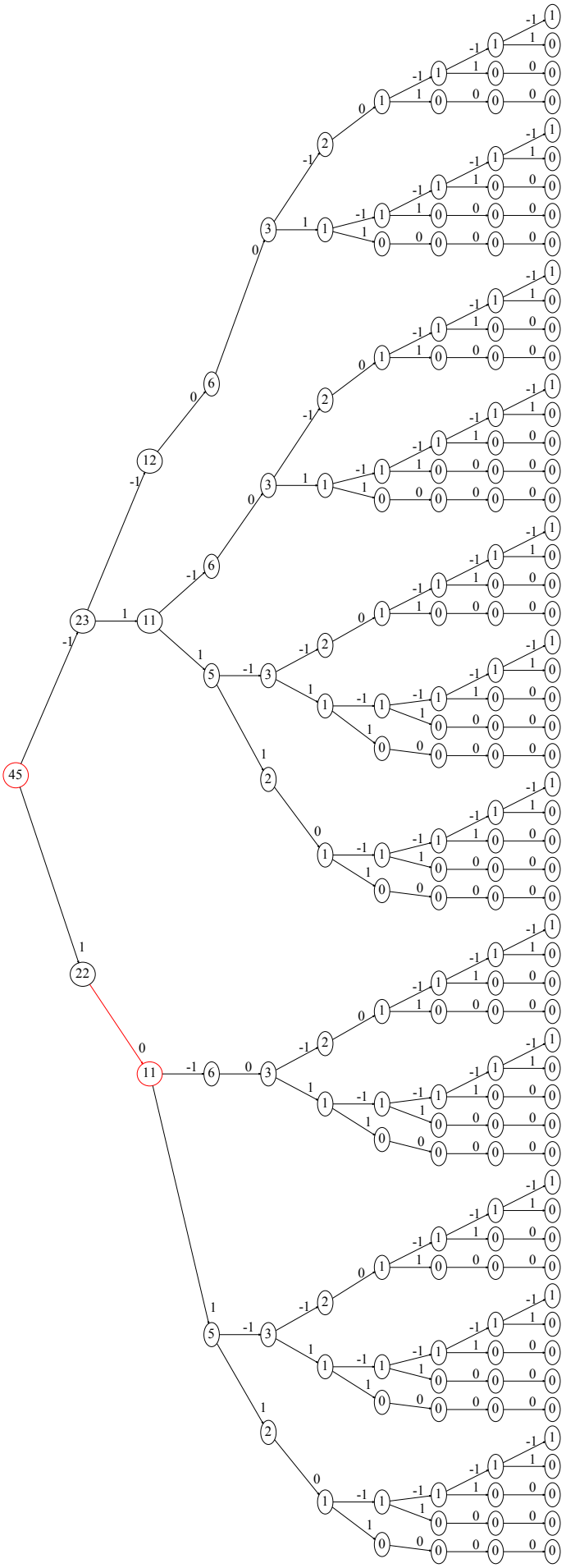


Figure 5.3: An example of the representation tree for the value $v = 281$ using BSD with wordlength $L = 9$. Edge values are the digits of the representations.

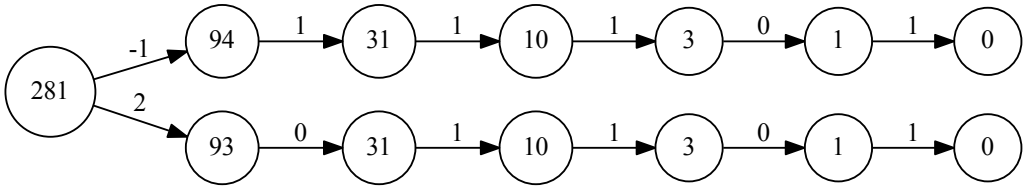


Figure 5.4: An example of the encoding tree for the value $v = 25$ using number system with $r = 3$ and $\mathbb{D} = \{\overline{1}, 0, 1, 2\}$.

Another example of applying the algorithm in Listing 1 is the encoding of the value 281 using the number system with $r = 3$ and $\mathbb{D} = \{\overline{1}, 0, 1, 2\}$ as shown in Figure 5.4. The digit set is partitioned to three disjoint sets which are $\mathbb{D}_0 = \{0\}$, $\mathbb{D}_1 = \{1\}$, and $\mathbb{D}_2 = \{\overline{1}, 2\}$. The tree shown in Figure 5.4 indicates that the tree size is related to the radix and digit set of the used number system. This relation is further investigated in Section 5.4.

5.2 GRAPH ENCODER

The tree encoder described in Section 5.1 generates the representation tree in breadth first search. It restrict the parent-child relation such that each child at depth L_i has only one parent at depth L_{i-1}. If the restriction parent-child relation is removed, the representations of a value v is found by traversing a graph. For example, the representation tree shown in Figure 5.3 is collapsed to the graph shown in Figure 5.5.

A possible implementation for the graph encoder algorithm is shown in Listing 2. It builds the encoding graph recursively by using depth first search (DFS) as shown in step 8. At each node the DFS traverses only one of the node out edges. Other edges are stored due to the recursion implementation and traversed later when the algorithm backtrack. This method of constructing the graph encoder helps in enumerating the representations of a value in succession.

```
1: procedure GRAPH(G, v, 𝔻, r)
2:     ADDNODE (G, v)                          ▷ Add node v to graph G
3:     γ ← mod (v, r)                          ▷ Partition index
4:     for each d ∈ 𝔻 do                       ▷ Iterate over digits
5:         if mod (d, r) = γ then              ▷ Partition digits
6:             v′ ← (v − d)/r                  ▷ Next node value
7:             if v′ ∉ G then
8:               GRAPH(G, v′, 𝔻, r)            ▷ Build recursively
9:             ADDEDGE (G, v, v′, d)           ▷ Add edge d
```

Listing 2: An algorithm to generate encoding graph G for a value v with digit set $\mathbb{D}$ and radix r.

Consider the encoding graph shown in Figure 5.5. A representation is generated starting from the root of value 281 (which is in the position of LSD) and ending at the node of zero value (which is in the position of MSB). The self loop at node 0 yields

zero padding at the left of a representation. The number of traversing the self loop is specified by the wordlength. Next representations are generated when the algorithm backtrack to the stored unvisited yet edges. Another example is shown Figure 5.6 which depict the graph encoder of the value 281 using the number system with $r = 3$ and $\mathbb{D} = \{-1, 0, 1, 2\}$.

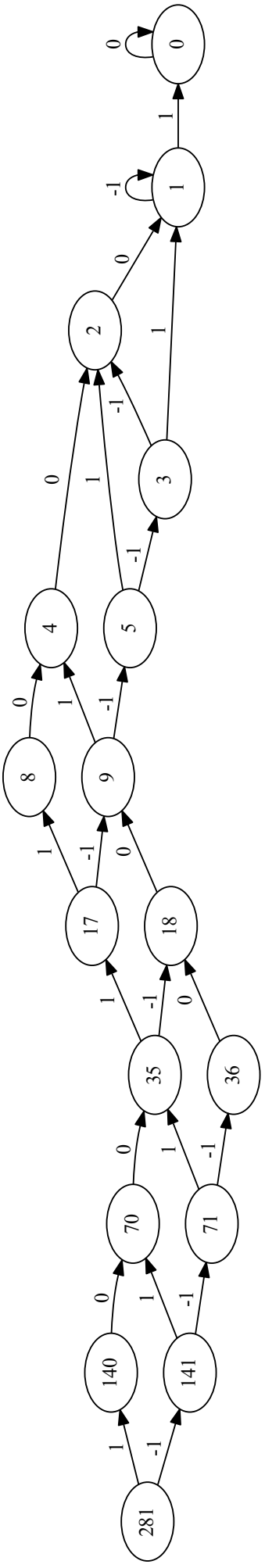


Figure 5.5: An example of the encoding graph for the value $v = 25$ using BSD.

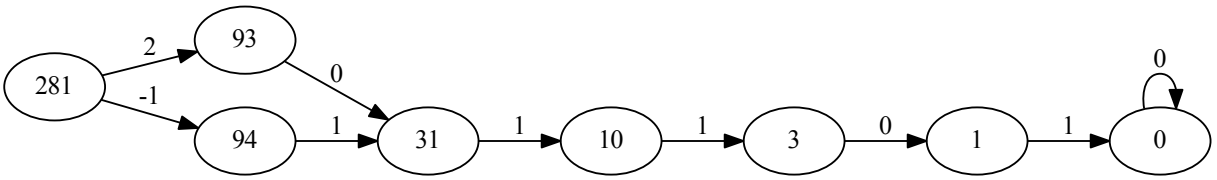


Figure 5.6: An example of the encoding graph for the value $v = 25$ using the number system with $r = 3$ and $\mathbb{D} = \{\overline{1}, 0, 1, 2\}$.

5.3 REPRESENTATION TREE SIZE

The complexity of tree algorithm is specified by the size of representation tree, where this size is equal to the number of nodes in the tree. Developing a closed form for the tree size can help in determining the complexity of the tree algorithm. This will be more clear in the next chapter that modifies the tree algorithm to search for common subexpression. Therefore, it is the purpose of this chapter is drive a closed forms for the size of representation tree, number of representations, and number of nodes at a certain depth of the tree.

The growth of the representation tree depends on the digit set, radix, and partitions. This requires defining the tree growth factor g. For a consistent digit set (has equal cardinality partitions) the growth factor is $g = |\mathbb{D}_0| = |\mathbb{D}_1| = \cdots = |\mathbb{D}_{r-1}|$, where $|\mathbb{D}_\gamma|$ is defined in Equation 5.10. If the digit set is inconsistent then it is required to find the cardinality of the largest partition

$$|\mathbb{D}_\mathrm{m}| = \max(|\mathbb{D}_0|, \cdots, |\mathbb{D}_{r-1}|). \tag{5.12}$$

A maximum tree growth occurs when $|\mathbb{D}_\gamma| = |\mathbb{D}_\mathrm{m}|$ for $\gamma = 0, \cdots, r-1$. In this case, the cardinality of the digit set equals to $r\,|\mathbb{D}_\mathrm{m}| > |\mathbb{D}|$. Therefore, the growth factor is reduced by the ratio $\frac{|\mathbb{D}|}{r|\mathbb{D}_\mathrm{m}|}$ and the growth factor for an inconsistent digit set becomes

$$g = |\mathbb{D}_\mathrm{m}| * \frac{|\mathbb{D}|}{r\,|\mathbb{D}_\mathrm{m}|} = \frac{|\mathbb{D}|}{r} \tag{5.13}$$

When $g > 1$ the average size of the tree will grow exponentially as a result of summing a geometric series. Hence the average tree size at depth L, denoted $T(L)$ is:

$$T(L) = \sum_{k=0}^{L} g^k \tag{5.14}$$

$$= \frac{g^{L+1} - 1}{g - 1} \quad , g \neq 1 \tag{5.15}$$

For BSD, $g = 3 - (2 \times 2)/3 = 1.66$ resulting in an exponential growth of the tree. When $g = 1$, such as for the case of binary, L'Hôpital's rule needs to be applied to Equation 5.15, resulting in a linear growth of the tree which equals to L.

The tree growth changes once the tree depth L is large enough to reach a zero node.

This is the minimum wordlength ($L_{\min}$) required to encode the value. An example of $L_{\min}$ is shown in Figure 5.7 which results from encoding the value 25 in BSD using the tree encoder. The corresponding $L_{\min}$ on the graph encoder of the value 25 is shown in Figure 5.8.

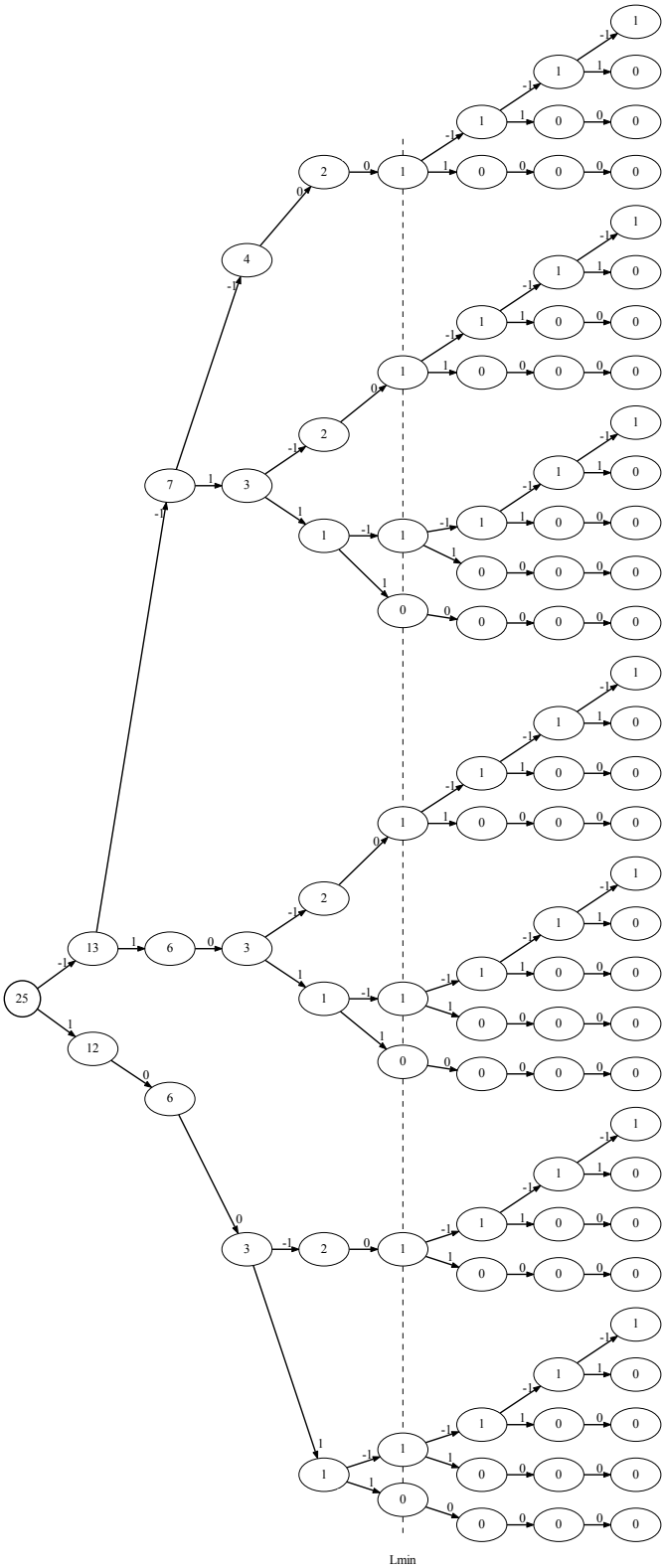


Figure 5.7: An example of the encoding tree for the value $v = 25$ using BSD. $L_{\min}$ is the minimum wordlength to reach a zero node.

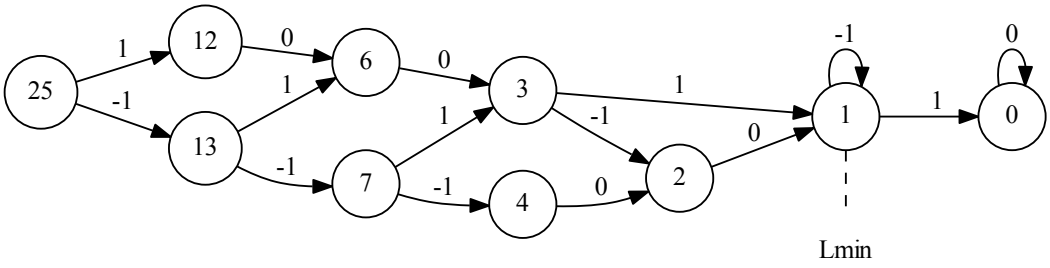


Figure 5.8: An example of the encoding graph for the value $v = 25$ using BSD. $L_{\min}$ is the minimum wordlength to traverse a self loop.

In this case, $L_{\min}$ is defined as the minimum wordlength required to finish traversing a self loop. To find $L_{\min}$, consider Equation 4.32 of Theorem 4.5.2 (page 70). Assign the symbol v^- to the lower limit of Equation 4.32 and v^+ for the upper limit results,

$$[v^-, v^+] = \Big[-\hat{d}\,\frac{|r|^{L_-} - 1}{|r| - 1}, \hat{d}\,\frac{|r|^{L_+} - 1}{|r| - 1}\Big]. \tag{5.16}$$

Consider a general case of asymmetric digit set such that, $d_\uparrow$ is the most positive digit in $\mathbb{D}$, and $d_\downarrow$ is the most negative one, then for the upper limit (in BSD $d_\uparrow = 1$ and $d_\downarrow = -1$),

$$v^+ = d_\uparrow \frac{|r|^{L_+} - 1}{|r| - 1}, \tag{5.17}$$

so

$$\frac{v^+}{d_\uparrow}(|r| - 1) = |r|^{L_+} - 1, \tag{5.18}$$

then

$$|r|^{L_+} = \frac{v^+}{d_\uparrow}(|r| - 1) + 1, \tag{5.19}$$

which results

$$L_+ = \lceil \log_r(\frac{v^+}{d_\uparrow}(|r| - 1) + 1)\rceil. \tag{5.20}$$

Similarly for L_-

$$L_- = \lceil \log_r(\frac{v^-}{|d_\downarrow|}(|r| - 1) + 1)\rceil. \tag{5.21}$$

In this case, $L_{\min}$ is found using the following formula

$$L_{\min} = \max(L_-, L_+). \tag{5.22}$$

A conservative ciel function is used to find L_- and L_+ as shown in Equation 5.20 and Equation 5.21 respectively. An example of calculating $L_{\min}$ is for the range

$[v^-, v^+] = [-256, 256]$ using BSD. In this case, it is found that $L_{\min} = 9$.

Mathematically speaking, the nodes after $L > L_{\min}$ are belong to a reduced set $\mathbb{S}$. An example of the reduced set is the set $\mathbb{S} = \{\bar{1}, 0, 1\}$ for the BSD as shown in Figure 5.9.

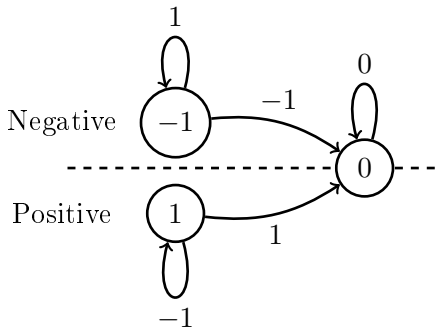


Figure 5.9: BSD state transitions in the recursive region.

The transitions between the states in $\mathbb{S}$ shown in Figure 5.9 can be described by a Markov transition matrix $\mathbf{Q}$ [Wai-K and Michael, 2006], where the matrix element q_{ij} is the number of paths from state i to state j. If we define the average state proportions in the tree at depth L by a vector $\mathbf{p}(L)$, then after k transitions the proportions will be given by

$$\mathbf{p}(L + k) = \mathbf{Q}^k \mathbf{p}(L). \tag{5.23}$$

The total tree size can be found by summing these proportions to yield

$$T(L_{\min} + K) = T(L_{\min}) + M(L_{\min})\mathbf{1}^T \left[\sum_{k=1}^{K} \mathbf{Q}^k \right] \mathbf{p}(L_{\min}), \tag{5.24}$$

where $\mathbf{1}$ is the ones vector of length $|\mathbb{S}|$, and $M(L_{\min})$ is the number of nodes at depth $L_{\min}$, i.e

$$M(L_{\min}) = T(L_{\min}) - T(L_{\min} - 1). \tag{5.25}$$

In the case of BSD, the transition matrix for $\mathbb{S} = \{-1, 0, 1\}$ is

$$\mathbf{Q} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.26}$$

and so after k transitions

$$\mathbf{Q}^k = \begin{bmatrix} 1 & 0 & 0 \\ k & 1 & k \\ 0 & 0 & 1 \end{bmatrix}, \tag{5.27}$$

and thus

$$\sum_{k=1}^{K} \mathbf{Q}^k = K \begin{bmatrix} 1 & 0 & 0 \\ (K+1)/2 & 1 & (K+1)/2 \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.28}$$

The average proportion vector at $L = L_{\min}$ can be calculated from the ratios of different states transitions that shown in Figure 5.9. In the case of encoding negative values, the proportion vector is given by

$$\mathbf{p}^{-}(L_{\min}) = \begin{bmatrix} \frac{1}{3} & \frac{2}{3} & 0 \end{bmatrix}^T. \tag{5.29}$$

Substituting Equation 5.28 and Equation 5.29 into Equation 5.24 yields

$$T(L_{\min} + K) = T(L_{\min}) + M(L_{\min})K\left(\frac{K+7}{6}\right). \tag{5.30}$$

The proportion vector for encoding positive values is given by

$$\mathbf{p}^{+}(L_{\min}) = \begin{bmatrix} 0 & \frac{2}{3} & \frac{1}{3} \end{bmatrix}^T. \tag{5.31}$$

Substituting Equation 5.28 and Equation 5.31 into Equation 5.24 yields the same result as Equation 5.30. This shows that the average size of the tree grows quadratically once $L > L_{\min}$.

5.4 INVESTIGATING THE PERFORMANCE OF THE GRAPH AND TREE ALGORITHMS

The performance of the graph and tree algorithms were investigated by generating redundant representations for several examples. The tree algorithm was used to generate BSD representations for the numbers in the range $[-256, 256]$. The value of $L_{\min}$ was evaluated for each number in the given range according to Equation 5.22. The theoretical tree size was calculated according to Equation 5.15 for $L \leq L_{\min}$ and according to Equation 5.30 for $L > L_{\min}$. The measured tree size was recorded using programming in Python. The theoretical and measured tree sizes were averaged for each wordlength and compared as shown in Figure 5.10. The result in Figure 5.10 shows that the measured average tree size agrees with the theoretical one because the derivation was for BSD.

A set of experiments was carried out to compare between the size of representation tree computed using Equation 5.30 (theoretical) with that obtained from the simulation (experimental) for several number systems. The purpose of this comparison is to determine the possibility of generalizing Equation 5.30. The results of comparison are shown in Figure 5.11 to Figure 5.16. Inspecting the plots shows that the overlap between the theoretical and experimental results is perfect only in the case of BSD. Other number

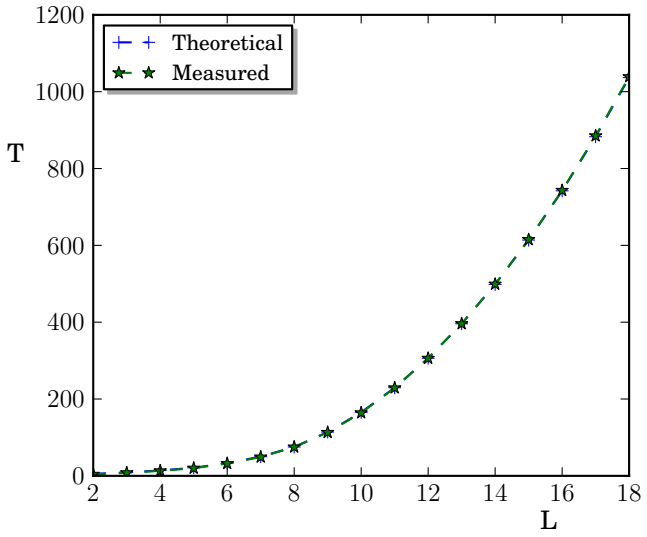


Figure 5.10: A comparison between theoretical and measured average BSD tree size for the number range $[-256, 256]$.

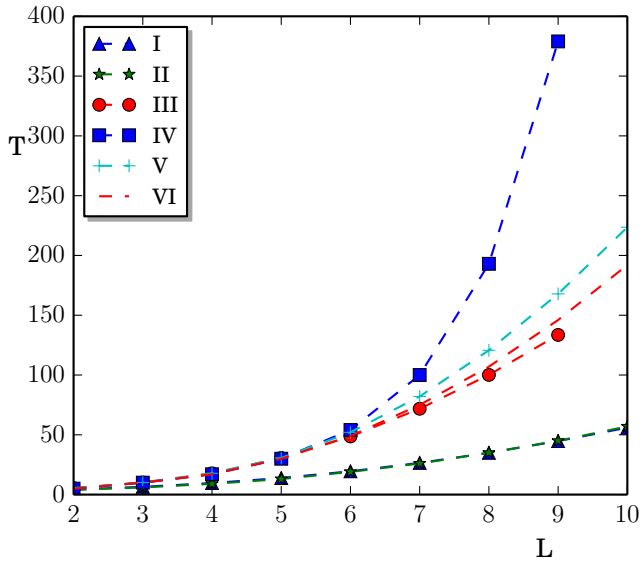


Figure 5.11: Average tree size, T, for the range $[-256, 256]$. Plots (I) and (II) are respectively the theoretical and measured average tree size which obtained from using number system with $r = 3$, $\mathbb{D} = \{\overline{1}, 0, 1, 2\}$. Plots (III) and (IV) are respectively the theoretical and measured tree size obtained from using number system with $r = 3$, $\mathbb{D} = \{\overline{1}, 0, 1, 2, 3\}$. Plots (V) and (VI) are respectively the theoretical and measured tree size obtained from using number system with $r = 3$, $\mathbb{D} = \{\overline{2}, \overline{1}, 0, 1, 2\}$.

systems show less degree of overlapping. Therefore, it is clear that Equation 5.30 cannot be generalized to find tree size of other number systems than BSD. A similar derivation to that given in Equation 5.25 to Equation 5.30 to find a closed form for the tree size obtained from using other number systems than BSD.

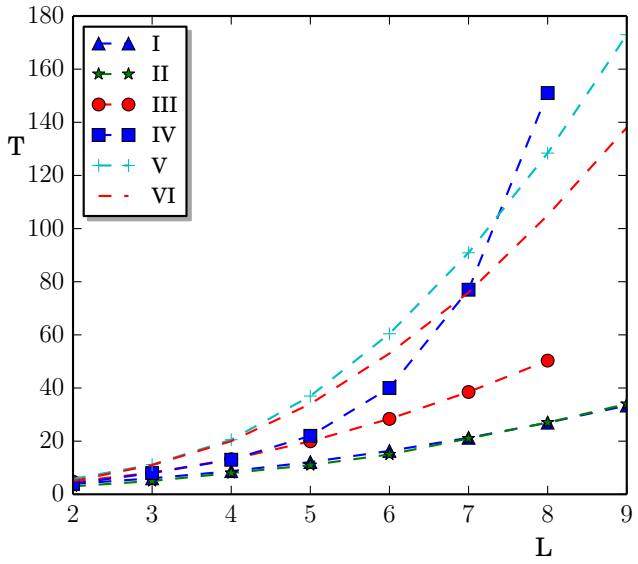


Figure 5.12: Average tree size, T, for the range $[-256, 256]$. Plots (I) and (II) are respectively the theoretical and measured average tree size which obtained from using number system with $r = 4$, $\mathbb{D} = \{\overline{1}, 0, 1, 2, 3\}$. Plots (III) and (IV) are respectively the theoretical and measured tree size obtained from using number system with $r = 4$, $\mathbb{D} = \{\overline{1}, 0, 1, 2, 3, 4\}$. Plots (V) and (VI) are respectively the theoretical and measured tree size obtained from using number system with $r = 4$, $\mathbb{D} = \{\overline{3}, \overline{2}, \overline{1}, 0, 1, 2, 3\}$.

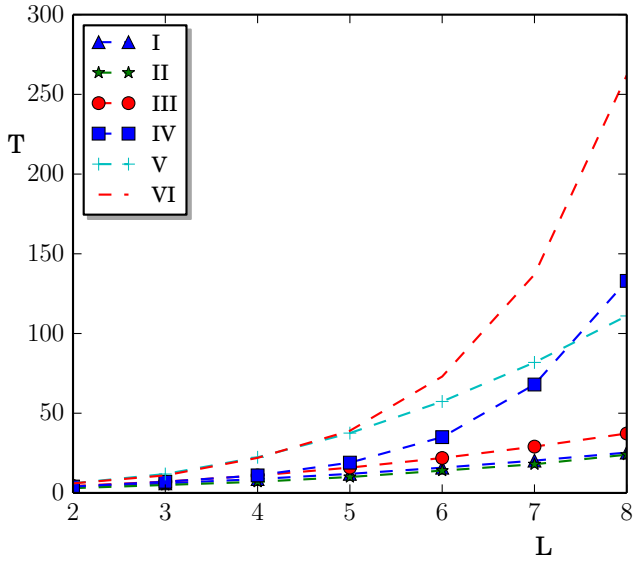


Figure 5.13: Average tree size, T, for the range $[-256, 256]$. Plots (I) and (II) are respectively the theoretical and measured average tree size which obtained from using number system with $r = 5$, $\mathbb{D} = \{\overline{1}, 0, 1, 2, 3, 4\}$. Plots (III) and (IV) are respectively the theoretical and measured tree size obtained from using number system with $r = 5$, $\mathbb{D} = \{\overline{1}, 0, 1, 2, 3, 4, 5\}$. Plots (V) and (VI) are respectively the theoretical and measured tree size obtained from using number system with $r = 5$, $\mathbb{D} = \{\overline{4}, \overline{3}, \overline{2}, \overline{1}, 0, 1, 2, 3, 4\}$.

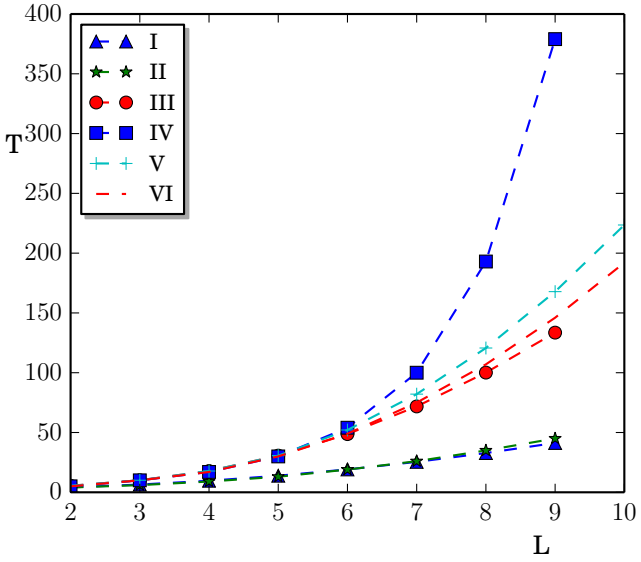


Figure 5.14: Average tree size, T, for the range $[-256, 256]$. Plots (I) and (II) are respectively the theoretical and measured average tree size which obtained from using number system with $r = 3$, $\mathbb{D} = \{0, 1, 2, 3\}$. Plots (III) and (IV) are respectively the theoretical and measured tree size obtained from using number system with $r = 3$, $\mathbb{D} = \{\overline{1}, 0, 1, 2, 3\}$. Plots (V) and (VI) are respectively the theoretical and measured tree size obtained from using number system with $r = 3$, $\mathbb{D} = \{\overline{2}, \overline{1}, 0, 1, 2\}$.

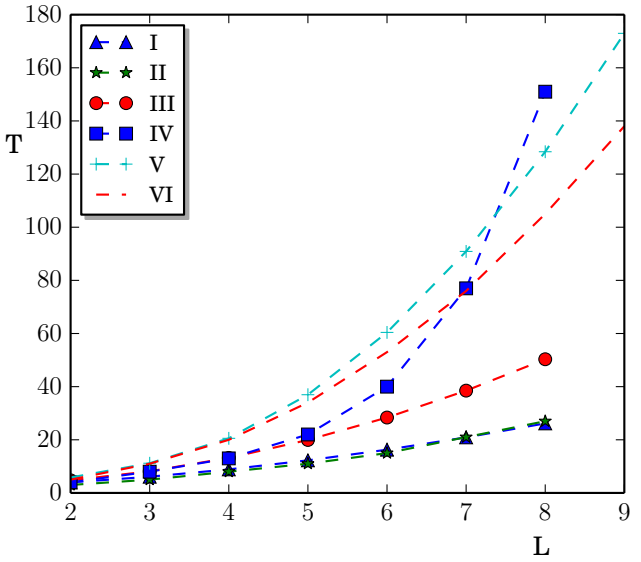


Figure 5.15: Average tree size, T, for the range $[-256, 256]$. Plots (I) and (II) are respectively the theoretical and measured average tree size which obtained from using number system with $r = 4$, $\mathbb{D} = \{0, 1, 2, 3, 4\}$. Plots (III) and (IV) are respectively the theoretical and measured tree size obtained from using number system with $r = 4$, $\mathbb{D} = \{\overline{1}, 0, 1, 2, 3, 4\}$. Plots (V) and (VI) are respectively the theoretical and measured tree size obtained from using number system with $r = 4$, $\mathbb{D} = \{\overline{3}, \overline{2}, \overline{1}, 0, 1, 2, 3\}$.

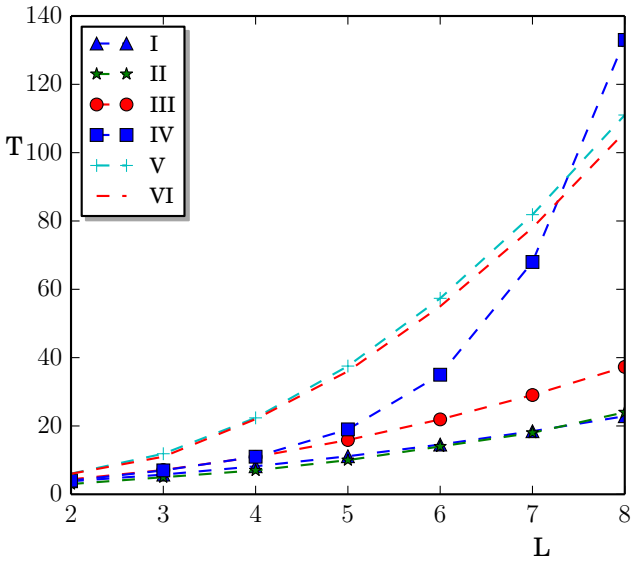


Figure 5.16: Average tree size, T, for the range $[-256, 256]$. Plots (I) and (II) are respectively the theoretical and measured average tree size which obtained from using number system with $r = 5$, $\mathbb{D} = \{0, 1, 2, 3, 4, 5\}$. Plots (III) and (IV) are respectively the theoretical and measured tree size obtained from using number system with $r = 5$, $\mathbb{D} = \{\overline{1}, 0, 1, 2, 3, 4, 5\}$. Plots (V) and (VI) are respectively the theoretical and measured tree size obtained from using number system with $r = 5$, $\mathbb{D} = \{\overline{4}, \overline{3}, \overline{2}, \overline{1}, 0, 1, 2, 3, 4\}$.

The time elapsed by the tree algorithm was recorded for generating the representations of bunches of numbers with the same $L_{\min}$ as shown in Figure 5.17 and Figure 5.18. Each curve in these figures represents the average of all the numbers with the same $L_{\min}$. The values of $L_{\min}$ were $6, 7, \cdots, 13$. Each curve is rising exponentially until $L_{\min}$ as described in Equation 5.14. After $L_{\min}$, the time curve continue rising with a steepest slope. The steepest slope is due to the near duplication in the number of nodes at depth $L_{\min} + 1$ as compared with that at $L_{\min}$.

The algorithm were also used to generate the BSD representations of the coefficients for filters: L_1 (as given in Example 1 of Lim and Parker [1983]) and S_2 (as given in Examples 2 of Samueli [1989]). These filters are symmetric with lengths 121 and 60, respectively. Figure 5.19 shows the number of BSD representations plotted against wordlength L for the two filters. The wordlength was varied from $L_{\min}$ to a maximum wordlength of 32 digits. The result in Figure 5.19 show the exponential growth of number of representation with wordlength. It also show that the larger filter (L_1 in this case) results in larger set of representations and consequently larger search space. Similar results are obtained when the algorithm used to generate the BSD representations for the filters: L_3 with length 36 (as given in Example 3 of Lim and Parker [1983]) and S_1 with lengths 25 (as given in Example 1 of Samueli [1989]) as shown in Figure 5.20.

Though the tree continue grow exponentially even when $L > L_{\min}$, it is possible to reduce the algorithm computational complexity by preventing nodes of zero value from spawn a child node. Nodes with zero value has no effect on the number of representations

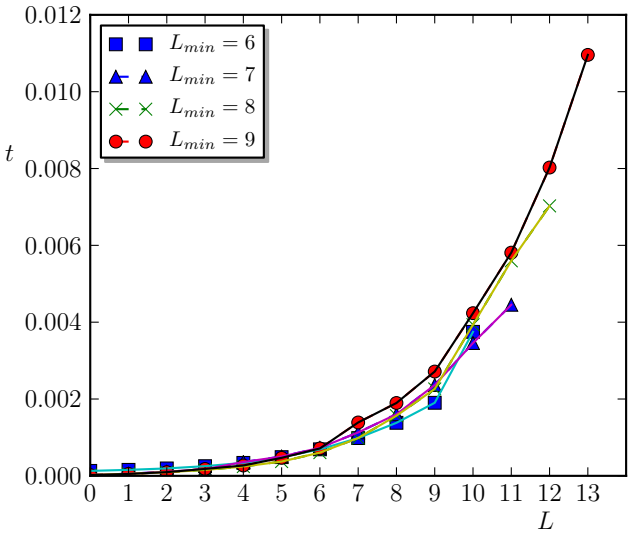


Figure 5.17: Elapsed time in seconds to generate the BSD representations for bunches of numbers with $L_{\min} = 6, 7, 8,$ and 9 respectively.

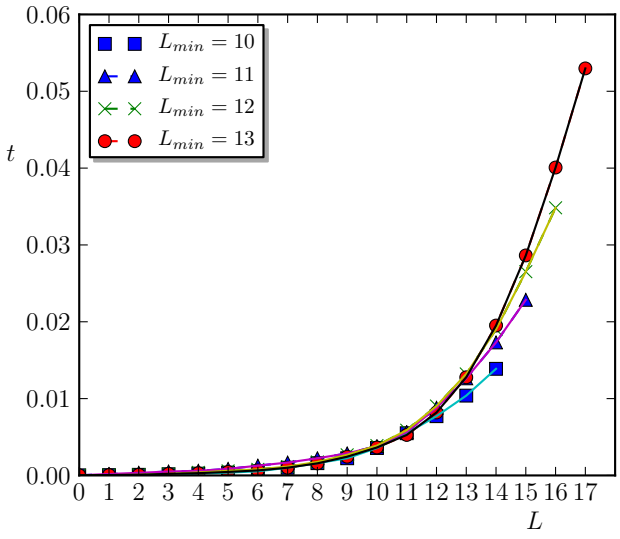


Figure 5.18: Elapsed time in seconds to generate the BSD representations for bunches of numbers with $L_{\min} = 10, 11, 12,$ and 13 respectively.

at the next depth, they are just appeared as zero padding to the left of a representation. For example, consider the tree of value 25 shown in Figure 5.7. The branches beyond zero valued nodes are pruned as shown in Figure 5.21. This make the number of generated representations at each new depth after $L_{\min}$ constant (equals to 7 in this example) as shown in Figure 5.21.

To study the effect of digit set cardinality on the number of representations, five digit sets radix-2 were chosen as shown in Figure 5.22. In this figure, the y-axis is

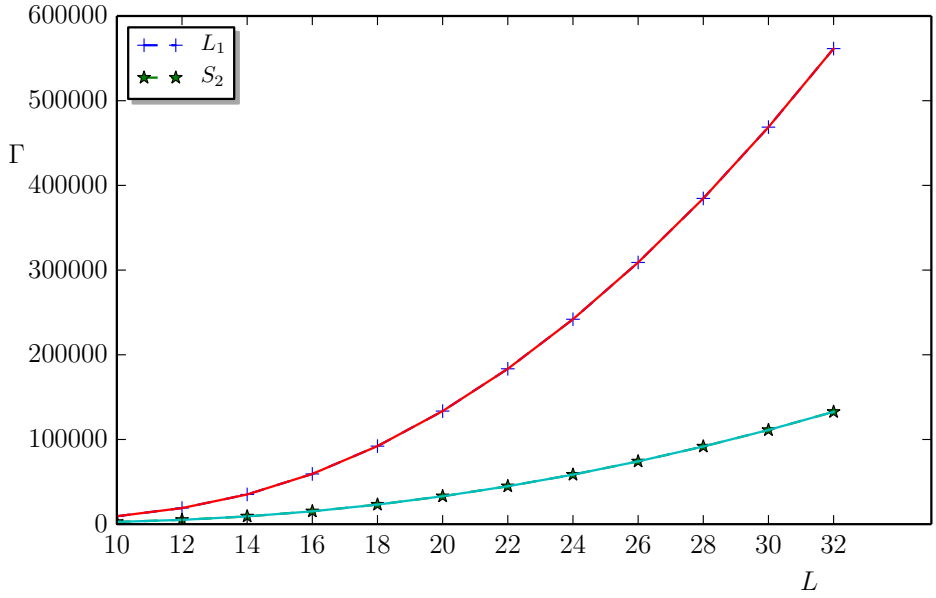


Figure 5.19: Number of BSD representations Γ versus wordlength L for the filters L_1 and S_2.

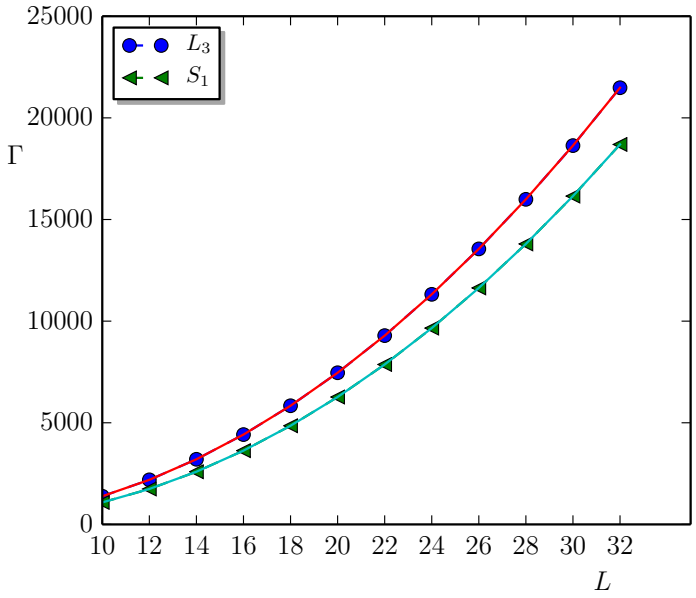


Figure 5.20: Number of BSD representations Γ versus wordlength L for the filters: L_3 and S_1.

the average number of representations and the x-axis is the digit sets radix-2 given by $\{\overline{1}, 0, 1\}$, $\{\overline{1}, 0, 1, 2\}$, $\{\overline{1}, 0, 1, 2, 3\}$, $\{\overline{1}, 0, 1, 2, 3, 4\}$, and $\{\overline{1}, 0, 1, 2, 3, 4, 5\}$. The plot shows that increasing the number of digits causes the average number of representations to increase. This is because increasing the digit set cardinality increases the branching factor.

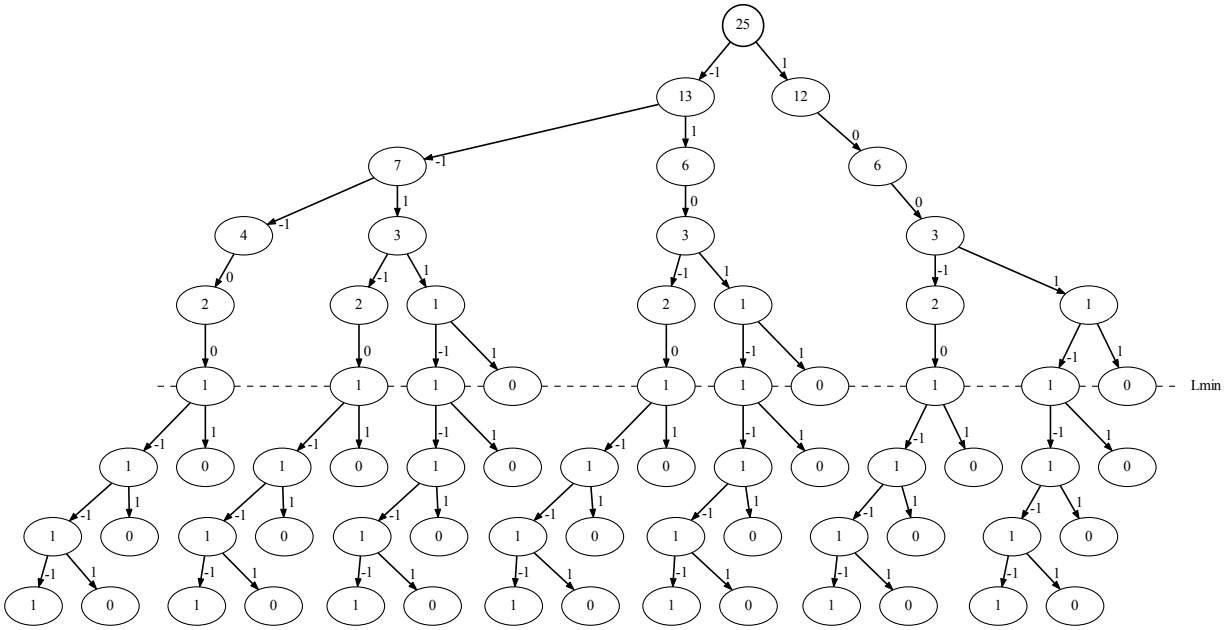


Figure 5.21: The encoding tree for the value $v = 25$ using BSD. The branches after zero nodes are pruned.

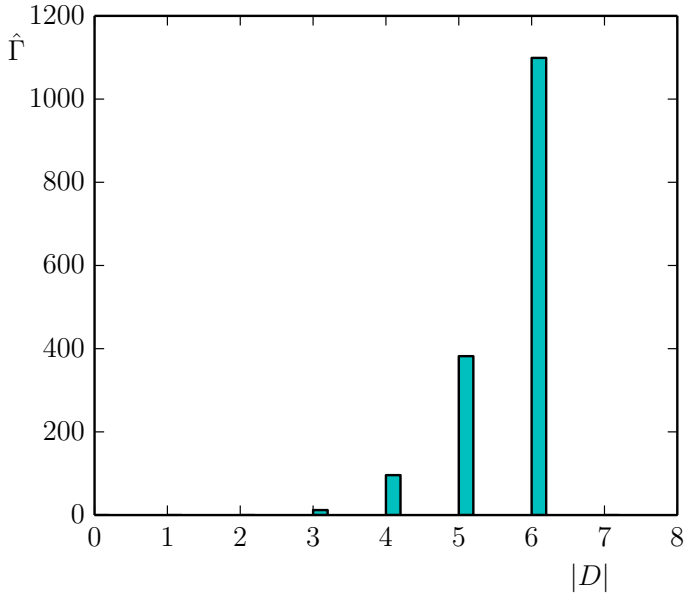


Figure 5.22: The average number of representations $\hat{\Gamma}$ versus digit set cardinality $|\mathbb{D}|$ for the values in the range $[-256, 256]$ using number systems with $r = 2$, $L = 8$, and alphabets $\{\overline{1}, 0, 1\}$, $\{\overline{1}, 0, 1, 2\}$, $\{\overline{1}, 0, 1, 2, 3\}$, $\{\overline{1}, 0, 1, 2, 3, 4\}$, and $\{\overline{1}, 0, 1, 2, 3, 4, 5\}$.

5.5 CHAPTER SUMMARY

Graph and tree algorithms are proposed to generate redundant representations. The difference between the graph and tree encoder is in their traversal. The graph encoder

is generated recursively using depth first search, while the tree encoder is generated using breadth first search. A mathematical model is derived to find the size of the tree for BSD. The size of the tree grows as a function of the growth factor which is in turn depends on the digit set partition. Tree growth in the region $L_{\min}$ is exponential on the growth factor. However, the tree may continue to grow exponentially or even linearly within $L > L_{\min}$.

Markov transition matrix $\mathbf{Q}$ [Wai-K and Michael, 2006] was used to find a closed form for the size of tree. It is found that the value of the nodes after $L > L_{\min}$ belong to a reduced set $\mathbb{S}$. The transition matrix $\mathbf{Q}$ is found with the states are the elements of the set $\mathbb{S}$. A proportion vector, $\mathbf{p}(L_{\min})$, is derived to have its elements associated with the states to calculate the states after K transitions. An attempt was made to generalize the BSD closed form to find tree size for other number systems. The generalizing is failed which indicates that each number system require finding its own parameters which are: the reduced set $\mathbb{S}$, transition matrix $\mathbf{Q}$, and proportion vector $\mathbf{p}(L_{\min})$.

The tree encoder was used to generate the BSD for a number of number bunches. Each bunch contains numbers with the same value of L_{min}. The results show that a bunch of larger numbers consumes longer run time to generate the representations of the numbers in the bunch. The tree encoder was used to find the representations of several benchmark filters that found in the literature. The results showed that larger filters consumes longer run time to generate the representations.

Tree growth after L_{min} can be slowed down by stopping the branching from nodes with zero value because it is just pad zeros to the left of a representation. Results showed that the average number of representation increases (as expected) with the number of digits in the digit set as a result of increasing the growth factor.

Chapter 6

SUBEXPRESSION TREE ALGORITHM

There are variety of common subexpression elimination (CSE) methods designed to work on subsets of the binary signed digit (BSD) representations. For example, Park and Kang [2001] proposed an algorithm to generate the minimum Hamming weight (MHW) representations from canonical signed digit (CSD). The algorithm replaces the CSD sequences of $(10\bar{1}0\bar{1}\cdots\bar{1}0\bar{1})_2$ with MHW sequences $(01010\cdots011)_2$ and the CSD sequences of $(\bar{1}0101\cdots101)_2$ with MHW sequences $(0\bar{1}0\bar{1}0\cdots0\bar{1}\bar{1})_2$. The authors showed that using MHW representations with the CSE method yields better results than that obtained from using CSD. Dempster and Macleod [2004] proposed a tree-like method to find binary signed-digit (BSD) representations. They concluded that using BSD representations improves performance of the CSE methods as compared with CSD or MHW representations. However, the search for common subexpressions occurs after BSD generation. This forces the CSE procedure to search the whole BSD space to find the best subexpression sharing. Ho et al. [2008] proposed a 0-1 mixed integer linear programming (MILP) common subexpression elimination method to optimize the MCM. In this work, the MHW set is extended to include some other representations. The authors stated that this extension improves the performance of the MILP method. However, the work did not show how to generate these additional representations. In our work [Al-Hasani et al., 2011], we proposed using a reduced search space called the zero dominant set. The space is derived by Kamp [2010] to include the MHW representations and some other representations. The other representations are obtained by giving equal scores for zero digit positions and MHW. We found that using the zero dominant set improves the CSE algorithm performance.

In this chapter we develop a common subexpression elimination method that integrates the search for best subexpression sharing with BSD generation. The method differs from the conventional approach that generates the representations first then searches for common subexpressions [Vinod et al., 2010] [Potkonjak et al., 1996] [Yu and Lim, 2009]. The proposed method uses the representation tree concept presented in Subsection 5.1 to find BSD representations of a coefficient. Possible subexpressions at each node are calculated during tree traversal. The subexpressions are used to find different decompositions for the coefficient to be encoded. This enlarges the search space and increase the possibility of finding solutions for the MCM with shorter logic depths. To reduce the search space, the algorithm terminates the tree's growth when it finds subexpressions with maximum sharing. Pruning the representation tree reduces

it to a subexpression tree and the algorithm is called the subexpression tree algorithm (STA).

This chapter is structured as follows: In Section 6.1 the concept of subexpression space is introduced including the formulation of the $\mathcal{A}-$operation [Voronenko and Püschel, 2007]. The subexpression tree algorithm is developed in Section 6.3. The results of comparing the algorithm with other algorithms are shown in Section 6.4. A summary of the chapter is given in Section 6.5.

6.1 SUBEXPRESSION SPACE

We have shown in the previous chapters that using redundant number systems provides multi-representations for a value. Each representation in turn can be considered as a set of weighted digits of cardinality equal to the representation Hamming weight, $\mathcal{H}$. Picking all the possible combinations of the weighted digits in each set forms the subexpression space. A mathematical development is introduced to determine the size of the subexpression space. Consider one of the BSD representations of 25 given by $1\bar{1}11\bar{1}\bar{1}$. This representation can be expressed by the set $P = \{2^5, -2^4, 2^3, 2^2, -2^1, -2^0\}$ of cardinality $\mathcal{H} = 6$. The number of k-combinations found in P is equal to the binomial coefficient [Roberts and Tesman, 2009],

$$\binom{\mathcal{H}}{k} = \frac{\mathcal{H}!}{k!(\mathcal{H}-k)!}. \tag{6.1}$$

Note that k is restricted to $2 \leq k \leq \mathcal{H}$, because all the single digit subexpressions are of values $\{\pm 2^j : j = 0, 1, \cdots, L-1\}$ and they evaluate to 1. The set containing all the possible subexpressions in the representation i is known as the subexpression subspace $\mathbb{S}_i$. The number of subexpressions in the set $\mathbb{S}_i$ equals its cardinality, $|\mathbb{S}_i|$, and given by

$$|\mathbb{S}_i| = \sum_{k=2}^{\mathcal{H}} \binom{\mathcal{H}}{k} + 1. \tag{6.2}$$

The value 1 is added to the sum of binomial coefficients shown in Equation 6.2 to represent the value of all the single digit subexpressions. Using the identity in [Roberts and Tesman, 2009]

$$\sum_{k=0}^{\mathcal{H}} \binom{\mathcal{H}}{k} = 2^{\mathcal{H}}, \tag{6.3}$$

and rewriting Equation 6.2 as

$$\begin{aligned} |\mathbb{S}_i| = &\ \textstyle\sum_{k=2}^{\mathcal{H}} \binom{\mathcal{H}}{k} + \binom{\mathcal{H}}{1} - \binom{\mathcal{H}}{1} + \binom{\mathcal{H}}{0} - \binom{\mathcal{H}}{0} + 1, \\ = &\ \textstyle\sum_{k=0}^{\mathcal{H}} \binom{\mathcal{H}}{k} - \binom{\mathcal{H}}{1} - \binom{\mathcal{H}}{0} + 1, \end{aligned} \tag{6.4}$$

gives a tractable expression for the cardinality of $\mathbb{S}_i$ as

$$|\mathbb{S}_i| = 2^{\mathcal{H}} - \mathcal{H}, \tag{6.5}$$

which equals the total number of subexpressions in the set $\mathbb{S}_i$.

If the number of representations for a value is B, then the resulting subexpression space is given by

$$\mathbb{S} = \mathbb{S}_1 \bigcup \mathbb{S}_2 \cdots \bigcup \mathbb{S}_B \tag{6.6}$$

6.2 $\mathcal{A}-$OPERATION

For a particular k-combination subexpression, s'_1, there must be a matching subexpression s'_2 such that the value v is given by $v = s'_1 + s'_2$. Therefore, the Hamming weight of s'_2 will be $\mathcal{H} - k$. This decomposes the value v into two additive parts. Using two part decompositions requires only a single adder to compose the value. To eliminate the redundancy in computing, each additive part should be made co-prime with the radix, or simply co-prime [Ayres and Jaisingh, 2004]. For a number system with radix r, the subexpression s is co-prime if $s \bmod r \neq 0$. Decomposing the value v into two parts is defined by an $\mathcal{A}-$operation proposed by Voronenko and Püschel [2007] for radix 2 number systems. The $\mathcal{A}-$operation is refined in this work with an algebraic development that considers radix r number systems.

Definition 6.2.1 ($\mathcal{A}-$operation)**.** An $\mathcal{A}-$operation with two co-prime positive integers s_1 and s_2 as inputs and one co-prime positive integer as an output is given by

$$\begin{aligned} \mathcal{A}_w(s_1, s_2) = & \quad |r^i s_1 \pm r^j s_2| \\ = & \quad |(s_1 \ll i) \pm (s_2 \ll j)|, \end{aligned} \tag{6.7}$$

where $\mathcal{A}_w(s_1, s_2)$ is the decomposition of w, $\ll$ denotes left shift, and $i, j \geq 0$.

The following theorem is introduced to specify the restrictions on the $\mathcal{A}-$operation to make the output always a co-prime positive integer.

Theorem 6.2.1. Each co-prime positive integer can be decomposed using two co-prime positive integers s_1 and s_2, with only one of them shifted to the left by zero or more positions.

Proof. Let $w = (s_1 r^i) \pm (s_2 r^j)$, where $w \bmod r \neq 0$, $s_1 \bmod r \neq 0$, and $s_2 \bmod r \neq 0$. If $i \neq 0$ and $j \neq 0$, then $(s_1 r^i) \bmod r = 0$ and $(s_2 r^j) \bmod r = 0$. So $((s_1 r^i) \pm (s_2 r^j)) \bmod r = (s_1 r^i) \bmod r \pm (s_2 r^j) \bmod r = 0$ and consequently $w \bmod r = 0$ which is a contradiction. Now, if $i = 0$ and $j > 0$, then $s_1 \bmod r \neq 0$ and $(s_2 r^j) \bmod r = 0$ and consequently $s_1 \bmod r \pm (s_2 r^j) \bmod r \neq 0$. The case of $i = j = 0$ requires further inspection to determine whether the sum $s_1 \pm s_2$ is co-prime or not. For BSD, a co-prime number must be odd and the summation of two odd numbers is even which is not co-prime with 2. □

Following Theorem 6.2.1 and letting $i = 0$, the $\mathcal{A}-$operation is refined to

$$\mathcal{A}_w(s_1, s_2) = |s_1 \pm r^j s_2|. \tag{6.8}$$

Since s_1 in Equation 6.8 can be found from the representation of w, the value of s_2 is calculated from Equation 6.8 as

$$\begin{aligned} s_2 \ll j &= w - s_1, \\ s_2 &= |(w - s_1) \bmod 2 \neq 0|. \end{aligned} \tag{6.9}$$

Example 6.2.1. Consider the BSD representation $1\bar{1}11\bar{1}\bar{1}$ of 25. The representation is of Hamming weight $\mathcal{H} = 6$. So the size of the resulting subspace is obtained from applying Equation 6.5

$$|\mathbb{S}_1| = \ 2^6 - 6 = 58.$$

Table 6.1 illustrates four arbitrary samples of finding the subexpressions s_1 and s_2 from the above representation. Consider the third row in the table, the computing steps are

1. s_1 is found to equal 5.
2. Using Equation 6.9, we are trying to find the smallest number that has a non zero modulo 2 result $s_2 = |(25 - 5)/4| = |20/4| = 5$.

□

Table 6.1: Subexpressions comprising of two or more digits found in one of the 25 BSD representations, $1\bar{1}11\bar{1}\bar{1}$. s_1' and s_2' are the raw subexpressions. s_1 and s_2 are positive co-prime subexpressions.

BSD	$s_1' \rightarrow s_1$	$s_2' \rightarrow s_2$
$1\bar{1}11\bar{1}\boxed{\bar{1}}$	$-1 \rightarrow 1$	$26 \rightarrow 13$
$1\bar{1}11\boxed{\bar{1}\bar{1}}$	$-3 \rightarrow 3$	$28 \rightarrow 7$
$1\bar{1}\boxed{1}1\boxed{\bar{1}\bar{1}}$	$5 \rightarrow 5$	$20 \rightarrow 5$
$1\boxed{\bar{1}}11\boxed{\bar{1}}\bar{1}$	$-18 \rightarrow 9$	$43 \rightarrow 43$
	⋮	

6.3 SUBEXPRESSION TREE ALGORITHM

In this section, the subexpression tree algorithm (STA) is developed. The STA algorithm customizes the tree algorithm to search for CSE. At the beginning of the algorithm, the coefficients are made positive co-prime numbers with respect to the radix. The resulting co-prime coefficients are classified according to their complexity. Low complexity coefficients are that with the lowest Hamming weight and shortest wordlength. The algorithm starts looping over the coefficients by picking one coefficient

at a time and generate the representation tree for it. The looping over the coefficients can be started from coefficients of maximum complexity then descend to coefficients of minimum complexity. The is is called a descending search. The other direction of looping over the coefficients starts from coefficients with minimum complexity towards that of maximum complexity. This is called an ascending search. It determines all possible decompositions of the value to be encoded at each new generated node by using released digit. This enabled checking the subexpressions at each depth to find if they were used earlier to synthesize other coefficients. Optimal sharing is obtained if such subexpressions are found. At this point, the algorithm stops from generating the rest of the representation tree which makes it a subexpression tree. If the algorithm couldn't find optimal sharing, it determine the best sharing or maximum sharing that can be obtained from share the subexpressions with other coefficients without doing any synthesizing. At the end of generating the trees of subexpressions and coefficients, the algorithm search shared sets of subexpressions for the one of minimum cost but can realize maximum number of coefficients. If there are still unsynthesized coefficients, the algorithm starts looping over these coefficients in a direction that reverse to the first loop.

6.3.1 Subexpressions Determination

The tree algorithm described in Section 5.1 can be enhanced to simultaneously search for the possible subexpressions in each representation. New subexpressions are computed at each newly generated node and added to the parent subexpression set. Each node has the following attributes: Node value, subexpression set S, decomposition pairs, and subexpression subspace $\mathbb{S}$. Computation complexity can be reduced by letting each child node inherit all of its parents' attributes. So the algorithm tracks the generation of s_1 only by adding a released weighted digit to each s_1 that come from the parent. In other words, the subexpression subspace that is formed from an incomplete representation is

$$S_c = S_p \bigcup \{r^l \times d\} \bigcup \{s_1 + r^l \times d, \forall s_1 \in S_p\}, \tag{6.10}$$

where l is the tree depth, d is the edge connecting child node with its parent, and S_p and S_c are the parents' and childs' subexpression sets, respectively. To illustrate the process, consider the red branch of the tree of 281 shown in Figure 5.3. This branch is redrawn separately as shown in Figure 6.1. Each subexpression set S_l^n is calculated using Equation 6.10, where l is the tree depth and n is the rank of nodes at this depth. The complete representation at the leaf node should be 100011001. However, the whole representation is unknown when the algorithm starts generating the nodes. The algorithm starts from the root 281 with an empty set of attributes. The LSD digit 1 is assigned to s_1 forming the subexpression set $S_0^0 = \{1\}$. The subexpression s_2 is computed to be 35 using Equation 6.9, as shown in step-II. The resulting decomposition consists of the pair $(1, 35)$ which is stored in the attributes of node 140 (step-III). Similarly the subexpression space in step-IV, $\mathbb{S}_0^0$ is stored in node 140 attributes.

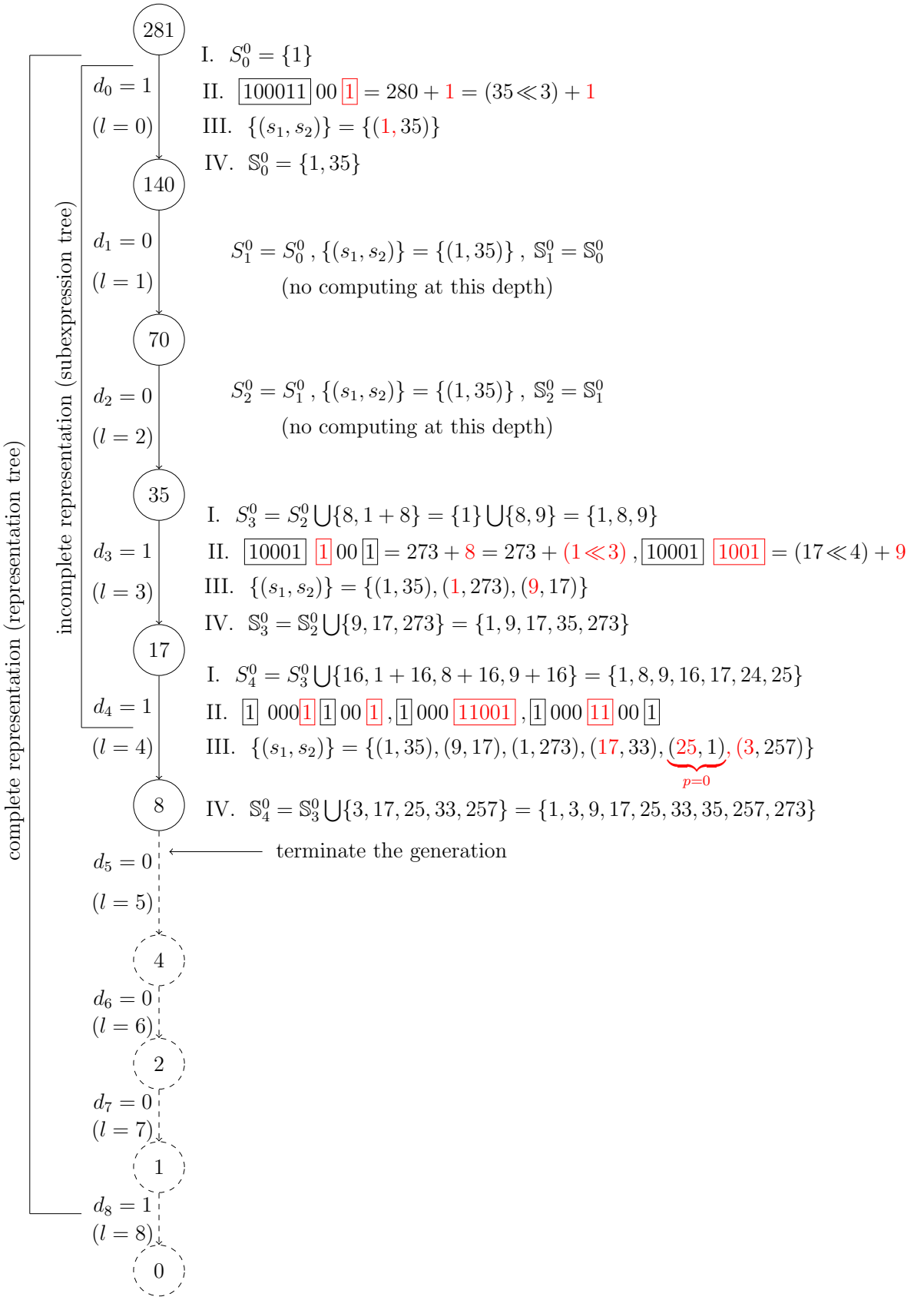


Figure 6.1: A branch of the representation tree of 281 results in a non-exhaustive search.

The next positional digit is 0 and released at depth $l = 1$ of the tree. A zero digit value has no effect on the parent attributes according to Equation 6.10. So node 70 attributes are that of the parent and no subexpressions are computed at this depth as shown in Figure 6.1. Similarly at depth $l = 2$, only parent attributes are inherited.

At depth $l = 3$, the released digit 1 is of weight 8. The weighted digit 8 is added to each subexpression value that comes from the parent as shown in step-I of depth $l = 3$ in Figure 6.1. The subexpression set is $S_0^3 = \{1, 8, 9\}$. No new decompositions can be obtained from the subexpressions 1 and 8. Only the subexpression $s_1 = 9$ is used to find a decomposition. So, the value of $s_2 = 17$ is computed using Equation 6.9 as illustrated in step-II of depth $l = 3$ in Figure 6.1. The new decomposition pair $(9, 17)$ is stored in the decomposition pair set as shown in step-III. However, the subexpression subspace contains all the values of s_1 and s_2 (step-IV).

Repeating the same procedure for depth $l = 4$ results in finding three new decompositions as shown in step-II of Figure 6.1. The corresponding decomposition pairs are $(17, 33)$, $(25, 1)$, and $(3, 257)$ as shown in step-III. Tree generation can be terminated at this point because the decomposition $(25, 1)$ is optimal. The decomposition $(25, 1)$ is considered optimal because both of its subexpressions are of zero cost. The subexpression 1 is already in the synthesized coefficient set and the subexpression 25 is in the coefficient set and will not require any additional cost over its inevitable synthesizing. This is known as a nonexhaustive algorithm because the entire representation tree has not been generated. If we do proceed to find the entire encoding of the representation then we will have performed an exhaustive search.

6.3.2 Decomposition Complexity

We have shown in Section 6.1 that using redundant representations results in many decompositions (subexpression pairs). Decompositions with maximum sharing among the coefficients are preferred. These decompositions are proposed to be of minimum complexity to guide the algorithm to find them. So, the complexity of a decomposition is based on, in descending order of importance, the priority, Hamming weight, and wordlength. The priority metric favours decompositions with maximum sharing. If there is no such decompositions, the algorithm searches for ones with minimum adder step using the Hamming weight as a metric. Similarly, subexpressions with shorter wordlength are preferred to reduce adder width.

The complexity measure of the decomposition $\chi(\mathcal{A}_w(s_1, s_2))$ can be defined as

$$\chi(\mathcal{A}_w(s_1, s_2)) = c \times p + b \times \mathcal{H}_s + L_s, \tag{6.11}$$

where c and b are weighting factors, p is the priority of $\mathcal{A}_w(s_1, s_2)$, and

$$\mathcal{H}_s = \max(\mathcal{H}_{s_1}, \mathcal{H}_{s_2}), \tag{6.12}$$

is the maximum Hamming weight of the subexpression pairs (s_1, s_2), and

$$L_s = \max(L_{s_1}, L_{s_2}), \tag{6.13}$$

is the maximum wordlength of (s_1, s_2). The priority value, p, is selected according to Table 6.2 to increase the possibility of finding an early solution without exploring the whole space. Preference is given to the synthesized subexpressions and to the common subexpressions to minimize the number of logic operators. One choice to achieve this is:

$$b = L_{\max} + 1. \tag{6.14}$$

The weight c converts the high priority information to low complexity. If the algorithm finds two decompositions with the same priority, it choses the one with the smallest Hamming weight to minimize the LO and LD. If the Hamming weights are the same, the decomposition with shorter wordlength is preferred. One choice of c to achieve this is:

$$c = b \times \mathcal{H}_{\max}, \tag{6.15}$$

where $\mathcal{H}_{\max}$ is the maximum Hamming weight that can be found in the representations. This can be taken to be equal the largest wordlength in the coefficients.

Example 6.3.1. Consider realizing the filter F_1 in [Farahani et al., 2010] with coefficient set $W = \{25, 103, 281, 655\}$. The synthesised coefficient set is initialized to $T = \{\}$, while the shared subexpression set is $S_s = \{\}$. The maximum wordlength is that of the coefficient 655 and equal to $L_{\max} = 10$, making $\mathcal{H}_{\max} = 10$ (i.e. the representation $111\bar{1}\bar{1}\bar{1}\bar{1}11$). Using Equation 6.14 and Equation 6.15 the weights are $b = 11$ and $c = 110$.

Since there are no coefficients with $\mathcal{H} = 2$, the algorithm tries to explore the subexpression space in descending order. This increases the possibility of finding subexpressions with $\mathcal{H} = 2$ and maximum sharing. The algorithm starts generating the representation tree of the coefficient 655. The complexity value, $\chi(\mathcal{A}_w(s_1, s_2))$, of each

Table 6.2: Priority values for the subexpression tree algorithm. W is the set of coefficients and subexpressions (fundamentals) waiting for synthesis, T is the set of synthesized fundamentals, and S_s is the set of shared (common) subexpressions. A low priority number is more desirable.

s_1, s_2	p
$s_1, s_2 \in W \bigcup T \bigcup S_s$	0
$s_1 \in W \bigcup T \bigcup S_s, s_2 \notin W \bigcup T \bigcup S_s$	1
$s_1 \notin W \bigcup T \bigcup S_s, s_2 \in W \bigcup T \bigcup S_s$	1
$s_1 = s_2 \notin W \bigcup T \bigcup S_s$	1
$s_1 \neq s_2 \notin W \bigcup T \bigcup S_s$	2

decomposition pair (s_1, s_2) is calculated according to Equation 6.11. These values are compared during tree growth to delete the pairs with highest complexity. Table 6.3 shows samples of decomposition complexity calculations for the coefficient 655. It is found that the decomposition $\chi(\mathcal{A}_{655}(15,5))$ has minimum complexity (shown in row 2 of Table 6.3). The decomposition pair $(15,5)$ is moved to the set S_s. Then, the representation tree of 281 is generated as shown in Figure 5.3. The decomposition $\chi(\mathcal{A}_{281}(25,1))$ is found at depth $l = 4$ with priority $p = 0$ (as described in Section 6.3.1). Generating the tree of 103 results in finding the decomposition $\chi(\mathcal{A}_{103}(25,1))$ at depth $l = 4$ with $p = 0$. The last tree generated is that of 25. The decomposition $\chi(\mathcal{A}_{25}(5,5))$ is found at depth $l = 3$ with priority $p = 0$ as shown in row 4 of Table 6.3. This decomposition is with maximum priority value because the subexpression 5 is already in the sharing set S_s. The resulting filter realization is shown in Figure 6.2.

Table 6.3: Samples of the decomposition complexity values obtained from the representation tree of 655.

w	(s_1, s_2)	CSD(s_1)	CSD(s_2)	$\mathcal{H}_s$	L_s	p	$\chi(\mathcal{A}_w(s_1, s_2))$
655	$(7, 81)$	$100\bar{1}$	1010001	3	7	2	$110 \times 2 + 11 \times 3 + 7 = 260$
655	$(15, 5)$	$1000\bar{1}$	101	2	5	2	$220 + 11 \times 2 + 5 = 247$
25	$(1, 3)$	1	$10\bar{1}$	2	3	1	$110 + 11 \times 2 + 3 = 135$
25	$(5, 5)$	101	101	2	3	0	$11 \times 2 + 3 = 25$

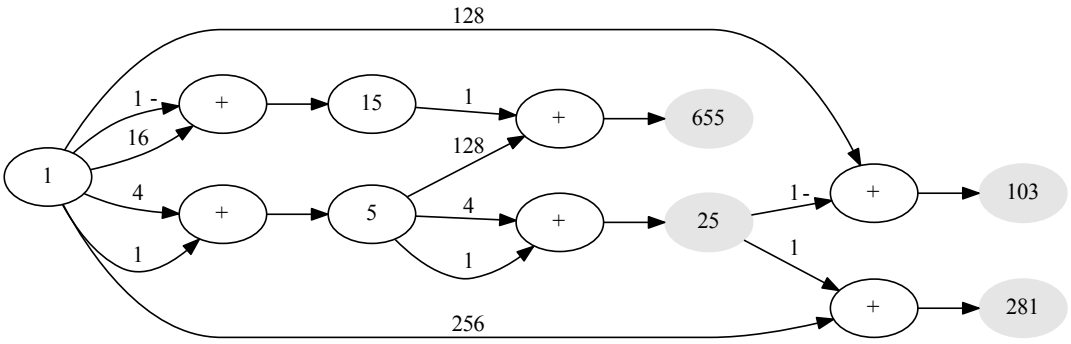


Figure 6.2: Realization of the filter F_1 in [Farahani et al., 2010] with coefficient set $W = \{25, 103, 281, 655\}$ using the STA.

□

The listing of the subexpression tree algorithm (STA) is shown in Listing 3. The algorithm generates the representation tree for each coefficient in breadth first manner (line 4 of Listing 3). Hence, decompositions with $p = 0$ could be found at shallow depth as shown in line 12 otherwise the algorithm continues finding new decomposition pairs (line 10). These decompositions form the subexpression space given by the set S_s in line 21. Listing 4 shows other modules used in conjunction with the STA algorithm. The Sort

function in line 26 of the main program sorts the coefficients in ascending/descending order. The set S_s in line 11 is the set of subexpressions.

```
1:  procedure STA(w, W, T, S_s)
2:      χ'_A ← ∞ , S' ← {} ▷ Initiate minimum complexity and subexpression subspace
3:      Tree ← {(root.v ← w, root.s ← {})}                          ▷ Tree node list
4:      for l ← 0 to w.L do          ▷ Generate the tree in a breadth-first manner
5:          C ← {}                                                   ▷ Empty child list
6:          for each node ∈ Tree do
7:              for each d ∈ {m : m ≡_2 (node.v); m ∈ {1̄, 0, 1}} do ▷ Partition digit set
8:                  child ← NewNode(node, d, l)                      ▷ New child is born
9:                  for each s_1 ∈ child.s do
10:                     s_2 ← |(w − s_1) mod 2 ≠ 0|   ▷ Find the decomposition of w as
                                                          in Equation 6.9
11:                     p, χ(A_w) ← Complexity(w, s_1, s_2, W, T, S_s) ▷ Find the complexity
                                                          as in Equation 6.11 and Table 6.2
12:                     if p = 0 then                                ▷ Optimum case
13:                         return Synthesise(w, s_1, s_2, W, T, S_s)   ▷ Prune the tree
14:                     else if χ(A_w) < χ'_A then                   ▷ Compare the complexity
15:                         χ'_A ← χ(A_w)                            ▷ Best minimum found yet
16:                         S' ← {s_1, s_2}
17:                     else if χ(A_w) = χ'_A then
18:                         S' ← S' ⋃ {s_1, s_2}                     ▷ Trim the subexpression subspace
19:                 C ← C + {child}                                  ▷ Concatenate child list
20:         Tree ← C                                                 ▷ Next generation
21:     S_s ← S_s ⋃ S'                          ▷ Minimum complexity subexpression space
22:     return W, T, S_s
23:
24: procedure NewNode(parent, d, l)                                  ▷ Child node value
25:     child.v ← (parent.v − d)/2
26:     child.s_1 ← parent.s_1 ⋃ {r^l × d} ⋃ {s_1 + 2^l × d : s_1 ∈ parent.s_1}   ▷ Update child
                                                          subexpression set as in Equation 6.10
27:     return child
```

Listing 3: Subexpression tree algorithm (STA).

6.3.3 Multiple Iterations

The coefficients can be synthesised either in ascending or descending order of complexity. The complexity $\chi(w)$ of the coefficient w is defined as

$$\chi(w) = b \times \mathcal{H}_w + L_w, \tag{6.16}$$

where $\mathcal{H}_w$ is the Hamming weight of w, L_w the wordlength of w, and b is given in Equation 6.14. The search in ascending order is preferred if there are coefficients with $\mathcal{H} = 2$. Here, the algorithm tries to maximize coefficient sharing without violating the logic depth constraint. Synthesis in descending order is used when the MCM misses coefficients with $\mathcal{H} = 2$. This search type enables the algorithm to explore a larger space to find subexpressions with $\mathcal{H} = 2$ and maximum sharing.

```
1:  procedure Synthesise(w, s1, s2, W, T, Ss)
2:      if s1 ∉ W and s1 ∉ T then
3:          W ← W ⋃ {s1}                          ▷ Prepare s1 for synthesis
4:      if s2 ∉ W and s2 ∉ T then
5:          W ← W ⋃ {s2}                          ▷ Prepare s2 for synthesis
6:      W ← W \ {w}                               ▷ Remove w from W
7:      T ← T ⋃ {w}                               ▷ w is synthesised
8:      Ss ← Ss \ {s1, s2}                        ▷ Remove s1 and s2 from Ss
9:      return W, T, Ss

10: main
11: input H, LD                                   ▷ Input coefficient set and logic depth
12: T ← {1}                                       ▷ Synthesised fundamental set
13: W ← Prepare(H)                                ▷ Make coefficients unique
14: W, ord ← Sort(W, ord = True)                  ▷ Descending order
15: for i ∈ [0, 1] do
16:     for each w ∈ W do                         ▷ One iteration
17:         W, T, Ss ← STA(w, W, T, Ss)
18:     if W = {} then
19:         Break
20:     W, T ← BestShare(Ss)                      ▷ Best shared subexpressions
21:     W, ord ← Sort(W, ord)                     ▷ Reverse W order
22: end main
```

Listing 4: Modules of the STA algorithm.

When the STA iterates over the coefficients in ascending order it is denoted by STA^{a} if it has only one iteration and by STA^{aa} for two ascending iterations. Similarly, it is denoted by STA^{d} and STA^{dd} for one and two iterations in descending order. Mixed iterations are also possible, hence the algorithm is denoted by STA^{ad} if the first iteration in ascending order and the second in descending. It is denoted by STA^{da} for iterating in descending order then in ascending.

6.3.4 Search Space Size

The subexpression space that derived in Section 6.3.1 considers finding the number of subexpressions in the coefficient representations. This might be not the case in the STA algorithm. The STA used to generate the representation tree for coefficients and calculate the decompositions along all the branches whether the latter ended with zero node (a representation) or not. On the other hand, the algorithm is used to terminate tree generation when it finds a decomposition with $p = 0$. This makes determining the size of search space a hard task. The worst situation occurs when the algorithm generates the whole representation trees for all the coefficients. The number of branches that contribute the search space equals to the number of leaf nodes (with any value). The worst case of search space size can be found from Equation 6.6 which is derived to find the space that results from B representations of arbitrary value. In this case the

subspace results from generating the representation tree of coefficient w_i is given by:

$$\mathbb{S}_{w_i} = \mathbb{S}_{<i,0>} \bigcup \mathbb{S}_{<i,1>} \cdots \bigcup \mathbb{S}_{<i,M_{<w_i,L>}>} \tag{6.17}$$

where $\mathbb{S}_{w_i}$ is the subexpression subspace generated by the tree of coefficient w_i, $\mathbb{S}_{<i,j>}$ is the subspace contribution of branch j of w_i tree, and $M_{<w_i,L>}$ is the number of leaf nodes at depth L of the tree of w_i. The set union operation shown in Equation 6.17 implies removing any duplication in the subexpressions of the branches. A hypothetical worst case is obtained if:

$$\mathbb{S}_{<i,0>} \bigcap \mathbb{S}_{<i,1>} \cdots \bigcap \mathbb{S}_{<i,M_{<w_i,L>}>} = \emptyset \tag{6.18}$$

In this case, the number of subexpressions in the space $\mathbb{S}_{w_i}$ is obtained from using Equation 6.5 and Equation 6.17 as:

$$|\mathbb{S}_{w_i}| = [(2^{\mathcal{H}_{<w_i,0>}} - \mathcal{H}_{<w_i,0>}) + (2^{\mathcal{H}_{<w_i,1>}} - \mathcal{H}_{<w_i,1>}) + \cdots + (2^{\mathcal{H}_{<w_i,M_{<w_i,L>}>}} - \mathcal{H}_{<w_i,M_{<w_i,L>}>})] \tag{6.19}$$

where, $M_{<w_i,L>}$ is the number of leaf nodes at depth L of the tree of the coefficient w_i, and $\mathcal{H}_{<w_i,j>}$ is the Hamming weight of the branch ranked j in the tree of the coefficient w_i. For a MCM with N coefficients, the space results from generating all the representation trees of coefficients is given by:

$$\bigcup_{i=0}^{N-1} \mathbb{S}_{w_i} = \mathbb{S}_{w_0} \bigcup \mathbb{S}_{w_1} \cdots \bigcup \mathbb{S}_{w_{N-1}} \tag{6.20}$$

A hypothetical worst case is obtained if:

$$\mathbb{S}_{w_0} \bigcap \mathbb{S}_{w_1} \cdots \bigcap \mathbb{S}_{w_{N-1}} = \emptyset \tag{6.21}$$

In this case, the number of overall subexpressions is obtained from using Equation 6.19 and Equation 6.20 as:

$$\begin{aligned} \sum_{i=0}^{N-1} |\mathbb{S}_i| = {} & [(2^{\mathcal{H}_{<w_0,0>}} - \mathcal{H}_{<w_0,0>}) + (2^{\mathcal{H}_{<w_0,1>}} - \mathcal{H}_{<w_0,1>}) \\ & + \cdots + (2^{\mathcal{H}_{<w_0,M_{<0,L>}>}} - \mathcal{H}_{<w_0 M_{<0,L>}>})] \\ + \cdots + [(2^{\mathcal{H}_{<w_{N-1},0>}} - \mathcal{H}_{<w_{N-1},0>}) & + (2^{\mathcal{H}_{<w_{N-1},1>}} - \mathcal{H}_{<w_{N-1},1>}) \\ & + \cdots + (2^{\mathcal{H}_{<w_{N-1},M_{<N-1,L>}>}} - \mathcal{H}_{<w_{N-1} M_{<N-1,L>}>})] \end{aligned} \tag{6.22}$$

where, $M_{<i,L>}$ is the number of leaf nodes at depth L of the tree of the coefficient w_i, and $\mathcal{H}_{<i,M_i>}$ is the Hamming weight of the representation ranked M_j in the tree of the coefficient i.

The actual size of search space is expected to be smaller than the worst case shown in Equation 6.22 because of the duplication in the subexpressions. In addition, the

STA used to trim the representation tree when it find a decomposition of optimal value. Therefore the STA algorithm used to reduce the subexpression space during the iteration over the coefficients.

6.4 RESULTS

The STA algorithm was implemented using Python language. The resulting code was run on a platform with the following specifications: 3.6 GHz Quad Core CPU with 8 GB RAM. The exhaustive and non-exhaustive algorithms were compared to synthesise several FIR filters found in the literature. These filters are: S_2 (as given in Example 2 of Samueli [1989]), L_1, L_2 (as given in Examples 1 and 2 of Lim and Parker [1983]), and D (as given in Dempster and Macleod [1995]). These filters are all symmetric with lengths 60, 121, 63, and 25, respectively. Tests were also conducted against randomly generated coefficients. The resulting logic depth (LD) and logic operators (LO) for the filters were compared. The exhaustive and non-exhaustive algorithms both show similar performance levels. Since the execution time for the former is usually longer, it is sufficient to use only the non-exhaustive algorithm. It was also found that it is sufficient to use either STA^{ad} or STA^{da}. However, the compromising between them depends on the search space. If the whole subexpression space is generated, the algorithm STA^{ad} was found to be faster than STA^{da} for some filters and the reverse was correct for other filters. In addition, the execution time for some filters was impractical. So the search space is reduced as shown in steps 14-18 of Listing 3 with only minimum complexity decomposition pairs kept. However, there was some difference in the performance of the two algorithms in the case of reducing the search space, with the longest execution time found with the algorithm STA^{da}. Since this choice of iteration sequence always ensures finding both minimum LO and LD so it will be used in the rest of this section. Other possibilities were introduced to explore the characteristics of the algorithm.

The STA algorithm was compared with the algorithms (MITM) in [Farahani et al., 2010], difference based adder graph (DBAG) in [Gustafsson, 2007a], and Yao in [Yao et al., 2004], as shown in Table 6.4 and Figure 6.3. The two algorithms DBAG [Gustafsson, 2007a], and Yao [Yao et al., 2004] were implemented in this work using python language and the same platform described above. The filters F_1 and F_2 [Farahani et al., 2010] each with order $N = 4$ are used in the comparison. The results for filter F_1 show that the STA^{da} algorithm outperforms the MITM in both LD and LO, DBAG in LD, and Yao in both LD and LO. For the filter F_2, the STA^{da} algorithm outperforms the MITM in both LD and LO, DBAG in LD, and results the same LO and LD of Yao. The time comparison in Table 6.4 shows that the STA^{da} algorithm requires more time than the DBAG and Yao methods. However, the former can find realizations with minimum LD because it searches a larger space of subexpressions.

The other comparison was between the STA^{da} and the algorithms Tsao [Tsao and Choi, 2010], BCSE [Smitha and Vinod, 2007], DBAG [Gustafsson, 2007a], and Yao [Yao et al., 2004]. The algorithms synthesise the filters S_2 and L_1 and the results are shown in Table 6.5. The results again show that the proposed algorithm can find realizations

Filter	N	MITM		DBAG			Yao			STA^{da}		
		LD	LO	LD	LO	t(s)	LD	LO	t(s)	LD	LO	t(s)
F_1	4	3	9	4	6	0.04	3	7	0.1	3	6	0.78
										2	7	0.54
F_2	4	5	12	5	6	0.17	3	6	0.17	3	6	0.76

Table 6.4: A comparison between the STA^{da} algorithm and the algorithms MITM [Farahani et al., 2010], DBAG [Gustafsson, 2007a], and Yao [Yao et al., 2004] to realize the filters F_1 and F_2 in Farahani et al. [2010].

with shorter logic depth than the others but it consumes more time.

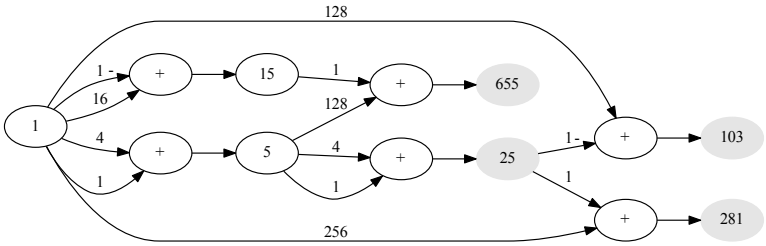
Filter	N	Tsao		BCSE		DBAG			Yao			STA^{da}		
		LD	LO	LD	LO	LD	LO	t(s)	LD	LO	t(s)	LD	LO	t(s)
S_2	60	3	32	4	29	5	26	4.6	3	26	2.8	3	26	6
									2	28	1.6	2	28	3.2
L_1	121	5	58	4	58	6	52	23	4	52	18	4	52	1100
									3	54	20.14	3	53	640

Table 6.5: A comparison between the STA^{da} algorithm and the algorithms Tsao [Tsao and Choi, 2010], BCSE [Smitha and Vinod, 2007], DBAG [Gustafsson, 2007a], Yao [Yao et al., 2004] to realize the filters S_2 and L_1.

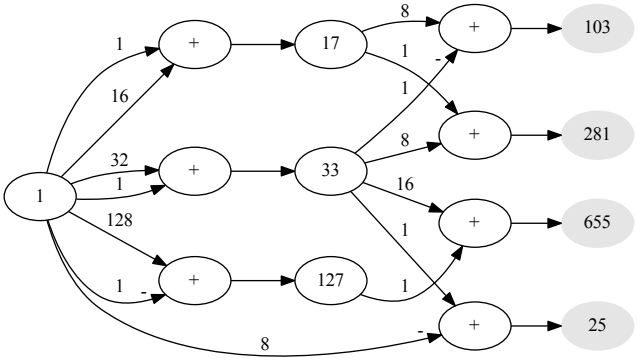
The STA was compared with the algorithms RAG-n [Dempster and Macleod, 1995], C1 [Dempster et al., 2002], MILP [Ho et al., 2008], and DBAG [Gustafsson, 2007a] to realize the filter D [Dempster and Macleod, 1995] with order $N = 25$. The results of this comparison are shown in Table 6.6. In this case, the STA also has a better performance than other algorithms in finding minimum logic depth realization but require longer time.

Filter	N	RAG-n		C1		Yao		MILP		DBAG			STA^{da}		
		LD	LO	LD	LO	LD	LO	LD	LO	LD	LO	t(s)	LD	LO	t(s)
D	25	9	18	4	19	4	18	3	18	7	17	84	3	18	242

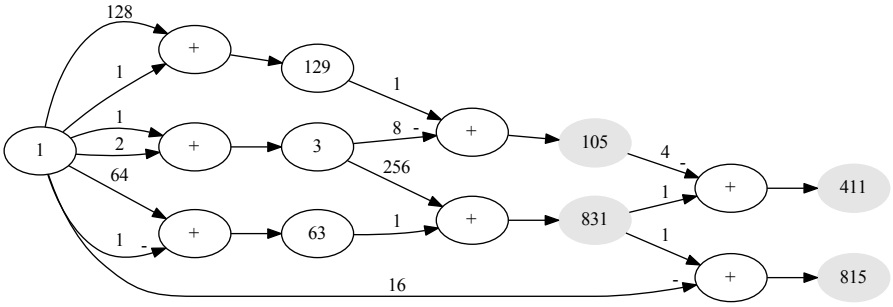
Table 6.6: A comparison between the algorithms RAG-n [Dempster and Macleod, 1995], C1 [Dempster et al., 2002], Yao [Yao et al., 2004], MILP [Ho et al., 2008], DBAG [Gustafsson, 2007a], and STA^{da} to realize the filter D in [Dempster and Macleod, 1995] with $N = 25$.



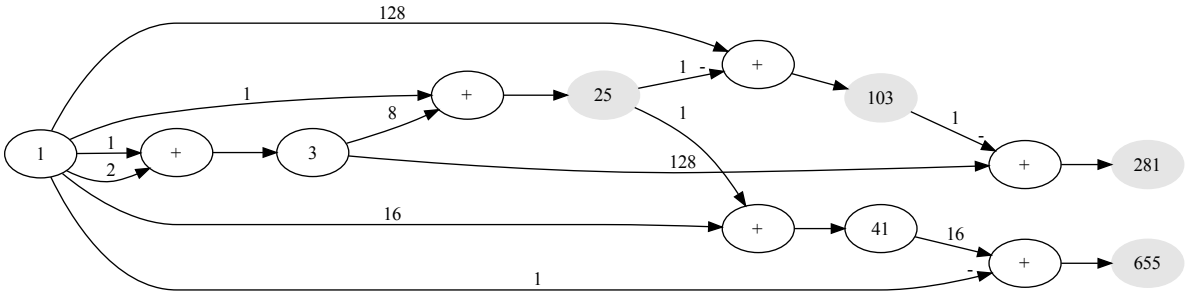
(a)



(b)



(c)



(d)

Figure 6.3: The synthesis of the filter F_1 with coefficients $\{25, 103, 281, 655\}$ using (a) STA^{da} algorithm with LD = 3 results LO = 6. (b) STA^{da} algorithm with LD = 2 results LO = 7. (c) Yao [Yao et al., 2004] with LD = 3 results LO = 7. (d) DBAG [Gustafsson, 2007a] results LO = 6 and LD = 4.

Another set of comparisons was made to estimate the performance of the STA. In this case, a set of 100 MCM each of length $N = 5$ was used to compare the STA with the algorithms PPA, DBG [Gustafsson, 2007a], and Yao [Yao et al., 2004]. A minimum LD constraint was at the top of the problem in the case of STA and Yao algorithms. The DBG and PPA are designed to minimize the LO at the expense of LD. The result of comparing the LO that shown in Figure 6.4 shows the DBAG has better adder saving than other algorithms. The STA and PPA have better adder saving than Yao. The STA performance in terms of LD is the same as Yao but it is better than the other algorithms as shown in Figure 6.5. The result of time complexity of the algorithms is shown in Figure 6.6. The PPA shows a worst case of run time because of searching all the possible combinations of Hartley's table. Yao algorithm shows a best performance in terms of run time, while STA and DBAG are of same performance.

The same experiments are repeated for another set of 100 MCM but of length $N = 10$. The PPA algorithm excluded from comparison because of its long run time. The STA performance in terms of adder saving is found similar to DBAG as shown in Figure 6.7. Both algorithms have better adder saving performance than Yao. The STA performance in terms of LD is better than DBAG as shown in Figure 6.8. In this comparison, Yao algorithm is excluded because it has a similar performance like STA and their curves are coincided exactly. The result of comparing both the LO and LD shows in this time the STA outperforms all algorithms including DBAG. The improvement is due to increase the coefficient set cardinality which increase the size of search space as shown in Equation 6.22 which increase the chance of optimizing both LO and LD simultaneously. The time complexity comparison shown in Figure 6.9 shows that Yao is of better performance than DBAG. The STA starts rising faster than DBAG once $L > 15$.

The STA was compared with the algorithm Hcub [Voronenko and Püschel, 2007] that argued as one of the best GD methods [Voronenko and Püschel, 2007]. A website named Spiral Multiplier Block Generator uses the Hcub algorithm to synthesize MCMs up to 32-bit and 20 tap. Spiral website is `http://spiral.ece.cmu.edu/mcm/gen.html`. It is found for these samples that the STA can beat the Hcub though it is supposed to find the optimal solution for any MCM [Voronenko and Püschel, 2007].

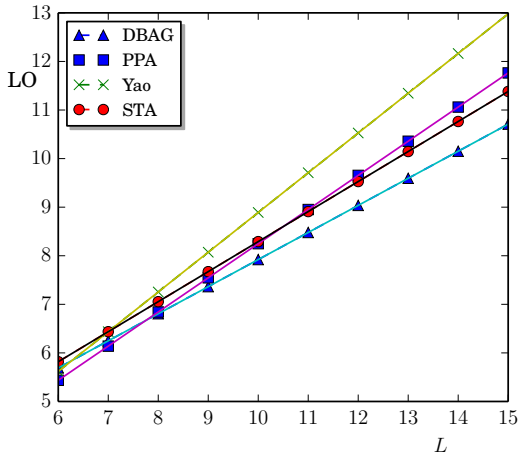


Figure 6.4: A comparison between the number of adders that obtained from using the algorithms DBAG [Gustafsson, 2007a], PPA, Yao [Yao et al., 2004], and STA used to synthesize random MCMs of 5 coefficients.

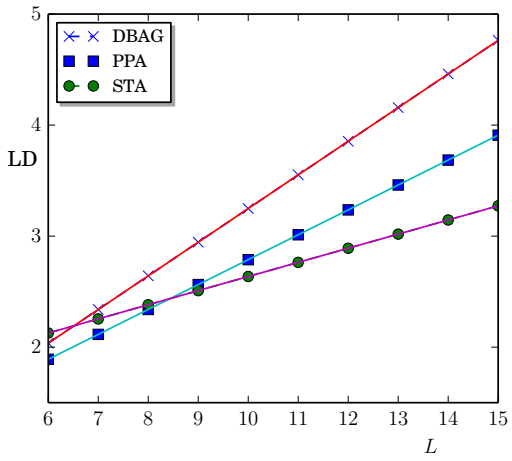


Figure 6.5: A comparison between the LD's that obtained from using the algorithms DBAG [Gustafsson, 2007a], PPA, Yao [Yao et al., 2004], and STA used to synthesize random MCMs of 5 coefficients.

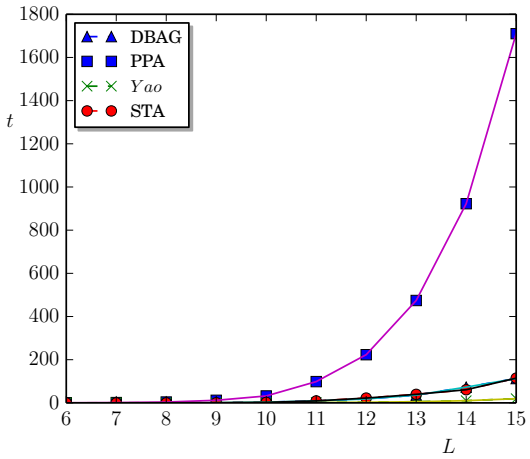


Figure 6.6: The run time comparison between the algorithms DBAG [Gustafsson, 2007a], PPA, Yao [Yao et al., 2004], and STA used to synthesize random MCMs of 5 coefficients.

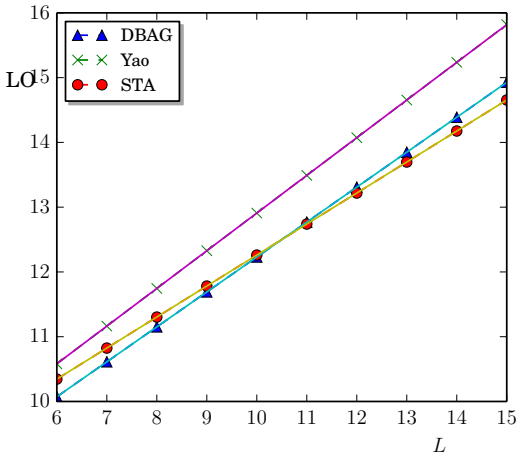


Figure 6.7: A comparison between the number of adders that obtained from using the algorithms DBAG [Gustafsson, 2007a], Yao [Yao et al., 2004], and STA used to synthesize random MCMs of 10 coefficients.

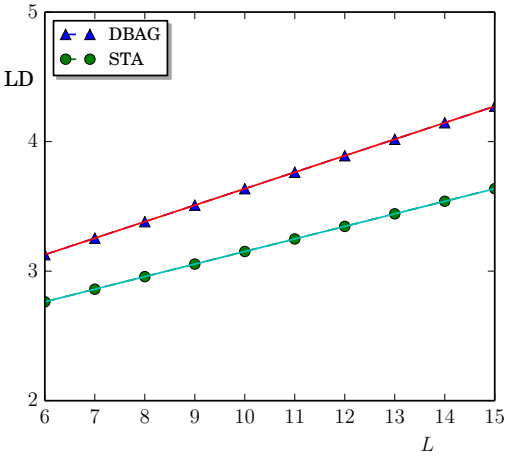


Figure 6.8: A comparison between the LD's that obtained from using the algorithms DBAG [Gustafsson, 2007a], and STA used to synthesize random MCMs of 10 coefficients. Yao [Yao et al., 2004] and STA have the same LD.

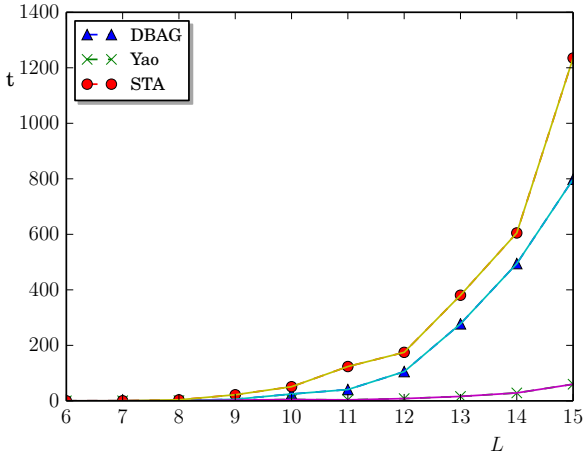


Figure 6.9: The run time comparison between the algorithms DBAG [Gustafsson, 2007a], Yao [Yao et al., 2004], and STA used to synthesize random MCMs of 10 coefficients.

SPIRAL
Software/Hardware Generation for DSP Algorithm

Home Generator Benchmarks Publications Software Hardware

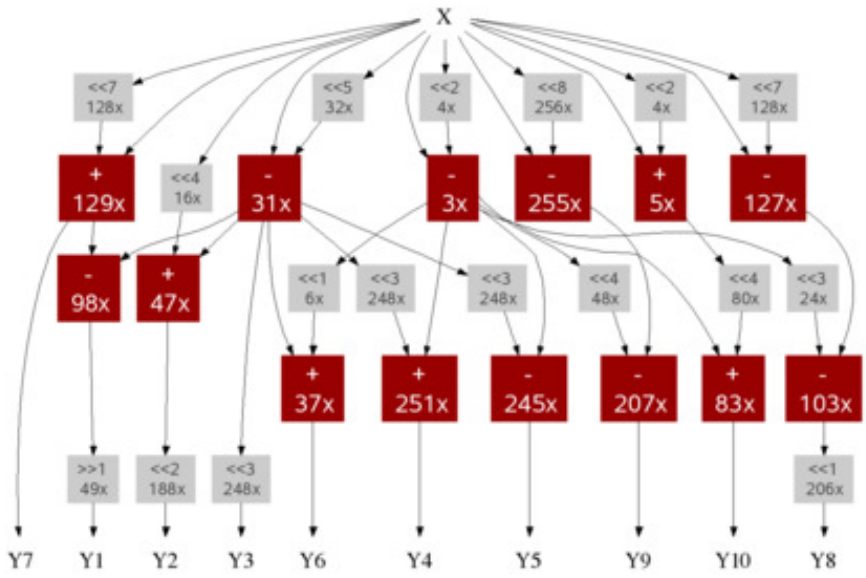
Return to the Multiplier Block Generator.

Output

Algorithm: Hcub
Auxiliary distance estimate: CSD-Cost(z)
Depth bound: 0
Secondary optimization: none, randomize **(reload the webpage to see different random**
Bitwidth: 8 mantissa + 0 fractional = 8 total
Integer constants: 49 188 248 251 245 37 129 206 207 83

```
./synth/acm1  -maxdepth 0 -expensive -aux  -b 8  '49' '188' '248' '251' '245' '3
Solution infeasible with MAX_DEPTH=0. Increasing MAX_DEPTH=2.
cat ./dags/dot1390691735 | ./dot.sh ./dags/dag1390691735
/usr/bin/convert -resize 425x425 ./dags/dag1390691735.large.png ./dags/dag139069
```

// Cost: 14 adds/subtracts 17 shifts 0 negations
// Depth: 2




```
./firgen/multBlockGen.pl  '49' '188' '248' '251' '245' '37' '129' '206' '207' '8
```

Download C output
Download Verilog output

More information

Return to the Multiplier Block Generator.

Contact: yvoronen at ece.cmu.edu

Figure 6.10: Realizing the filter $\{49, 188, 248, 251, 245, 37, 129, 206, 207, 83\}$ using Hcub method.

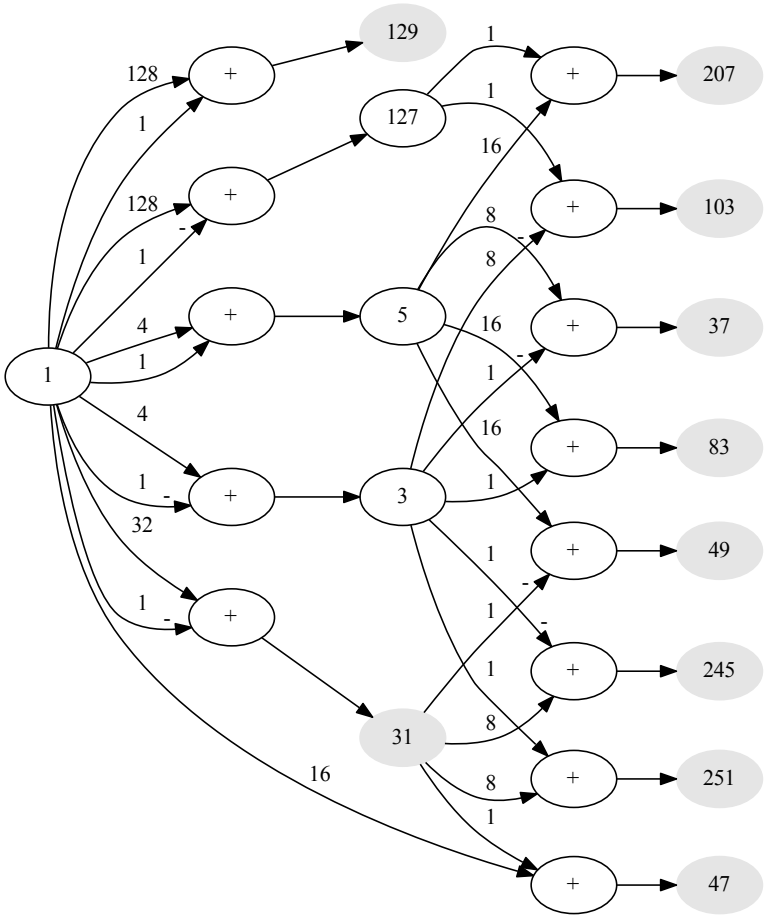


Figure 6.11: Realizing the filter $\{49, 188, 248, 251, 245, 37, 129, 206, 207, 83\}$ using STA.

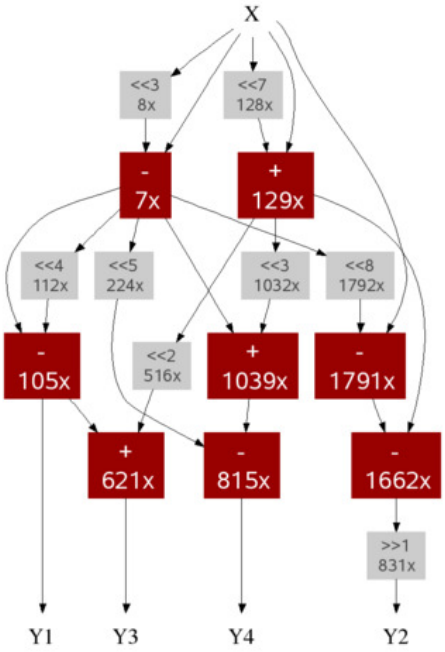
Return to the Multiplier Block Generator.

Output

Algorithm: Hcub
Auxiliary distance estimate: CSD-Cost(z)
Depth bound: 0
Secondary optimization: none, randomize **(reload the webpage to see different random graphs)**
Bitwidth: 10 mantissa + 0 fractional = 10 total
Integer constants: 105 831 621 815

```
./synth/acm1  -maxdepth 0 -expensive -aux  -b 10  '105' '831' '621' '815' -dotcode ./dags/dot1390691346 2>&1 > ./dags/mc
Solution infeasible with MAX_DEPTH=0. Increasing MAX_DEPTH=3.
cat ./dags/dot1390691346 | ./dot.sh ./dags/dag1390691346
/usr/bin/convert -resize 482x482 ./dags/dag1390691346.large.png ./dags/dag1390691346.png
```

// Cost: 8 adds/subtracts 8 shifts 0 negations
// Depth: 3




```
./firgen/multBlockGen.pl  '105' '831' '621' '815' -inData X -outData Y -fractionalBits 0 -acmOutput ./dags/mcm1390691346
```

Download C output
Download Verilog output

More information

Return to the Multiplier Block Generator.

Contact: yvoronen at ece.cmu.edu

Figure 6.12: Realizing the filter $\{4105, 831, 621, 815\}$ using Hcub method.

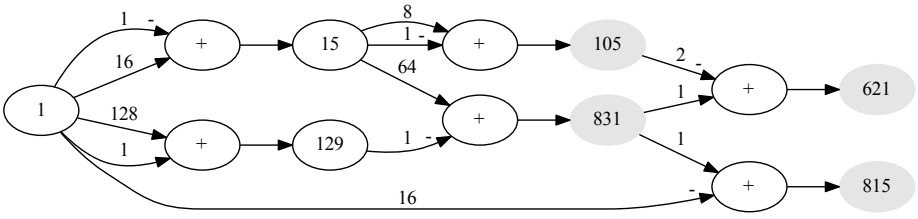


Figure 6.13: Realizing the filter $\{105, 831, 621, 815\}$ using STA.

11

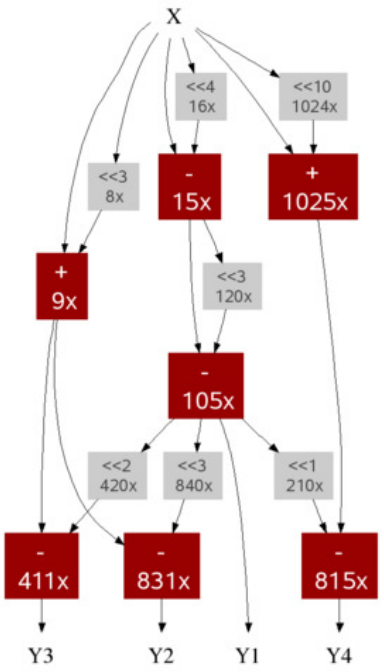
Return to the Multiplier Block Generator.

Output

Algorithm: Hcub
Auxiliary distance estimate: CSD-Cost(z)
Depth bound: 0
Secondary optimization: none, randomize **(reload the webpage to see different random graphs)**
Bitwidth: 10 mantissa + 0 fractional = 10 total
Integer constants: 105 831 411 815

```
./synth/acm1  -maxdepth 0 -expensive -aux  -b 10  '105' '831' '411' '815' -dotcode ./dags/dot1390661634 2>&1 > ./dags/mcm1390661634
Solution infeasible with MAX_DEPTH=0. Increasing MAX_DEPTH=3.
cat ./dags/dot1390661634 | ./dot.sh ./dags/dag1390661634
/usr/bin/convert -resize 495x495 ./dags/dag1390661634.large.png ./dags/dag1390661634.png
```

// Cost: 7 adds/subtracts 7 shifts 0 negations
// Depth: 3




```
./firgen/multBlockGen.pl  '105' '831' '411' '815' -inData X -outData Y -fractionalBits 0 -acmOutput ./dags/mcm1390661634 -outFile .
```

Download C output
Download Verilog output

More information

Return to the Multiplier Block Generator.

Contact: yvoronen at ece.cmu.edu

Figure 6.14: Realizing the filter $\{105, 831, 411, 815\}$ using Hcub method.

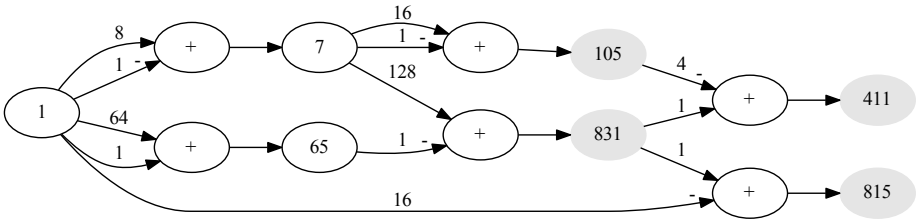


Figure 6.15: Realizing the filter $\{105, 831, 411, 815\}$ using STA.

6.5 CHAPTER SUMMARY

A common subexpression elimination algorithm is proposed to minimize the complexity of the MCM operation. The algorithm used BSD representation tree to find the possible decompositions. The results show that the possibility of finding MCM realizations with minimum logic depth is increased because of the subexpression space becomes larger. A comparison with other algorithms supports this argument where the algorithm has superior performance in realizing the coefficients with wordlengths up to 16 bits. This is useful in realizing short wordlength DSP systems such as in mobile communications in which power consumption is the main concern. However, the algorithm can be modified in the future to synthesize longer wordlength coefficients. A possible choice is to use number system other than BSD by letting the digit set include the synthesized fundamentals (non-constant digit set). In this case, the search space could be reduced to the minimum Hamming weight. Another possibility to reduce the search space is by using number systems with variable radix and/or digit set. However, these arguments require further investigation.

This work also introduced the concept of co-prime subexpressions as a generalization for the positive odd coefficients terminology used in the literature with respect to the BSD representations. This allows the work to be extended in the future to include other redundant number systems.

Chapter 7

COMBINATORIAL MODEL OF MULTIPLE CONSTANT MULTIPLICATION

The MCM problem has been an active research area for the last two decades. Many heuristic algorithms are proposed in the literature to solve this problem (as we have seen in the previous chapters). Finding a more efficient search algorithm than heuristics requires developing a combinatorics model for the MCM problem. Once this model is found, metaheuristics algorithms can be used instead of heuristics to search a very large space of candidate solutions. Examples of metaheuristics include evolutionary computation [De Jong, 2006] and ant colony optimization [Dorigo and Stützlea, 2004]. One definition for metaheuristics can be found in [Dorigo and Stützlea, 2004] which sates that "a metaheuristic can be seen as a general-purpose heuristic method designed to guide an underlying problem-specific heuristic toward promising regions of the search space containing high-quality solutions". According to our best of knowledge, there is no combinatoric model in the literature that describes the MCM problem. In this chapter we develop a model to the MCM problem using the subexpression space concept proposed in Chapter 6. We combine the individual subexpression spaces into one large space resulting in an acyclic directed graph. In this way, subexpressions are shared which makes the problem of minimizing the MCM a graph traversal. Traversing an arc on the graph costs a number of adders determined by the complexity of the subexpression attached to this arc. The traversed arc becomes a deadheading which means there is no cost when visiting this arc in the next time [Evans and Minieka, 1992]. The tours on the demand graph are analogized to the real life problem of dynamic winter gritting. The Ant Colony Optimization (ACO) technique is proposed as an efficient metaheuristic method that can search a graph in parallel.

This chapter is structured as follows: Some basic arc routing problems are discussed in Section 7.1. In Section 7.2, we propose the concept of the decomposition, demand, and augmented demand graphs. We propose in Section 7.3 the winter gritting problem as an analogy for the MCM problem. The ACO metaheuristic is explained in Section 7.4.

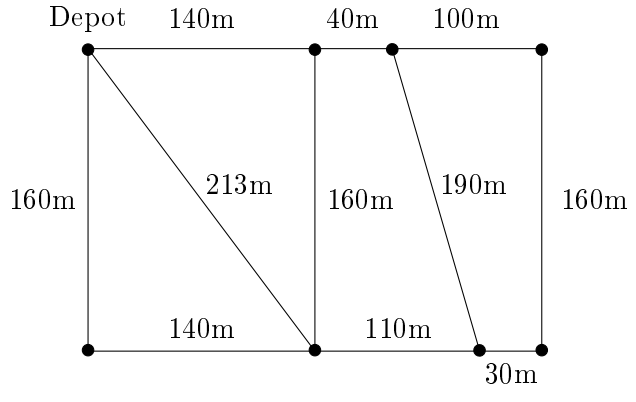
7.1 ARC ROUTING PROBLEMS

Graph routing problems are classified into two main categories which are vertex (node) routing and arc (edge) routing problems [Evans and Minieka, 1992]. Vertex routing includes problems where service (demand) occurs at the vertices of the graph. Examples

include traveling salesman, traveling tourist, etc. Arc routing includes problems where the service occurs on the graph arcs. Examples include street maintenance, garbage collection, milk or mail delivery, school bus tour, electric meter reading, and winter gritting. The MCM belongs to the arc routing category (this will be shown in Section 7.3). A literature review is given in the following subsections that considers some of the arc routing problems.

7.1.1 The Chinese Postman Problem

The Chinese Postman Problem (CPP) is a graph arc route inspection problem proposed by Meigu Guan, a mathematician at the Shaugtun Normal College who spent sometime as a post office worker during the Chinese Culture Revolution [Eiselt et al., 1995a]. The problem is stated as follows: Suppose the mailman needs to deliver mail to a suburb as shown in Figure 7.1. His route starts and ends at the Depot. The mailman needs to go through every street at least once. However, he wants to find the shortest route through the streets. If the graph has an Eulerian Circuit (Section A.4), this circuit is the ideal solution. Only the directed or undirected (Section A.1) CPP can be solved in polynomial time. The mixed problem is raised when there are directed (one way) and undirected (two way) arcs which is shown to be NP-hard [Eiselt et al., 1995a].



;

Figure 7.1: Streets that must be traversed by a mailman.

7.1.1.1 Problem Formulation

Let $G(V, A)$ be a connected graph (Section A.3), $V = \{v_1, v_2, \cdots, v_n\}$ is the vertex set, and $A = \{(v_i, v_j) : v_i, v_j \in V \text{ and } i \neq j\}$ is the arc set such that there is a distance (cost) c_{ij} associated with arc (v_i, v_j). The graph should be unicursal (Section A.4) so the postman is able to make a complete tour (circuit). This requires augmenting the graph with extra arcs if it contains vertices with odd in-out-degree (Section A.1). Consider the graph G in Figure 7.2 (a). Since the in-degree is not equal to the out-degree for the vertices 17, 31, and 33, no Eulerian tour can be found in the graph. Applying an augmentation procedure by adding one arc from vertex 33 to vertex 17 and two arcs from vertex 31 to vertex 33, results in the symmetric graph $\hat{G}$ shown in Figure 7.2 (b). An Eulerian tour can be found in $\hat{G}$ which corresponds to an optimal postman tour. For example, the tour

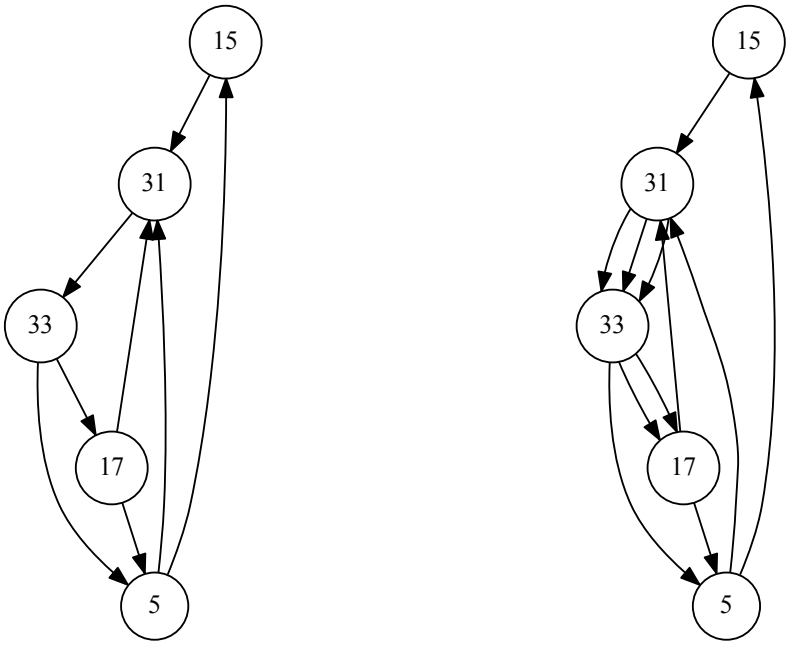


Figure 7.2: An example of making a non-unicursal graph G to unicursal one $\hat{G}$. (a) A graph G with the in-degree not equal the out-degree for the vertices $17, 31$, and 33. (b) A graph $\hat{G}$ with in-degree equals out-degree for all the vertices.

$((15,31),(31,33),(33,17),(17,31),(31,33),(33,17),(17,5),(5,31),(31,33),(33,5),(5,15))$ is Eulerian.

Generally, let f_{ij} be the number of extra arcs (v_i, v_j) required to augment G, $\delta(i)$ the set of arcs incident to v_i, and let $U \subseteq V$ be the set of vertices with the in-degree not equal to the out-degree, then the CPP is

$$\text{Minimize} \sum_{(v_i,v_j)\in V} c_{ij} f_{ij}, \tag{7.1}$$

subject to:

$$\sum_{(v_i,v_j)\in\delta(i)} f_{ij} = \begin{cases} 1 & \text{if } v_i \in U \\ 0 & \text{if } v_i \in V \setminus U, \end{cases} \tag{7.2}$$

where $v_i \in V \setminus U \equiv v_i \in V$ and $v_i \notin U$. For the example in Figure 7.2, $v_i = 33$, U $=\{17,31,33\}$, $f_{(33)(17)} = -1$, $f_{(31)(33)} = 2$. The negative sign is for the arcs departing a vertex, while the positive sign is for the arcs arriving at the vertex.

7.1.2 The Capacitated Chinese Postman Problem

The Capacitated Chinese Postman Problem (CCPP) arises when one vehicle or more (called a fleet of vehicles) is/are required to service a set of customers or network of roads with constraints on vehicle capacity, fuel amount, and time. Examples include postmen

deriving mail by fleet of vehicles with the amount of fuel, or limited capacity vehicle as constraints on each vehicle. Another example of limited capacity vehicles is when a fleet of vehicles is required to service a network of roads. Each vehicle is assigned a feasible route of roads to service them such that each road (arc) has a positive demand. Thus, the CCPP determines a set of routes from the depot (place of vehicle departure) that services all the roads with minimum cost without violating vehicle capacity constraints. The CCPP is shown to be NP-hard [Li and Eglese, 1996].

7.1.3 The Capacitated Arc Routing Problem

The Capacitated Arc Routing Problem (CARP) arises when some of the arcs have zero demand. In this case, only a subset $R \subseteq A$ of arcs require servicing. Other arcs that do not require servicing are used to deadhead between arcs that require service. In this case, deadheading corresponds to traveling on roads that do not require service. Usually we wish to minimize deadheading. The CARP is NP-hard [Evans and Minieka, 1992]. An example of the CARP is the rural postman problem (RPP) [Eiselt et al., 1995b]. In the rural postman problem not all street segments require service. Here, the postman traverses deadheading streets to arrive at the street segments to be serviced. The undirected and the directed rural postman problems are NP-hard so heuristics are required to solve the problem. Another example of the CARP is the winter gritting or salting problem described in the next section.

7.1.4 Winter Gritting (Salting) Problem

When roads become hazardous due to ice or snow, a de-icing agent (usually salt) is spread on them for safety reasons. The problem of winter gritting is how to design routes for a fleet of vehicles departing from a depot to minimize the costs. A depot is where the vehicles are loaded with salt. This problem belongs to the CARP, therefore it is an NP-hard problem [Li and Eglese, 1996; Eiselt et al., 1995a,b]. Many constraints may be imposed on the driver such as time constraints, different priorities of roads, and limited vehicle capacity.

7.1.5 Dynamic Winter Gritting (Salting) Problem

In snowy weather, the most significant problem is in the priority of which roads to grit [Handa et al., 2005]. A wrong decision of spreading salt on roads that do not require service is a financial loss. On the other hand, untreated roads are a major hazard. Therefore the decision should be taken on dynamic basis. This means that the driver may re-plan the route according to newly received weather forecast information. The route in this case become dynamic and using static heuristics may be insufficient to find a solution.

7.2 DECOMPOSITION, DEMAND, AND AUGMENTED DEMAND GRAPHS

Each MCM may have many individual realizations. Combining all of these realizations in one graph establishes the sharing among them. For example, consider the realizations shown in Figure 7.3 for the coefficients $\{5, 37, 47\}$. Combining these realizations results in the decomposition graph shown in Figure 7.4 (a). In this figure, the arcs with similar color represent the input of the $\mathcal{A}-$operation defined in Equation 6.8.

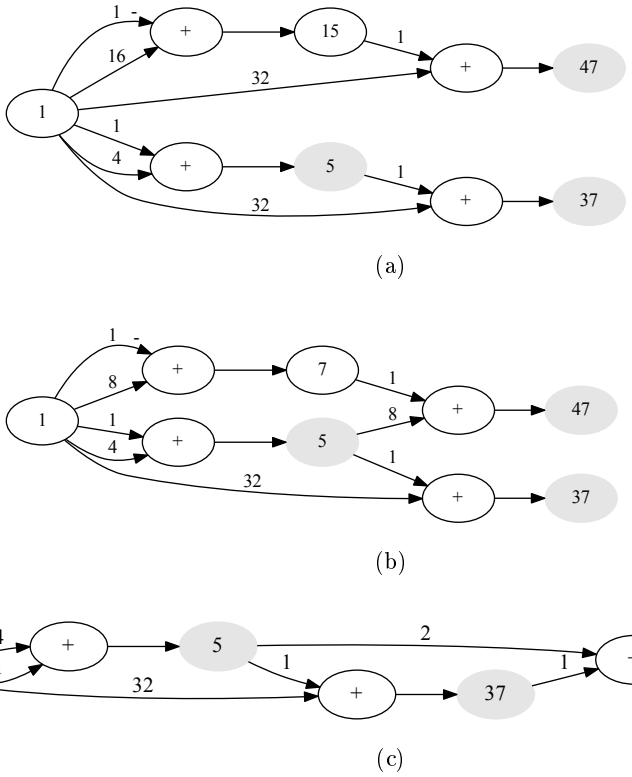


Figure 7.3: Three possible realizations for the coefficients $\{5, 37, 47\}$.

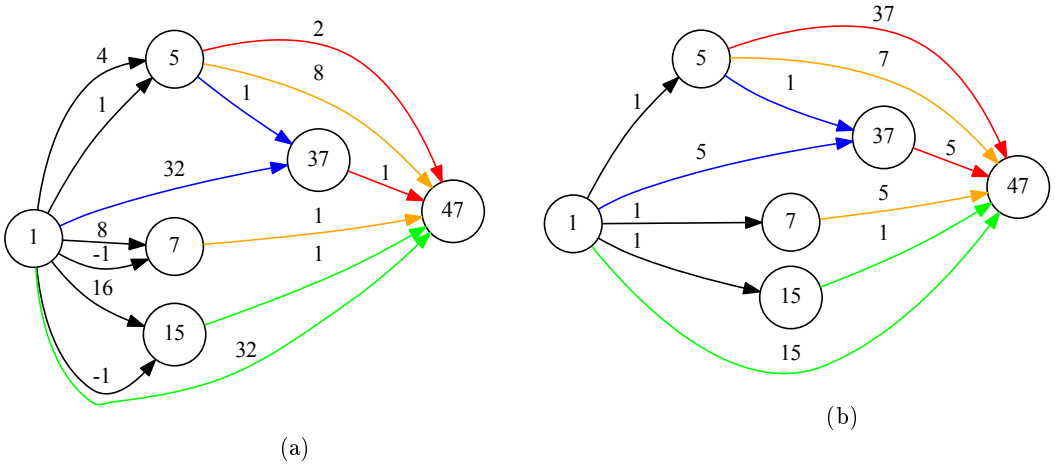


Figure 7.4: (a) Combining the realizations in Figure 7.3 results in the decomposition graph. (b) Transforming the decomposition graph results in the demand graph.

To make the graph unicursal (routable), a transformation should be applied to the $\mathcal{A}-$operation. The transformation steps are shown in Figure 7.5. The graph representation for the $\mathcal{A}-$operation is shown in Figure 7.5 (a). In this case, the arcs only carry the shift information. Figure 7.5 (b) combines the addition vertex with vertex w (the output of the $\mathcal{A}-$operation). The last step that shown in Figure 7.5 (c) illustrates replacing the shift information with subexpression information. The subexpression information is considered as a demand on a traversed arc. Thus, traversing the arc from vertex s_1 to w requires synthesizing (servicing) vertex s_2. Similarly, traversing the arc from vertex s_2 to w requires servicing vertex s_1. Applying this transformation to the decomposition graph, shown in Figure 7.4 (a), results in the demand graph shown in Figure 7.4 (b).

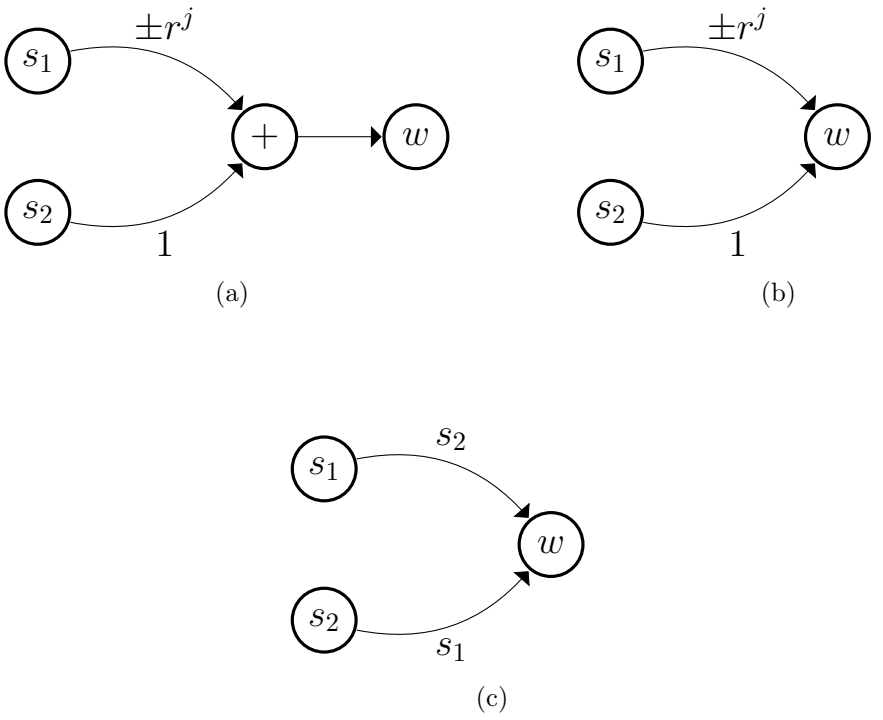


Figure 7.5: Transforming the shift information to a subexpression information. (a) Representing the $\mathcal{A}-$operation where the arcs carry shift information. (b) Combining the addition vertex with the output vertex. (c) A new representation for the two part decomposition.

Augmenting the demand graph in Figure 7.4 (b) with deadheading arcs make it unicursal as shown in Figure 7.6. The deadheading arcs are of zero cost because there is no demand to do any services when traversing these arcs. The realizations shown in Figure 7.3 are just a subset of the realization set. This is because we considered only a subset of the subexpression space of the coefficients $\{5, 37, 47\}$. Considering all the subexpression space results in the demand graph shown in Figure 7.7. The augmented demand graph is shown in Figure 7.8. There is an enormous number of possible routes on the graph in Figure 7.8. Finding the optimum route(s) can be compared with other real life arc routing problems as described in Section 7.3.

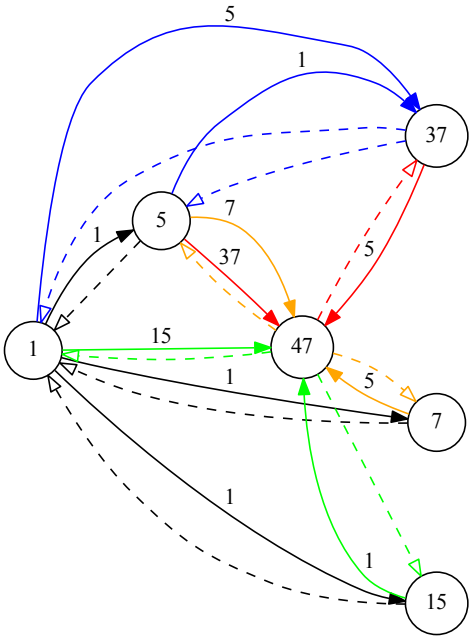


Figure 7.6: Adding deadheading arcs to the demand graph shown in Figure 7.4 (a).

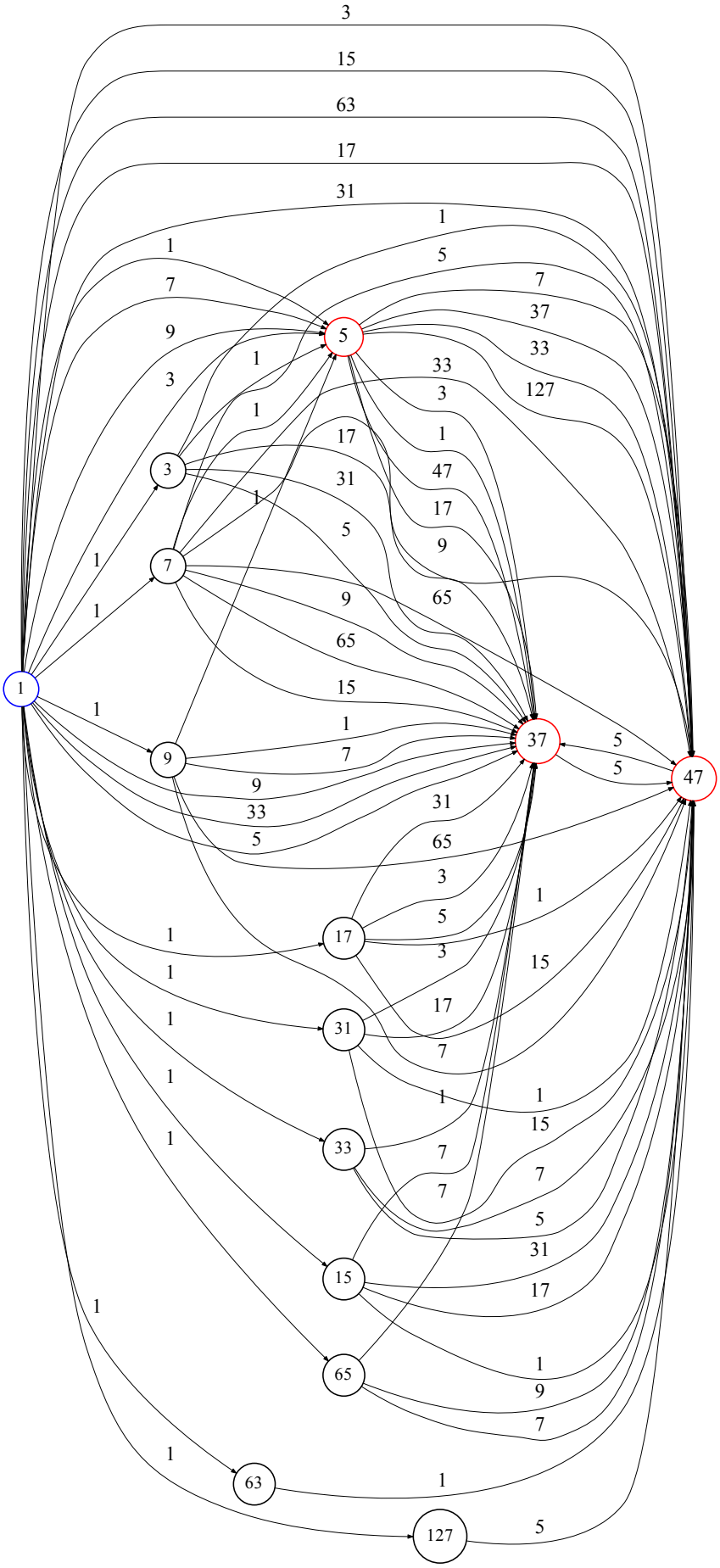


Figure 7.7: The demand graph of the coefficients $\{5, 37, 47\}$.

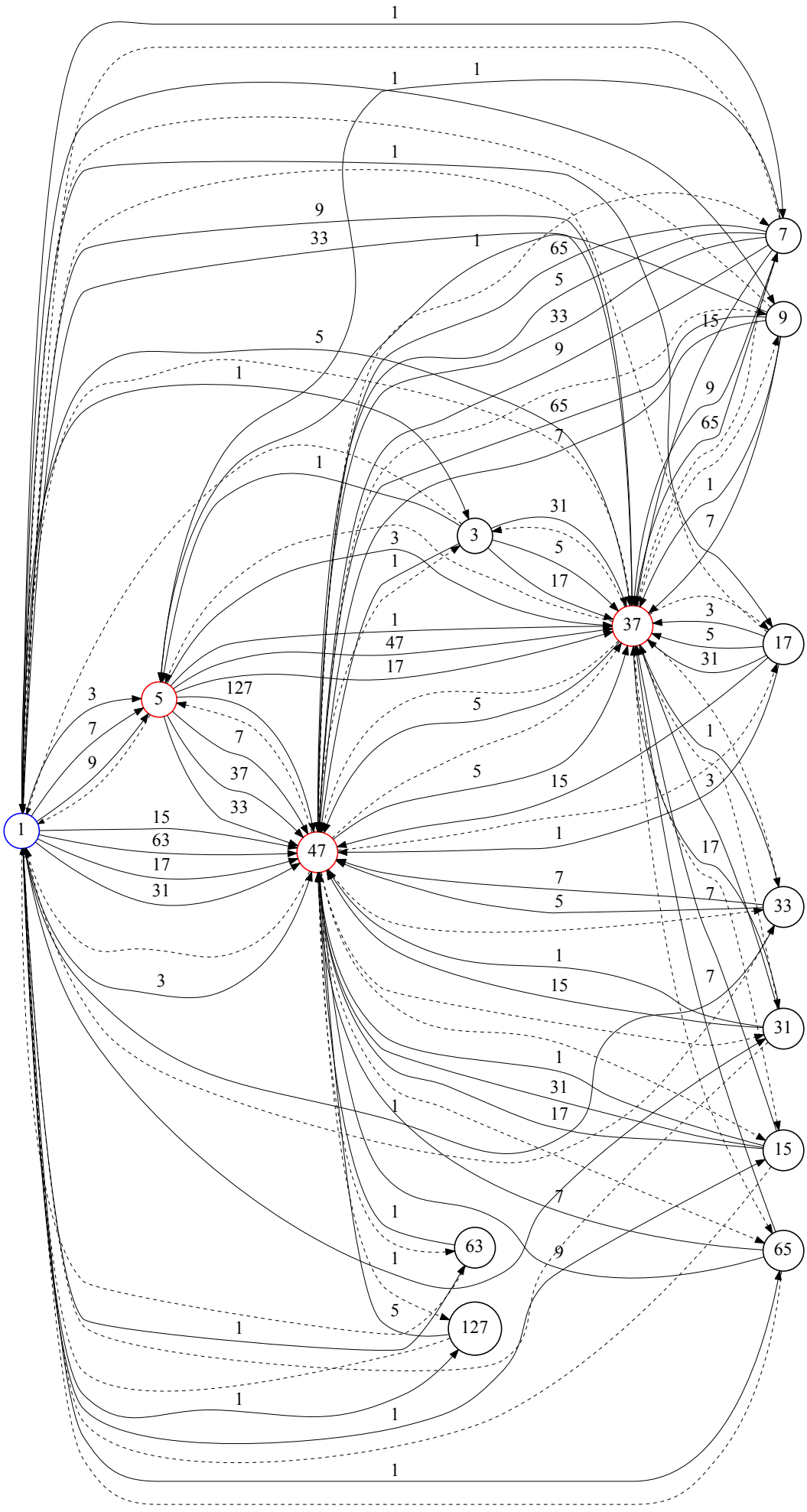


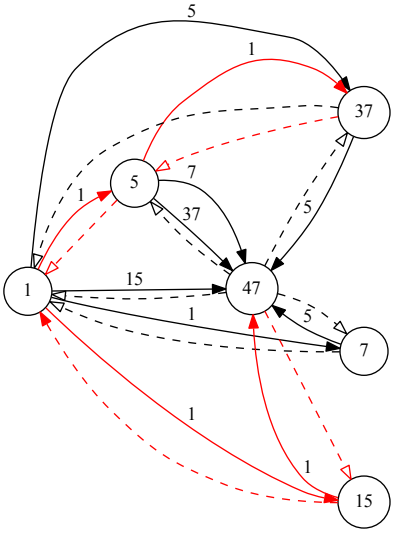
Figure 7.8: Adding dead heading arcs to the demand graph of the coefficients $\{5, 37, 47\}$.

7.3 DYNAMIC WINTER GRITTING ANALOGY

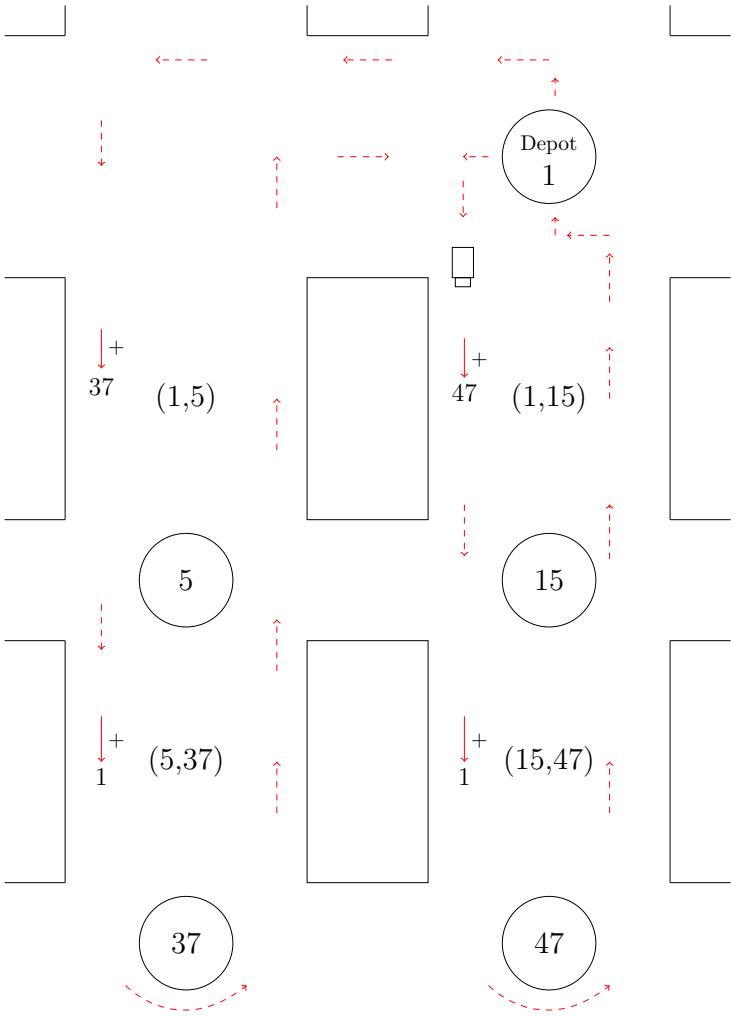
The existence of demands on the arcs of the graph in Figure 7.6 makes it a dynamic routing problem. It is similar to other dynamic arc routing problems. One particular arc routing is the dynamic winter gritting problem discussed in Subsection 7.1.5. Consider one possible tour on the graph shown in Figure 7.9(a). This tour results in the realization shown in Figure 7.3(a) with logic operator LO = 4 adder and logic depth constraint of LD = 2 adder. The analogy of the tour in Figure 7.9(a) is the winter gritting route shown in Figure 7.9(b). In this case, a vehicle departs from the depot to service the roads $5, 37$, and 47. The blocks are just for illustration and not necessary represent city blocks because such service usually occurs on high way road networks. To make things simple, the capacity of the vehicle is assumed to be 20 tons and each road consumes 10 tons of salt. Assuming that the cost correspondence between the salt weight and adder cost is that each 10 tons $\equiv$ 1 adder. The correspondence between the adder depth constraint and vehicle capacity is that violating the constraint LD = 2 is equivalent to violating the vehicle capacity which is 20 tons. The depot in Figure 7.9(b) corresponds to the signal vertex in Figure 7.9(a), with label 1. Assume that due to a change in the weather forecast, road 15 becomes more hazardous than other roads and requires urgent servicing. Therefore, the station would signal to the driver to service road 15 first. The driver in this case should re-plan his/her route and go to service road 15. Again for simplicity and to illustrate the concept, road 15 is draw to be next to the depot and in the reality the deriver could drive a long distance from the depot when he/she received the redirection. The corresponding movement on the graph in Figure 7.9(a) is to go from vertex 1 to synthesize coefficient 15. The red arrow in Figure 7.9(b) illustrates this service which is labeled by $(1, 15)$. The corresponding movement in Figure 7.9(a) is shown by the red arc $(1, 15)$. The cost of servicing road 15 equals 10 tons which is indicated by the plus sign on the direction (arrow) of the vehicle. The plus sign here is introduced to keep the corresponding with the augmented demand graph. After servicing road 15, the driver returns to the original plan and finds that road 47 is the shortest distance to his/her position as shown in Figure 7.9(b). The service of road 47 from the end of 15 is labeled by $(15, 47)$ which consumes another 10 tons. The corresponding movement on Figure 7.9(a) is to synthesize 47 from 15 resulting LD = 2 adders. At this point the vehicle became empty and the deriver cannot continue to serve other roads. Therefore, the driver decides to return to the depot through the deadheading roads 47 and 15 indicated by the red dashed lines shown in Figure 7.9(b). These roads are considered deadheading because they have been serviced. Similarly for the demand graph, synthesizing other coefficients from 47 (for example 5 or 37) will violate the logic depth constraint. The corresponding movement of the demand graph is traversing the dashed red arcs $(47, 15)$ and $(15, 1)$ respectively as shown in Figure 7.9(a). These arcs are also deadheading because their attached subexpressions were synthesized. The route is now (depot $\rightarrow 15 \rightarrow 47 \rightarrow 15 \rightarrow$ depot) which is corresponding to the tour $(1, 15), (15, 47), (47, 15), (15, 1)$ on the demand graph in Figure 7.9(a). Then the driver services road 5 with 10 tons of salt and this service is denoted by $(1, 5)$. The last road is 37, which is serviced after road 5 and require 10 tons of salt. This is correspond to

synthesize 5 from 1 in Figure 7.9(a), then synthesize 37 from 5. The resulting logic depth after synthesizing 37 equals 2 adders which will not violate the logic depth constraint. On the other hand, the winter gritting route ends after finishing road 37 and the driver returns to the depot via the deadheading arcs. The vehicle route is now (depot $\rightarrow$ 15 $\rightarrow$ 47 $\rightarrow$ 15 $\rightarrow$ depot $\rightarrow$ 5 $\rightarrow$ 37 $\rightarrow$ 5 $\rightarrow$ depot). The corresponding tour on the graph in Figure 7.9(a) is $((1,15),(15,47),(47,15),(15,1),(1,5),(5,37),(37,5),(5,1))$. This is a feasible tour on the demand graph because it will not violate the adder depth constraint. The resulting LO of this tour equals 4 adders. The LO would be compared with that results from other tours to find the one(s) with minimum cost and not violating logic depth constraint. Another possible tour on the demand graph is shown in Figure 7.10(a) and its corresponding winter gritting route is shown in Figure 7.10(b). This tour results in the realization shown in Figure 7.3(b) with LO = 4 and LD = 2.

Obtaining realizations with smaller LO is possible if the LD constraint is relaxed (increased). This can be illustrated by using the same example of the coefficients $5, 37$, and 47 when LD is allowed to be of maximum value of 3 adders. In the corresponding winter gritting route, the vehicle is assumed to be upgraded to have a capacity of 30 tons. In this case, the driver can serve roads $5, 37$, and 47 in succession without violating vehicle capacity as shown in Figure 7.11(b). The corresponding tour on the demand graph is illustrated in Figure 7.11(a) which results in the realization shown in Figure 7.3(c) with LO = 3 and LD = 3. The same realization would be obtained using the graph dependent method.

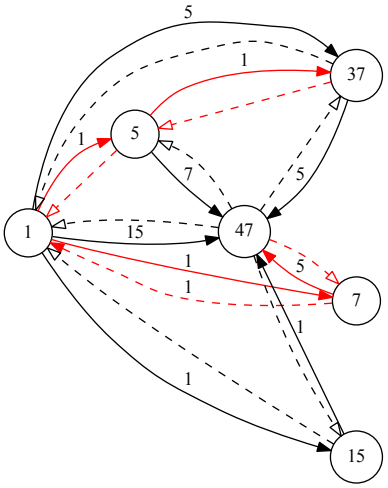


(a)

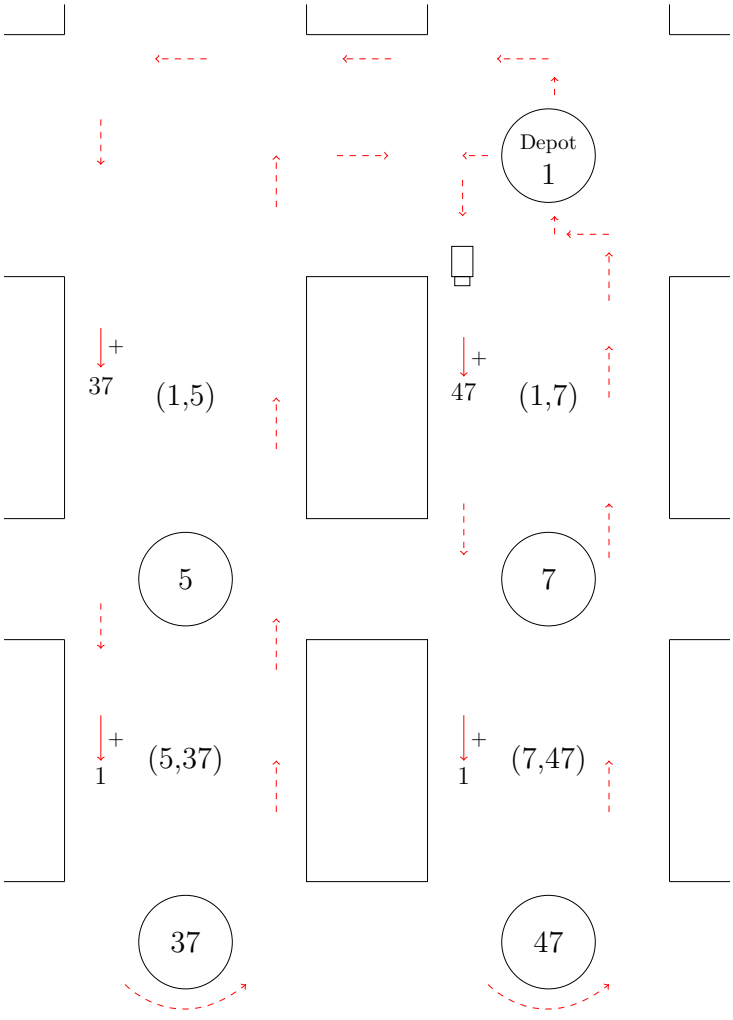


(b)

Figure 7.9: (a) A possible tour on the augmented demand graph results in the realization shown in Figure 7.3(a). (b) The corresponding winter gritting route. A solid arrow with plus sign indicates the salting action and the label preceding each arrow is the next destination after this road. A dashed arrow means traversing a deadheading road. The roundabouts are labeled by the road number and the road is labeled by the path between two roundabouts.

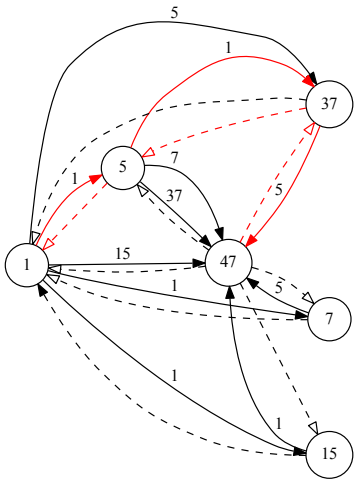


(a)

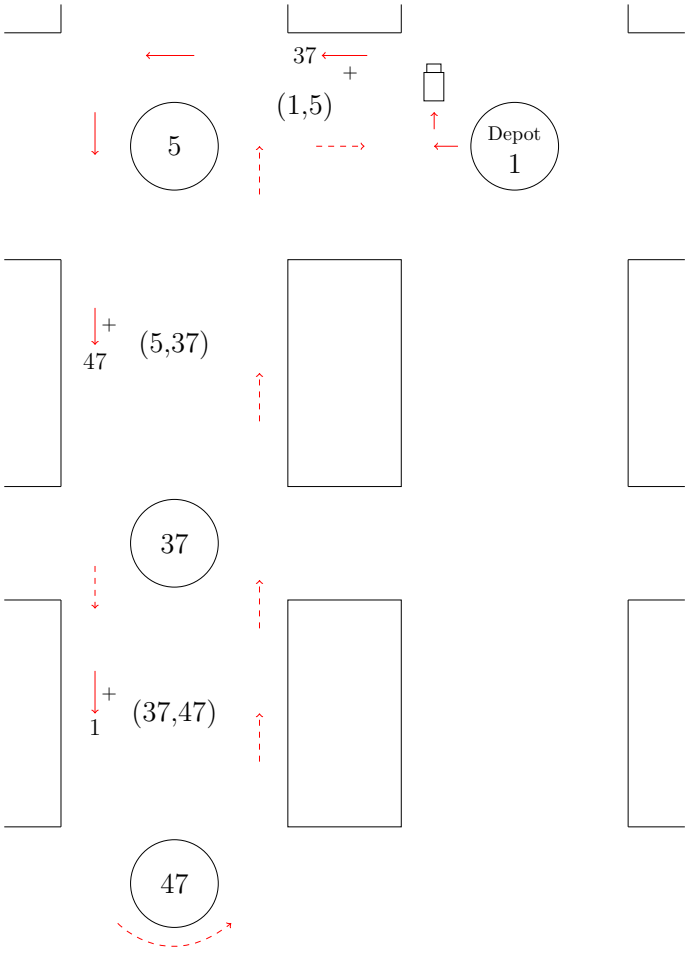


(b)

Figure 7.10: (a) A possible tour on the augmented demand graph results in the realization shown in Figure 7.3(b). (b) The corresponding winter gritting route.



(a)



(b)

Figure 7.11: (a) A possible tour on the augmented demand graph results in the realization shown in Figure 7.3(c). (b) The corresponding winter gritting route.

Another example of the touring on the demand graph is shown in Figure 7.12 for the coefficients $\{45, 195\}$. A possible tour on this graph under the constraint LD = 2 is shown in blue color. The tour starts from the depot to the subexpression 65 with no demand, then from 65 to 195 with demand 65 which is already synthesized. Going beyond 195 violates the adder depth constraint and therefore it is required to return to the depot through the blue dashed path. The tour is continued from the depot to the subexpression 3 with no demand, then from 3 to 45 with no demand too because the subexpression 3 is already synthesized. The tour ends with a realization of LO = 4 and LD = 2 adders. The LO can be reduced if the logic depth constrained is relaxed to LD = 3. This is shown by the tour shown in green. This tour starts from the depot to the subexpression 15 with no demand. Traversing the arc from 15 to 45 is now with zero cost because the demand 15 is already synthesized. Crossing the arc from 45 to 195 is for free too for the same reason. The tour finished with a realization of LO = 3 and LD = 3 adders. Again, this realization could be obtained from using the graph dependent method. In conclusion, the routing on the demand graph gives solutions that can be obtained from using the CSE and GD methods. This is a big contribution in this branch of science that different heuristics used to solve the MCM are unified when traversing the demand graph. Therefore developing the demand graph in this work is considered a significant step towards finding optimum solutions for any MCM.

Consider the tour that starts from the depot to synthesize 17 then moves to 45 through the demand 31. At this point, the coefficient 45 cannot be synthesized because the demand 31 is unsynthesized. The tour is redirected by searching for synthesized ancestors of 31 which is the depot in this case. Therefore, instead of going from 45 to 195, the tour is redirected to the depot which is considered a dynamic behavior. The tour continues from the depot to the subexpression 31 with a demand is 1. The subexpression is synthesized and the tour goes from 31 to 45 through the arc with demand 17. This crossing is free because 17 has been already synthesized. Therefore, the coefficient 45 is completely synthesized because its decomposition parts of 17 and 31 are both synthesized. Going from 45 to 195 requires synthesizing the demand 15. The tour is redirected to go from 195 to the depot then to 15 which is now fully synthesized. The tour can cross from 15 to 195 with zero cost because the arc is of demand 45 which has already been synthesized. Therefore, the coefficient 195 is completely synthesized. This tour is isolated from the demand graph as shown in Figure 7.13(a). In this case, Figure 7.13(a) shows the realization of the coefficients by considering solid arrows and neglecting the dashed arrows. Each vertex includes addition operation except the depot. The dynamic behavior in this tour is obvious because the demand on the arcs forces the router to search for their synthesized ancestor instead of going to the next vertex. The corresponding winter gritting route is shown in Figure 7.13(b). Letter signs are added here to help the reader track the route easily.

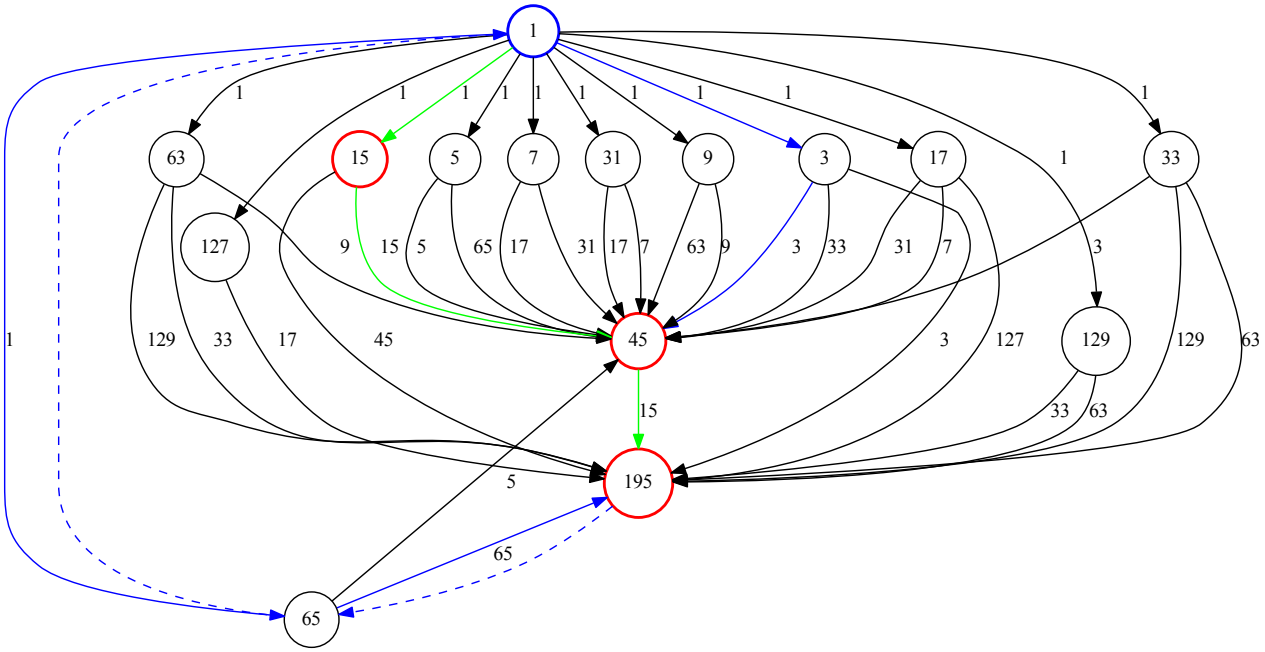


Figure 7.12: Two possible tours on the demand graph for the coefficients $\{45, 195\}$.

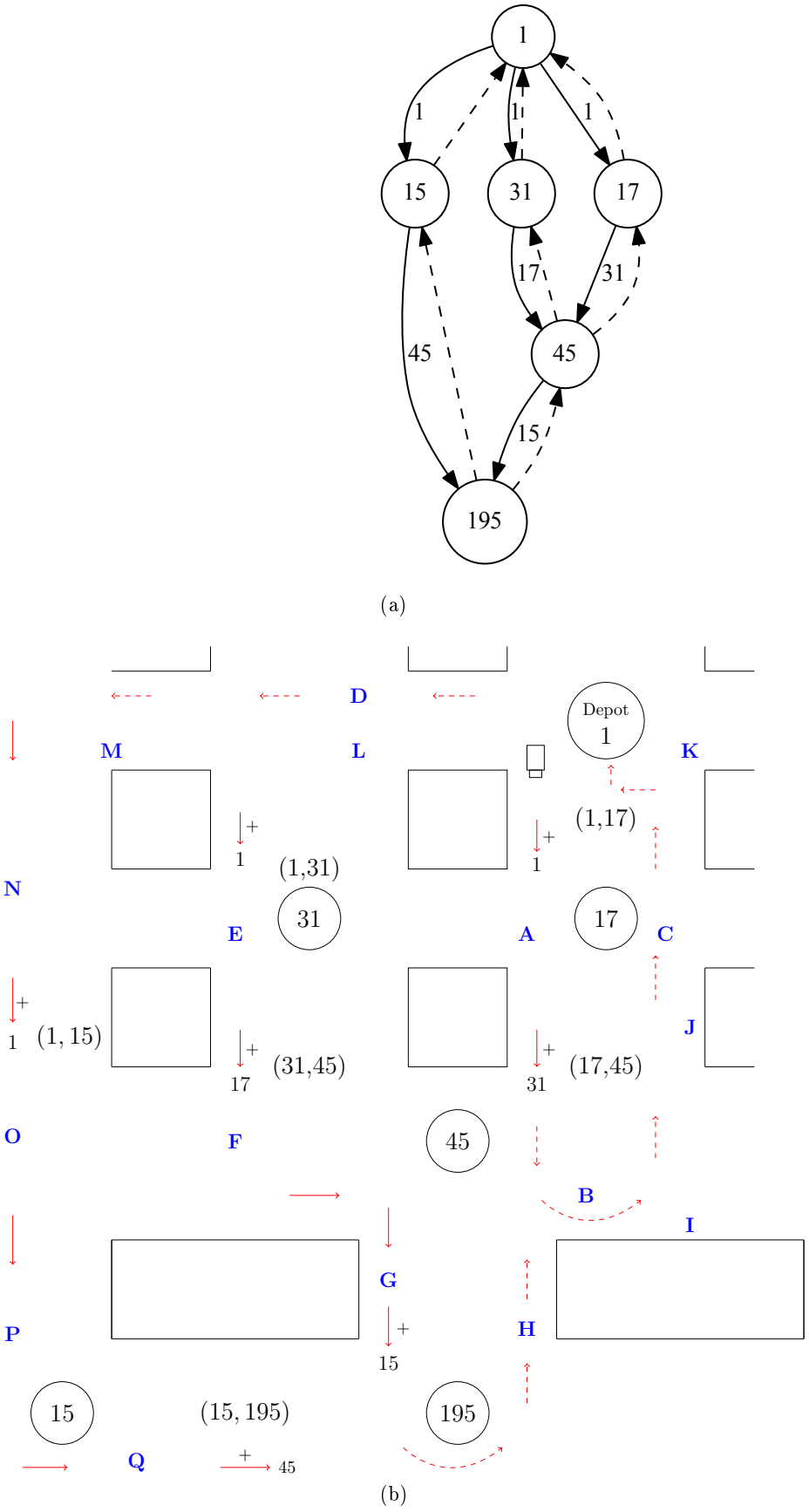


Figure 7.13: (a) The tour (depot $\rightarrow$ 17 $\rightarrow$ 45 $\rightarrow$ 17 $\rightarrow$ depot $\rightarrow$ 31 $\rightarrow$ 45 $\rightarrow$ 195 $\rightarrow$ depot $\rightarrow$ 15 $\rightarrow$ 195 $\rightarrow$ depot) on the demand graph for the coefficients $\{45, 195\}$. (b) The corresponding winter gritting route.

7.4 ANT COLONY OPTIMIZATION

We have shown in Section 7.3 that the routing on the augmented demand graph is analogous to the dynamic winter gritting. Therefore, the MCM problem is a dynamic capacitated arc routing problem. The ant colony optimization algorithm (ACO) has been used successfully to tackle a large variety of the dynamic vehicle routing problems including the dynamic capacitated arc routing problems [Silvia and Irene, 2004; Yu et al., 2008a; Montemanni et al., 2002; Ding et al., 2012]. The ACO algorithm is a metaheuristics proposed by Colorni et al. [1991] to solve combinatorial optimization problems. The algorithm is inspired by the swarm intelligence of the real life ant colony when searching for food. Each ant excretes a chemical factor called pheromone and puts it on its trail. The quantity of the pheromone on the shortest paths to the food is reinforced by the successive passing of other ants. On the other hand, the pheromone on the longer paths evaporates during that time. Since ants choose the path with high pheromone concentrations, most of the other ants select that path too. While a minority selects alternative paths. Rizzoli et al. [2004] showed that the behavior of this minority is important because these ants continue searching for a better solution.

The ACO algorithms use artificial ant agents that search a graph concurrently by traversing the arcs. Each artificial ant builds its solution by constructing a tour on the graph. The tour starts from the source vertex and ends at the destination vertex. The ants use the stored information at arcs to decide the next destination. Each arc on the graph stores information about the pheromone quantity and the heuristic value associated with this arc. The heuristic values could be the length of the arc as in the CCPP problems. Before the ants start their first tour, graph arcs are initialized with an equal pheromone level. Arcs with low heuristic values and stronger pheromone quantities are more attractive to the ants. The ants start constructing their solutions by routing on the graph. After each ant finishes its tour, the pheromone level should be reduced on all the graph arcs. An evaporation formula is used to determine the evapuration rate. Then each ant deposits a pheromone on the arcs of its constructed tour. Pheromone evaporation and depositing continue until no more improvement on the solutions quality can be obtained from making more iterations under the given setting of parameters. If the lower bound of the MCM is known, then the solutions optimality can be examined. Since the artificial ants construct their solutions in parallel, ant colony optimization algorithms can easily amend themselves to parallel computing [Pedemonte et al., 2011; Yu et al., 2011]. There are four basic different ACO algorithms considered in this work. They are the Ant System (AS), Elitist Ant System (EAS), Rank based Ant System (AS_{rank}), and MAX-MIN Ant System (MMAS) [Dorigo and Stützlea, 2004] [Doerner et al., 2004]. Other algorithms are derived from these basic algorithms.

7.4.1 Ant System

In AS, m artificial ants are distributed randomly on chosen graph vertices. Before the ants start their first tour, the pheromone's level should be initialized. The initial value of the pheromone, τ_0, should not be too low, otherwise the search is quickly biased

by the first tours generated by the ants [Dorigo and Stützlea, 2004]. On the other hand, choosing high τ_0 may slow the convergence of the algorithm. After depositing τ_0 on all the graph arcs, ants start their movement on the graph. The ants construct their solution such that each one determines its next move according to a probabilistic formula. The probability with which ant k, currently at vertex v_i chooses to cross to vertex v_j [Dorigo and Stützlea, 2004] is

$$p^k_{(v_i,v_j)} = \frac{[\tau_{(v_i,v_j)}]^\alpha [\eta_{(v_i,v_j)}]^\beta}{\sum_{l \in V^k_{v_i}} [\tau_{(v_i,v_l)}]^\alpha [\eta_{(v_i,v_l)}]^\beta}, \quad \text{if} \quad v_j \in V^k_{v_i}, \tag{7.3}$$

where:

- The terms in the form $[x]^y$ means the value x is raised to the power y, while x^y means y is a superscript.
- $\tau_{(v_i,v_j)}$ is the strength of the pheromone on the arc (v_i, v_j).
- $\eta_{(v_i,v_j)}$ is a heuristic related to the cost of traversing arc (v_i, v_j). For example, $\eta_{(v_i,v_j)}$ may represent the distance between two cities in the case of salesman problem. In MCM, the value of $\eta_{(v_i,v_j)}$ represents the complexity of the subexpression (demand) attached to the arc (v_i, v_j). Synthesized demand is of zero complexity (most favored). More details about arch complexity is given in the next chapter.
- α and β are two parameters which determine the relative influence of the pheromone trail and the heuristic information.
- $V^k_{v_i}$ is the set of vertices that ant k has not visited yet from vertex v_i.

Equation 7.3 is interpreted as follows. The probability of the ant k to cross from vertex v_i to vertex v_j increases with the value of pheromone trail $\tau_{(v_i,v_j)}$ on this arc and with the heuristic value $\eta_{(v_i,v_j)}$ associated with this arc [Dorigo and Stützlea, 2004].

After each ant has constructed its tour, a pheromone update cycle starts with the pheromone evaporation

$$\hat{\tau}^n_{(v_i,v_j)} = (1-\rho)\tau^{n-1}_{(v_i,v_j)}, \ \forall (v_i, v_j) \in A, \tag{7.4}$$

where $\hat{\tau}^n_{(v_i,v_j)}$ is the pheromone strength after evaporation, $\tau^{n-1}_{(v_i,v_j)}$ is the pheromone strength of the last iteration, $0 < \rho \leq 1$ is the pheromone evaporation rate and A is the arc set. After evaporation, all the ants deposit pheromone on the arcs they have crossed in their tour according to,

$$\tau^n_{(v_i,v_j)} = \hat{\tau}^n_{(v_i,v_j)} + \sum_{k=1}^{m} \Delta\tau^k_{(v_i,v_j)}, \ \forall (v_i, v_j) \in A, \tag{7.5}$$

where $\tau^n_{(v_i,v_j)}$ is the new pheromone strength, and $\Delta\tau^k_{(v_i,v_j)}$ is the pheromone quantity deposited by ant k on the arcs visited by this ant. This value is chosen to reflect the solution quality of the ant k. For example, if ant k found a shortest path solution with

length R_k, then a possible selection of $\Delta\tau^k_{(v_i,v_j)}$ is

$$\Delta\tau^k_{(v_i,v_j)} = \begin{cases} \frac{1}{R_k}, & \text{if arc } (v_i, v_j) \text{ belongs to the tour of ant } k; \\ 0, & \text{otherwise.} \end{cases} \tag{7.6}$$

7.4.2 Elitist Ant System

The EAS is an improvement on the AS proposed by Colorni et al. [1991]. In this method, additional pheromone level is added to the best-so-far tour, R_{bs}. This tour is the best one since the start of the algorithm. Pheromone evaporation is defined in Equation 7.4, while pheromone updating is now

$$\tau^n_{(v_i,v_j)} = \hat{\tau}^n_{(v_i,v_j)} + \sum_{k=1}^{m} \Delta\tau^k_{(v_i,v_j)} + \Delta\tau^{\text{bs}}_{(v_i,v_j)}, \tag{7.7}$$

where $\Delta\tau^k_{(v_i,v_j)}$ is defined in Equation 7.6 and $\Delta\tau^{\text{bs}}_{(v_i,v_j)}$ is defined as follows:

$$\Delta\tau^k_{(v_i,v_j)} = \begin{cases} \frac{1}{R_{\text{bs}}}, & \text{if arc } (v_i, v_j) \text{ belongs to the best-so-far tour;} \\ 0, & \text{otherwise.} \end{cases} \tag{7.8}$$

7.4.3 Rank-Based Ant System

The AS_{rank} is another improvement over AS proposed by Bullnheimer et al. [1999]. The modification here includes ranking the ants according to their solution quality. The ant with the lowest solution quality is considered of the highest rank and deposits the lowest pheromone level and vice versa. Not all the ants are selected to deposit pheromone, only the $e < m$ best-ranked ants are chosen [Dorigo and Stützlea, 2004]. The ant that resulted in the best-so-far tour is selected to deposit pheromone too. Therefore, the pheromone update equation in AS_{rank} is given by:

$$\tau^n_{(v_i,v_j)} = \hat{\tau}^n_{(v_i,v_j)} + \sum_{k=1}^{e} (e-k+1)\Delta\tau^k_{(v_i,v_j)} + e\Delta\tau^{\text{bs}}_{(v_i,v_j)}, \tag{7.9}$$

where $\Delta\tau^k_{(v_i,v_j)}$ and $\Delta\tau^{\text{bs}}_{(v_i,v_j)}$ are given in Equation 7.6 and Equation 7.8 respectively.

7.4.4 Max-Min Ant System

The MMAS was introduced by Stüetzle and Hoos [1997] to modify the original AS. The algorithm allows either the best-so-far ant or the best-iteration ant (the ant that produced the best tour in the current iteration) to deposit pheromone. To prevent the early stagnation (all the ants follow a particular tour), the pheromone level is limited to the range $[\tau_{\min}, \tau_{\max}]$. When the algorithm starts, the pheromone levels are initialized to $\tau_{\max}$. If the algorithm reaches a stagnation state or there is no improvement in the solution, the pheromone level is reinitialized. The value of $\tau_{\max}$ is estimated from the best-so-far tour solution, i.e. $\tau_{\max} = \frac{1}{R_{\text{bs}}}$. In other words, the value of $\tau_{\max}$ is

updated each time a new best-so-far solution is found. The lower pheromone levels is estimated to $\tau_{\min} = \frac{\tau_{\max}}{b}$, where b is a parameter that can be tuned to prevent the stagnation [Dorigo and Stützlea, 2004]. The equation of pheromone evaporation is the same as Equation 7.4, while pheromone updating is given by:

$$\tau^{n}_{(v_i,v_j)} = \tau^{n-1}_{(v_i,v_j)} + \Delta^{\text{best}}_{(v_i,v_j)}, \tag{7.10}$$

where

$$\Delta^{\text{best}}_{(v_i,v_j)} = \begin{cases} \frac{1}{R_{\text{bi}}}, & \text{if best-iteration tour is chosen;} \\ \frac{1}{R_{\text{bs}}}, & \text{if best-so-far tour is chosen.} \end{cases} \tag{7.11}$$

The algorithm alternate using the best-so-far and the best-iteration updating rules. The frequency of the alternating between these rules is related to the problem size. It has been shown for small combinatorial problems that the algorithm gives better results when using only best-iteration tour. With larger combinatorial problems, the best-so-far tour gives better results [Dorigo and Stützlea, 2004].

7.5 CHAPTER SUMMARY

A combinatoric model for the MCM is found in this chapter. The model is a graph which is called the demand graph and the MCM problem become an arc routing problem. Finding complete tours with minimum LD and LO solves the MCM problem. Some of the real life arc routing problems are considered to find the class of the MCM problem. The considered arc routing problems are the Chinese Postman Problem, Capacitated Chinese Postman Problem, Capacitated Arc Routing, Winter Gritting, and Dynamic Winter Gritting. The Chinese Postman Problem is the basic one and therefore the problem formulation is considered. The Capacitated Arc Routing is the general class for the Capacitated Chinese Postman and Winter Gritting problems. This class of problems arises when there is a need to service a number of costumers or roads using a fleet of vehicles under some constraints such as the vehicle capacity. Winter gritting is an example of such tours since each vehicle carries a certain amount of salt. This requires the planning of the tour for each vehicle to not violate the vehicle capacity. However, this plan could be changed because of changes in weather forecast. This is called dynamic behavior.

The development of the MCM model is started by observing that the problem has many candidate solutions which are possible realizations. These realizations are enumerated and the subexpressions are shared between them resulting in an acyclic graph named the decomposition graph. The next step in our development is to make this graph routable. This requires transforming the $\mathcal{A}-$operation graph to a subexpression graph. In other words, the shift information attached to the decomposition graph arcs is transformed to subexpression information. The resulting transformed graph is named the demand graph. The demand graph is augmented with dead heading arcs to make the touring on it possible. It is found that the touring on the demand graph has a dynamic behavior because of existing demands on its arcs. Each demand is a subexpression to

be synthesized. Servicing a demand requires to re-plan the tour in a similar way to the dynamic winter gritting.

The demand graph includes enormous number of candidate solutions and therefore it is a large scale graph. Evolutionary algorithms are the most efficient algorithms to search such a large space of solutions. An example of evolutionary algorithms is the Ant Colony Optimization algorithm. This algorithm is proposed in this work to search the demand graph in parallel. It includes distributing a number of computational ants over a set of graph vertices and letting each ant searches for a single solution. A probability formula is used by an ant to determine its next move which is determined by a heuristics value represents the cost of crossing an arc and the pheromone strength of the arc. Four basic ant colony algorithms are discussed which are the Ant System, Elitist Ant System, Rank based Ant System, and MAX-MIN Ant System.

The solutions of the MCM are found by constructing routes over the demand graph which are similar to the dynamic winter gritting routes described in Section 7.3. We have shown in the last chapter that winter gritting belongs to the capacitated arc routing problem (CARP). Parallel ant colony optimization (ACO) algorithms are used effectively to solve the CVRP (for example consider [Doerner et al., 2004]). Therefore it is straightforward to induce that the routing over the demand graph can be implemented easily on a multi-core computer cluster using ACO algorithms. This is because each ant agent can traverse the graph independently and can construct a complete route. In this chapter a parallel ant colony algorithm is proposed to search the demand graph. This preceded by a serial implementation of the algorithm using a single core platform to proof the validity of the routes and to adjust the pheromone parameters that would be used later in the parallel implementation.

This chapter is constructed as follows: Heuristics parameters for the ACO are defined in Section 8.1 to accommodate the MCM problem. The serial and parallel implementations of the ACO are presented in Section 8.2 and Section 8.3 respectively. A chapter summary is given in Section 8.4.

Chapter 8

ANT COLONY IMPLEMENTATION

The solutions of the MCM are found by constructing routes over the demand graph which are similar to the dynamic winter gritting routes described in Section 7.3. We have shown in the last chapter that winter gritting belongs to the capacitated arc routing problem (CARP). Parallel ant colony optimization (ACO) algorithms are used effectively to solve the CARP (for example consider [Doerner et al., 2004]). Therefore it is straightforward to induce that the routing over the demand graph can be implemented easily on a multi-core computer cluster using ACO algorithms. This is because each ant agent can traverse the graph independently and can construct a complete route. In this chapter a parallel ant colony algorithm is proposed to search the demand graph. This is preceded by a serial implementation of the algorithm using a single core platform to proof the validity of the routes and to adjust the pheromone parameters that would be used later in the parallel implementation.

This chapter is constructed as follows: Heuristic parameters for the ACO are defined in Section 8.1 to accommodate the MCM problem. The serial and parallel implementations of the ACO are presented in Section 8.2 and Section 8.3 respectively. A chapter summary is given in Section 8.4.

8.1 HEURISTICS PARAMETERS OF THE MCM

Equation 7.3 should be modified to become suitable for the routing over the demand graph. In the demand graph each vertex is a subexpression, therefore each vertex notation in Equation 7.3 is changed from v_i to s_i. The other important modification is that Equation 7.3 considers the case of searching a graph while the demand graph is a multigraph. In this case, each ant choses to cross one of the arcs $A_{(s_i,s_j)} = \{(s_i,s_j)_1, (s_i,s_j)_2, \cdots, (s_i,s_l)_1, (s_i,s_l)_2, \cdots\}$ that depart from subexpression s_i. Therefore the probability of an ant k crosses from s_i to s_j using arc $(s_i,s_j)_h \in A_{(s_i,s_j)}$ is given by:

$$p^k_{(s_i,s_j)_h} = \frac{[\tau_{(s_i,s_j)_h}]^\alpha [\frac{1}{\chi_{(s_i,s_j)_h}}]^\beta}{\sum_{s_l \in V^k_{s_i}, (s_i,s_l)_n \in A_{(s_i,s_l)}} [\tau_{(s_i,s_l)_n}]^\alpha [\frac{1}{\chi_{(s_i,s_l)_n}}]^\beta}, \text{ if } s_j \in V^k_{s_i} \text{ and } (s_i,s_l)_n \in A_{(s_i,s_l)} \text{ for } n = 1, 2, \cdots \quad (8.1)$$

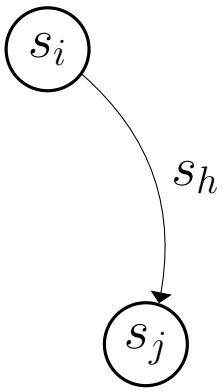


Figure 8.1: An ant cross arc $(s_i, s_j)_h$ with a demand of s_h.

where

1. $p^k_{(s_i,s_j)_h}$ is the probability of ant k to cross arc $(s_i, s_j)_h$ which connects the subexpressions s_i and s_j and $h = 1, 2, \cdots$ is arc rank.

2. $\tau_{(s_i,s_j)_h}$ is the pheromone strength on arc $(s_i, s_j)_h$.

3. $\chi_{(s_i,s_j)_h}$ is the complexity (heuristics value) of arc $(s_i, s_j)_h$ which is calculated as shown in Equation 8.2.

4. α and β are two parameters which determine the relative influence of the pheromone trail and the heuristic information

5. $A^k_{(s_i,s_l)}$ is a feasible arc set which does not violate the logic depth constraint when an ant k is being at vertex s_i and want to cross to vertex s_l.

6. $V^k_{s_i}$ is the set of unvisited vertices for ant k when being at vertex s_i.

Consider arc $(s_i, s_j)_h$ that shown in Figure 8.1. It describes the situation of an ant tries crossing from the subexpression s_i to synthesize s_j via an arc of demand s_h. The complexity of this arc is the complexity of its demand s_h which is given by:

$$\chi_{(s_i,s_j)_h} = \pi + L_{s_h} * \mathcal{H}_{s_h} + L_{s_h} \tag{8.2}$$

where L_{s_h} and $\mathcal{H}_{s_h}$ are respectively the wordlength and Hamming weight of the demand s_h, π is the priority of crossing arc $(s_i, s_j)_h$ and calculated according to Table 8.1. The quantities in this table are: W is the set of coefficients waiting for synthesis, T is the set of traversed vertices (synthesized subexpressions), $1 \leq a \leq 1$ are constants used to determine the priorities.

Table 8.1: Priority values of the demand s_h associated with arc $(s_i, s_j)_h$. W is the set of subexpressions waiting for synthesis, T is the set of synthesized subexpressions, $a > b > 1$ are constants.

s_i	s_h	s_j	π
T	T	W	∞
W	T	W	a
T	W	W	a
W	W	W	$\frac{a}{b}$
$\notin W \cup T$	T	W	$\frac{a}{b^2}$
T	$\notin W \cup T$	W	$\frac{a}{b^2}$
$\notin W \cup T$	W	W	$\frac{a}{b^3}$
W	$\notin W \cup T$	W	$\frac{a}{b^3}$
$\notin W \cup T$	$\notin W \cup T$	W	$\frac{a}{b^4}$
T	T	$\notin W$	$\frac{a}{b^5}$
W	T	$\notin W$	$\frac{a}{b^6}$
T	W	$\notin W$	$\frac{a}{b^6}$
W	W	$\notin W$	$\frac{a}{b^7}$
$\notin W \cup T$	T	$\notin W$	$\frac{a}{b^8}$
T	$\notin W \cup T$	$\notin W$	$\frac{a}{b^8}$
$\notin W \cup T$	W	$\notin W$	$\frac{a}{b^9}$
W	$\notin W \cup T$	$\notin W$	$\frac{a}{b^9}$
$\notin W \cup T$	$\notin W \cup T$	$\notin W$	0

8.2 SERIAL IMPLEMENTATION

It is shown in Section 7.3 that the solutions of MCM can be obtained by constructing routes on the demand graph. In this section, the MMAS ant system that described in Subsection 8.5 is implemented serially using single process. The MMAS was simulated using python high level language and run on a platform with 3.6 GHz Quad Core CPU and 8 GB RAM. Ants' routes are constructed by distributing them over vertices with $\delta = 1$. Each ant implements a re-directed depth first search as shown in Listing 6 to find a feasible complete route in which all the coefficients should be visited with a minimum cost. Graphs are implemented using Networkx. The demand graph was generated first as shown in step 4 of Listing 5 by using the subexpression tree algorithm described in Section 6.3. An ant agent is created (step 6) as another graph which grows when the ant crosses from one vertex to another. Pheromone levels are reinitialized to prevent early stagnation as shown in steps 16 and 17. Pheromone levels are reinitialized when

their values on some arcs became greater than an upper limit of $\tau_{\max}$ or less than a lower limit of $\tau_{\min}$. The reinitialize is accomplished by adding a small increment, Δ, to the initial pheromone levels which is work as an auto tunning to these levels. A choice of this increment is to be the reciprocal of best-so-far tour length, i.e. $\Delta = \frac{1}{R_{bs}}$.

```
1: main
2: input H, LD, α, β, τ_max                                   ▷ Input parameters
3: W ← Prepare(H)                                             ▷ Make the coefficients unique
4: DemandGraph ← GenerateDemand(W, LD)                        ▷ Generate the demand graph
5: DemandGraph ← Initialize(DemandGraph, τ_0)                 ▷ Initialize pheromone levels
6: ANTS ← GenerateAnts(DemandGraph, M, W)                     ▷ M ant graphs being stored in
                                                                ANTS array
7: while (true) do
8:     ANTS = DistributeAnts(DemandGraph, ANTS, W, LD, α, β)  ▷ Distribute the
                M ants over the coefficients with δ = 1 and start depth first search.
9:     for each ant ∈ ANTS do
10:        if PathLength(ant) < bs then                       ▷ Check for best-so-far tour
11:            bs = PathLength(ant)                           ▷ Save the new best-so-far tour cost
12:            ant^bs = ant                                   ▷ Best-so-far ant graph
13:    DemandGraph ← PheromoneEvaporate(DemandGraph, ANTS)    ▷ Pheromone
                                                                evaporation phase
14:    DemandGraph ← PheromoneUpdate(DemandGraph, ANTS)       ▷ Pheromone
                                                                updating phase
15:    for each edge ∈ DemandGraph do
16:        if τ_edge = τ_max then  ▷ Check if the pheromone level exceed the maximum
                                     levels
17:            DemandGraph ← Initialize(DemandGraph, τ_0 + 1/bs)   ▷ Re-initialize
                                                                     pheromone levels
18:    if bs = LowerLimit then  ▷ Stop search once finding the minimum realization
19:        break
20: end main
```

Listing 5: The implementation of MMAS ant system.

An ant calculates its next move by determining the arc with minimum heuristic value and maximum pheromone level. It do this by calling a 'sort' function that sorts arcs according to their complexity determined from heuristic value and pheromone level. The function sorts the arcs in ascending order of complexity and pushes them onto the stack in this order. Therefore an arc with minimum complexity will be at the top of the stack. A random shuffle is applied on the rest of the sorted edges to equally likely visit the other arcs and prevent any possible stagnation. If an ant crosses an arc with unsynthesized demand (as shown in step 17 of Listing 6), a predecessor map is searched recursively (steps 18-21) until finding first synthesized ancestors (step 22). This is actually the implementation of dynamic behavior. The popped edge in step 7 is pushed again onto the stack as shown in step 25. The out edges of the demand vertex are sorted according to step 26 then pushed onto the stack as shown in step 27. When an ant reaches a vertex that violates the logic depth constraint, it is used to prune the corresponding path that led to this vertex.

```
1: procedure AntTour(DemandGraph,ant, parent, W, LD, α, β)
2:     ant.W ← W                                              ▷ Coefficient set
3:     ant.T ← {1}                                            ▷ Synthesized coefficients set
4:     stack ← φ
5:     stack.push(parent.edges)                               ▷ Push parent out edges in stack
6:     while (stack.empty() = false) do
7:         (s1, s2, w) ← stack.pop()                          ▷ Pop an edge from the top of stack
8:         if w ∉ ant.T then
9:             δ_w ← max(δ_s1, δ_s2) + 1
10:            if δ_w ≤ LD then                               ▷ Check adder step
11:                if s2 ∈ ant.T then                         ▷ Synthesized demand
12:                    ant ← Synthesise(ant, w, s1, s2, ant.W, ant.T)   ▷ Synthesize w
13:                    if ant.W = φ then                      ▷ Stop search on empty coefficient set
14:                        return ant.T
15:                    w.edges ← SortEdges(w.edges, DemandGraph, α, β)   ▷ Sort out edges
16:                    stack.push(w.edges)
17:                else
18:                    predecessors ← Predecessors(DemandGraph, s2)   ▷ Get predecessor map if the demand is unsynthesized
19:                    pred ← predecessors.pop()
20:                    while (pred_s1 ∉ ant.T or pred_s2 ∉ ant.T) do   ▷ Find synthesized demand predecessors
21:                        pred ← predecessors.pop()
22:                    ant ← Synthesise(ant, s2, pred_s1, pred_s2, ant.W, ant.T)   ▷ Synthesise demand
23:                    if ant.W = φ then
24:                        return ant.T
25:                    stack.push(s1, s2, w)
26:                    s2.edges ← SortEdges(s2.edges, DemandGraph, α, β)
27:                    stack.push(s2.edges)
28:    return ant.T
```

Listing 6: An ant agent implemented as depth first search.

The parameters α and β were tuned experimentally to find the best setting for them. A set of 100 MCM with $N = 5$ and wordlengths $8 \leq L \leq 12$ was used. The percentage of the number of ants that found best-so-far tour was recorded against the values of α and β as shown in Figure 8.2. The number of generations (iterations) was 10 with each generation used 4 ants. A best setting is found when $\alpha = 1$ and $\beta = 2$. This result also coincides with the experiments of Dorigo and Stützlea [2004] (P71) about parameters setting for MMAS.

The MMAS ant system was tested using several filters found in the literature. The first used filter is F_1 [Farahani et al., 2010] which is of order $N = 4$. The initial pheromone level was adjusted to the value $\tau_0 = 0.125$ (after several experiments) whereas the maximum pheromone level was set to $\tau_{\max} = 1$. Therefore pheromone levels are re-initialized every eight generations. However, pheromone evaporation and updating are implemented as shown in Chapter 7. The number of ant agents was 17

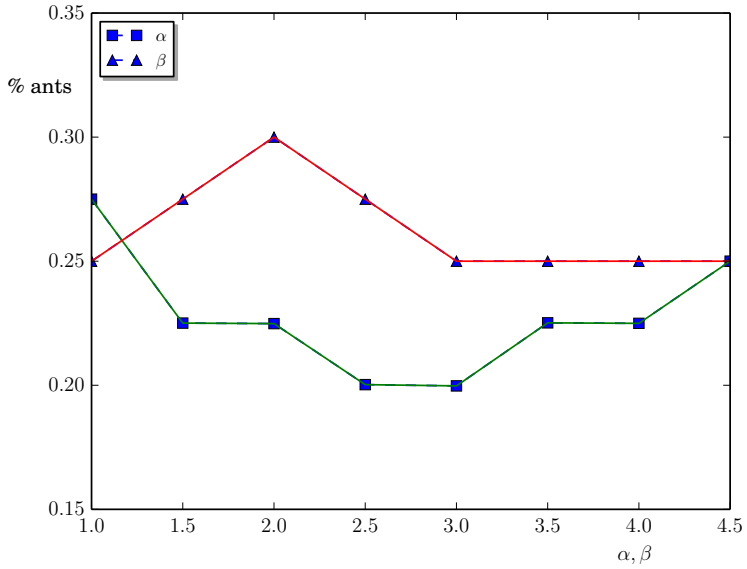


Figure 8.2: Percentage of ants found the best-so-far tour against the parameters α and β. A set of 100 MCM with $N = 5$ and wordlengths $8 \leq L \leq 12$ was used. The number of generations (iterations) is 10 and each generation used 4 ants.

which equals the out-degree of the depot vertex. This number was reduced to 5 ants to accelerate the program. The results of synthesizing the filter F_1 are shown in Table 8.2. In this case, eight solutions were found for the filter under the constraint LD = 3. Only five out of the eight solutions were distinct. The time of finding each solution is recorded as shown in Table 8.2. The last two columns in the table show the size of the demand graph. The same procedure was carried on the filter F_1 under logic depth constraint of LD = 2 for the same pheromone parameters. A total of six solutions were recorded with only two of them were distinct. The solution time was faster than that of LD = 3 because the size of the demand graph is smaller for shorter logic depth constraints.

Table 8.2: Synthesis the filter F_1 [Farahani et al., 2010] using MMAS ant system.

LD	LO	Generation	Solutions	Ant			t(s)			Vertices	Edges
3	6	1	2	3	4	-	245	286	-	336	289776
		2	2	2	4	-	556	678	-		
		3	2	2	3	4	819	825	873		
		4	3	2	3	4	1105	1525	2215		
2	7	1	0	-	-	-	-	-	-	18	68
		2	3	1	2	4	0.36	2.7	4.7		
		3	1	1	-	-	-6.9	-	-		
		4	2	2	4	-	18.26	29.47	-		

Another filter with coefficient set $\{21, 53, 171\}$ was synthesized using the MMAS as shown in Table 8.3. The result of the last row of the table shows that the number of solutions was increased to 2 at the fourth generation. In other words, increasing

the number of ant generations would increase the opportunity for the ants to explore the search space and find other minimum solutions. This also related to the stochastic behavior of the algorithm which is due to the pheromone updating and the random shuffling of output edges.

Table 8.3: Synthesis the filter $\{21, 53, 171\}$ using MMAS ant system.

LD	LO	Generation	Solutions	Ant			t(s)			Vertices	Edges
4	4	1	1	4	-	-	178	-	-	112	169344
		2	0	-	-	-	-	-	-		
		3	1	1	-	-	514.4	-	-		
		4	1	1	-	-	863.7	-	-		
3	4	1	1	3	-	-	33.7	-	-	209	121473
		2	1	1	-	-	77.6	-	-		
		3	1	1	-	-	151.8	-	-		
		4	2	1	2	-	226.3	228.53	-		

The MMAS was compared with the algorithms: MITM [Farahani et al., 2010], DBAG [Gustafsson, 2007a], and Yao [Yao et al., 2004]. The used filter examples were F_1 and F_2 [Farahani et al., 2010], and S_2 [Samueli, 1989]. The result of this comparison is shown in Table 8.4. It shows that the MMAS ant system has a performance similar to the STA algorithm in synthesizing short logic depth filters as in the cases of filters F_1 and S_2 Table 8.4. However, increasing the logic depth increases the size of search space and the time required to search it. For example, it is found that the demand graph of the filter S_2 has 1040 vertices and 1935407 edges when the logic depth constraint is relaxed to LD = 3. In this case it is found that the average time for an ant to complete its tour is 2 hours and this time would be increased to be more than one day when using 17 ants for example. Fortunately, the serial simulation is mainly to verify that the demand graph is the combinatorial model of the MCM and to investigate the ant colony optimization algorithm. These two targets are now assured and the next step is to implement the MMAS algorithm in parallel on a multi-core computer cluster.

Table 8.4: A comparison between the MMAS ant system, and the algorithms DBAG Gustafsson [2007a], Yao Yao et al. [2004], and STA[da] used to realize the filters F_1 Farahani et al. [2010], F_2 Farahani et al. [2010], and S_2 [Samueli, 1989] .

Filter	N	DBAG			Yao			STA[da]			MMAS		
		LD	LO	t(s)	LD	LO	t(s)	LD	LO	t(s)	LD	LO	t(s)
F_1	4	4	6	0.04	3	7	0.1	3	6	0.78	3	6	245
								2	7	0.54	2	7	0.36
F_2	4	5	6	0.17	3	6	0.17	3	6	0.76	3	6	1916.4
S_2	60	5	26	4.6	3	26	2.8	3	26	0.28			
					2	28	1.6	2	28	10.52	2	28	1.4

The serial implementation of MMAS algorithm was compared with the PPA and STA using a set of 100 MCM with $N = 5$. Minimum logic depth constraint is used in the STA and MMAS algorithms. A best adder saving performance was obtained from using the MMAS as shown in Figure 8.3. This is because ACO search a large space of candidate solutions and there is higher possibility to find one or more optimal solutions (if exist) because of using multi computational agents to search the demand graph. However, searching such a large space in sequence by ant agents make the performance of the serial MMAS algorithm in terms of run time poorer than that of PPA and STA as shown in Figure 8.4. Another comparison of the LO was made between the proposed algorithms PPA, STA, and MMAS and the algorithms DBAG and Yao as shown in Figure 8.5. The results show that the MMAS performance is as good as the DBAG method. However, the MMAS shows a worst run time than the other algorithms as shown in Figure 8.6. This is because the MMAS uses a number of ant agents to search the solution space. The LD was compared between the algorithms MMAS, PPA, and DBAG as shown in Figure 8.7. The LD was used as a constrained at the top of the in the case of MMAS. The other two methods of PPA and DBAG were unconstrained and the values of LD are obtained after solving the problem. The MMAS shows it is the best one among other algorithms because it can minimize the LO and LD simultaneously.

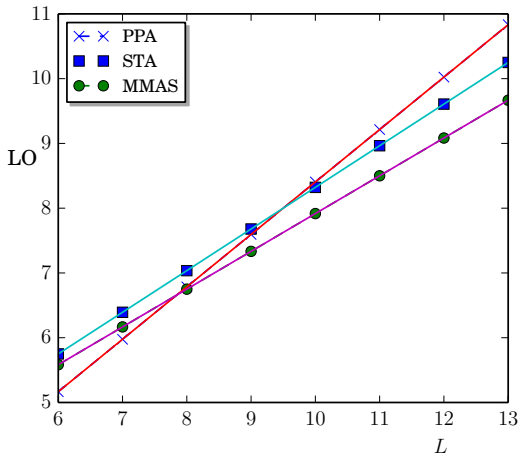


Figure 8.3: A comparison between the number of adders obtained from using the algorithms PPA, STA, and the serial implementation of MMAS used to synthesize random MCMs of 5 coefficients.

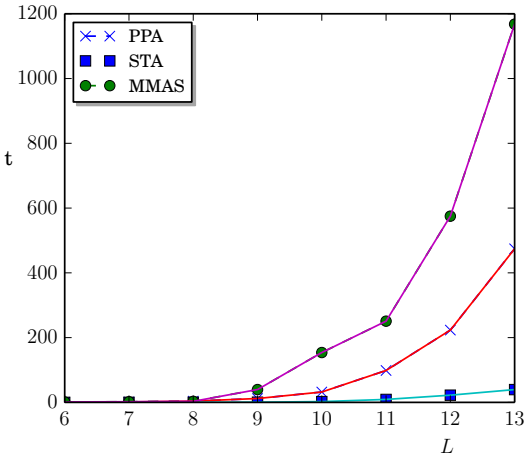


Figure 8.4: The run time comparison between the algorithms PPA, STA, and the serial implmentation of MMAS used to synthesize random MCMs of 5 coefficients.

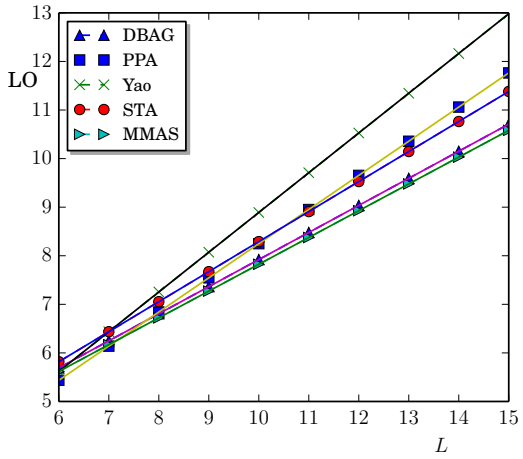


Figure 8.5: A comparison between the number of adders obtained from using the algorithms DBAG [Gustafsson, 2007a], PPA, Yao [Yao et al., 2004], STA, and the serial implmentation of MMAS used to synthesize random MCMs of 5 coefficients.

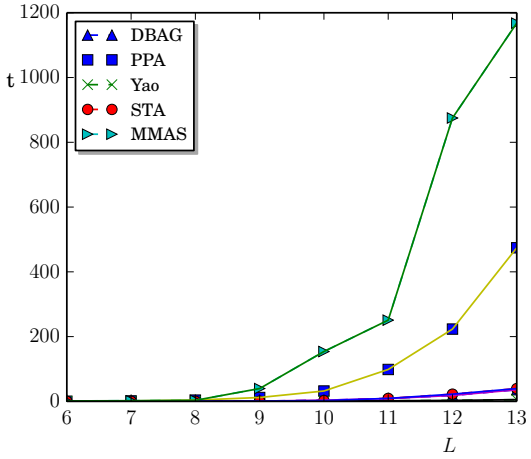


Figure 8.6: A comparison between the run time of the algorithms DBAG [Gustafsson, 2007a], PPA, Yao [Yao et al., 2004], STA, and the serial implmentation of MMAS used to synthesize random MCMs of 5 coefficients.

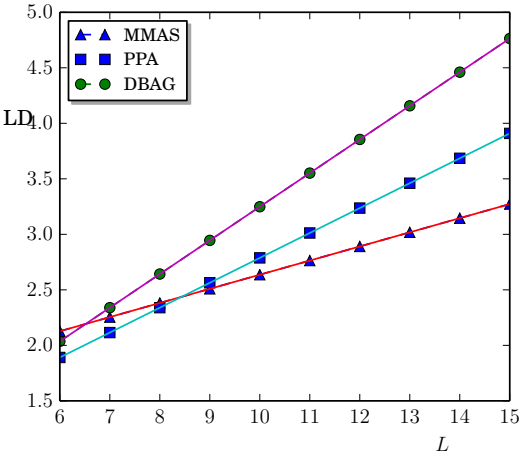


Figure 8.7: A comparison between LD for the methods MMAS, PPA, and DBAG [Gustafsson, 2007a].

8.3 PARALLEL IMPLEMENTATION OF ANT COLONY ALGORITHM

The parallel MMAS algorithm was implemented using the Boost Graph Library (PBGL) which is a C++ template graph library Siek et al. [2002] Edmonds et al. [2009]. Parallel computing was carried out using the IBM Power755 cluster of the super computer BlueFern of Canterbury University Mencl [2009]. The IBM Power755 cluster contains 11 Linux nodes, each one contains 128 Gigabytes of memory and 32 3.3 GHz IBM Power7 processor cores. The demand graph is generated on three stages of to speed up the computation time. Firstly, the representation tree of each subexpression is generated described in Section 5.1. Secondly, each representation tree is traversed using depth first search (DFS) algorithm Edmonds et al. [2009]. The DFS is modified to work as the subexpression tree algorithm proposed in Section 6.3. Subexpressions calculation is distributed between the processes. Therefore, the demand graph is partitioned into n parts, where n is the number processes. Each process save its part to the disk independently to avoid any communication between the processes. The third and final stage implies reading the n parts from the disk using only one processes. The single process combine the graph parts into one container of the demand graph.

The C++ Eigen template library for linear algebra was used to handle fast vector operations for calculating the subexpressions Free Software [2013].

The demand graph is used as a common data structure between the processes. Each processes is implement a computational ant which search the demand graph independently. Listing 7 shows the implementation of the parallel MMAS algorithm. Steps 9-12 show that each ant is assigned a process rank. Ants depart from subexpressions with $\delta = 1$ in similar way to that described in Section 8.2. The difference is the ants now search the graph concurrently. At each new iteration a random shuffle is applied on the departure subexpressions (step 8) to prevent any possible stagnation. An ant has four containers as shown in steps 14-16 and step 19 of Listing 7. These containers are as follows:

1. ant$[m].w$ contains subexpressions waiting for synthesizing.
2. ant$[m].t$ contains synthesized subexpressions.
3. ant$[m].r$ contains serviced arcs.
4. ant$[m].stack$ touring stack.

If an ant reached vertex v_k, it starts calculating its next move using a 'sort' function that sorts the out arcs of v_k in ascending order of complexity, the pushes these arcs onto the stack in this order (step 19). If it chosen to cross an arc with unsynthesized demand (as shown in steps 28 and 32 of Listing 7), a predecessor map is searched recursively (steps 29 and 33) until finding the first synthesized ancestors. This requires pushing again the last popped arc onto the stack (step 30). At this point the tour is redirected to the ancestors of the demand as shown in steps 31 and 35 of Listing 7. The Synthesize function in step 25 checks the adder step of the vertex w before an ant move beyond this

vertex. When an ant reaches a vertex that violates the logic depth constraint, it is used to prune the corresponding path that led to this vertex. Punning also includes paths that led to leaf subexpressions not in W (subexpressions waiting for synthesis). Costs of ant tours are calculated and compared using the function Cost which is called in step 49 of Listing 7. This function return the value of best-so-far tour (bs). Pheromone evaporation and updating functions are called in steps 50 and 51 respectively. Each function is implemented to loop over the graph arcs and change the pheromone levels according to the pheromone updating formulas given in Equation 7.4 and Equation 7.10.

```
1: main
2: input H, LD, α, β, τ_max, P                                   ▷ Input parameters
3: W ← Prepare(H)                                  ▷ Make the coefficients unique
4: G ← GenerateDemand(W, LD)                       ▷ Generate the demand graph
5: G ← Initialize(G, τ_0)                                  ▷ Initialize pheromone
6: ant ← ant.resize(M)                                ▷ Number of ants M ≤ P
7: for (k = 0; k ≤ 30; ++ k) do                              ▷ Number of iterations
8:     W_2 ← RandomShuffle(W_2)          ▷ Random shuffle of δ = 1 subexpressions
9:     start ← rank                      ▷ Distribute computation between processes
10:    stop ← start + 1
11:    for (m = start; m < stop; ++ m) do                        ▷ Ants departure
12:        w_2 ← W_2[m]                                ▷ Get departure vertex value w_2
13:        ant[m] ← ant[m].clear()                          ▷ Clear ant m containers
14:        ant[m].w ← W                              ▷ Initialize coefficients container
15:        ant[m].t ← {1}                ▷ Initialize synthesized coefficients container
16:        ant[m].r ← {}                             ▷ Initialize visited arcs container
17:        ant[m] ← Synthesize(ant[m], w_2, 1, 1, LD)                  ▷ Synthesis w_2
18:        ant[m].r ← ant[m].r + arc(1, w_2)          ▷ Add visited arc to ant m path
19:        ant[m].stack ← SortArcs(ant[m].stack, Vertex(w_2, G))  ▷ Best next move at
   the top of stack container
20:        while (ant[m].stack ≠ φ ∧ ant[m].w ≠ φ) do
21:            arc(s_1, w) ← Pop(ant[m].stack)      ▷ Extract s_1 and w from popped arc
22:            (s_2, τ) ← Weight(arc, G)                ▷ Find arc attributes, s_2 and τ
23:            if w ∉ ant[m].t then                                ▷ Unsynthesized vertex
24:                if s_1 ∈ ant[m].t ∧ s_2 ∈ ant[m].t then                ▷ Optimum case
25:                    ant[m] ← Synthesize(ant[m], w, s_1, s_2, LD)   ▷ Synthesis vertex w
26:                    ant[m].r ← ant[m].r + arc(s_1, w)
27:                    ant[m].stack ← SortArcs(ant[m].stack, Vertex(w, G))
28:                else if s_1 ∈ ant[m].t ∧ s_2 ∉ ant[m].t then
29:                    arc(p_1, p_2) ← predecessors(s_2, G) ▷ Find synthesized predecessors
   of s_2
30:                    ant[m].stack ← Push(ant[m].stack, arc(s_1, w))
31:                    ant[m].stack ← Push(ant[m].stack, arc(p_1, p_2))     ▷ Redirect the
   tour
32:                else if s_2 ∈ ant[m].t ∧ s_1 ∉ ant[m].t then
33:                    arc(p_1, p_2) ← predecessors(s_1, G)
34:                    ant[m].stack ← Push(ant[m].stack, arc(s_1, w))
35:                    ant[m].stack ← Push(ant[m].stack, arc(p_1, p_2))
36:            else if (ant[m].stack = φ ∧ ant[m].w ≠ φ) then     ▷ Check if all the
   required arcs have been serviced
37:                for each u_2 ∈ W_2 do                ▷ Paths for the non serviced arcs
38:                    if arc(1, u_2) ∉ ant[m].r then
39:                        ant[m] ← Synthesize(ant[m], u_2, 1, 1, LD)
40:                        ant[m].r ← ant[m].r + arc(1, u_2)
41:                        ant[m].stack ← SortArcs(ant[m].stack, Vertex(w_2, G))
42:                        break
43:    bs ← Cost(G, ant)                          ▷ Calculate the best-so-far tour cost
44:    G ← PheromoneEvaporate(G, ant)
45:    G ← PheromoneUpdate(G, ant, bs)
46:    for arc ∈ G do                                      ▷ Check for stagnation
47:        if τ_arc ≥ τ_max ∨ τ_arc ≤ τ_min then
48:            G ← Initialize(G, τ_0 + 1/bs)              ▷ Re-initialize pheromone levels
49:            break
50: end main
```

Listing 7: Parallel implementation of the MMAS algoritm.

The proposed parallel ant colony algorithm was used to synthesize the filters: S_2 (as given in Examples 2 of Samueli [1989]), L_1 (as given in Examples 1 Lim and Parker [1983]), and D_1 (as given in Dempster and Macleod [1995]). The umber of iteration was equal to 20. The result of using the algorithm to synthesize these filters is shown in Table 8.5. In this table, the time required to generate the demand graph is denoted by t_g and that required to search the graph is denoted by t_s. An ant were implemented to sort the arcs according to their heuristics values and pheromone levels and push them to the stack. The routing stops once the ant synthesize the set W. In addition, the parallel MMAS was implemented to consider the best-so-far route in pheromone updating. These modifications helped the algorithm to success in obtaining realizations with minimum logic depth constraint. The longer synthesis time required by the parallel MMAS is due to the stack were allowed to keep all the arcs in the neighbor of the vertex that an ant on it.

Table 8.5: Synthesis the filters S_2 , L_1, and D using the parallel MMAS algorithm. t_g and t_s are the generating and the search time respectively.

Filter	**# vertices**	**# edges**	**# ants**	t_g	t_s	**LD**	**LO**
S_2	45	503	22	3.5	2	2	28
D	649	861623	20	70	980	3	18
L_1	1203	2375596	24	420	3620	3	53

The advantage of implementing the MMAS algorithm on multi-core cluster over single process computer is the speed gain. The run time of parallel and serial implementations of the MMAS algorithm was compared as shown in Figure 8.8. Sets of MCM with $N = 5$ and variable wordlengths were used in this comparison. The parallel implementation shows better run time performance than the serial implementation. However, the former results a longer run time when compared with the algorithms STA, DBAG, and Yao as shown in Figure 8.9. This is expected since the stack growing issue that explained above is persist even in the parallel implementation.

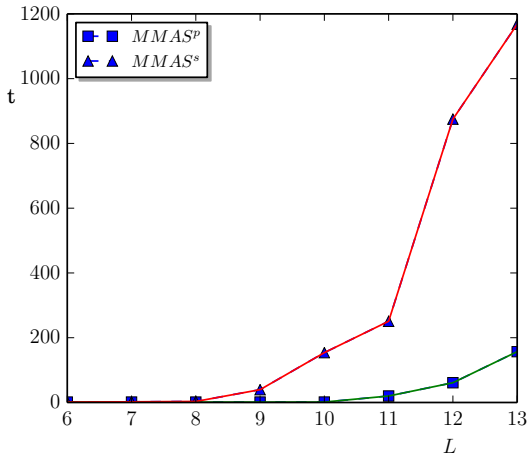


Figure 8.8: Run time comparison of the algorithm MMAS between its parallel implementation MMASp and the serial implementation MMASs.

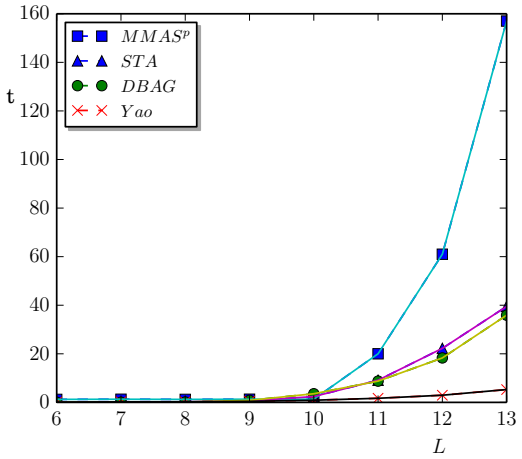


Figure 8.9: Run time comparison between the algorithms PPA, STA, and the parallel implmentation of MMAS ($MMAS^p$) when they used to synthesize random MCMs of 5 coefficients.

8.4 CHAPTER SUMMARY

The MMAS ant system is implemented in this chapter to solve the MCM. The first part of work was carried out using a single process PC and python language. The main purpose of this part is to prove that the routing over the demand graph can solve the MCM operation or alternatively to prove that the demand graph is the model of MCM operation. Several filters found in the literature are used for this purpose and to compare the algorithm performance with other algorithms. It is found that the run time grows rapidly when increasing the input size. However, the algorithm can find more than one solution (if exist) which is difficult to find them when using other heuristics methods.

The parallel implementation was carried out using the the IBM Power755 cluster and the C++ parallel boost graph library. The algorithm is implemented identically to the above serial one. However, the ants are now search the graph concurrently by assigning a process for each ant. This make the algorithm run faster than the serial one with the same quality of solution.

Chapter 9

CONCLUSIONS AND SUGGESTIONS FOR FUTURE WORK

The work in this thesis concerns using the redundant number representations with common subexpression elimination (CSE) algorithms to optimize the multiple constant multiplication (MCM). The CSE is preferred over the graph dependent (GD) methods because it searches a large space of subexpressions and therefore it can find MCM solutions with shorter logic depth (LD). The CSE methods are classified mainly into two types which are the Hartley's table and the multiplier block (MB) methods. The two methods are investigated in this research with greater focus on the MB method. A common subexpression elimination algorithm of Hartley's table type is proposed. The algorithm is called pattern preservation which is designed to work on the zero-dominant set (ZDS) redundant representations. The ZDS is larger than the minimum Hamming weight (MHW) representation that usually used in the literature. It is found that using the ZDS improves the performance of the CSE algorithm.

Achieving further improvement in the CSE algorithms' performance requires using larger sets of redundant representations. However, the improvement will be at the expense of increasing the CSE algorithm complexity. The CSE method is integrated with the process of generating the redundant representations to avoid any exhaustive search. A tree and graph encoders are proposed in this work to generate the representations. An efficient CSE algorithm called the subexpression tree algorithm (STA) is developed in this work to customize the tree encoder by integrating the search for the common subexpressions with the process of representations generation. The novelty in this algorithm is it trims the representation tree when it finds subexpressions with maximum sharing.

It is shown that the MCM problem is NP-hard. Solving such a combinatorial problem may require using metaheuristics algorithms whereas both the CSE and GD methods are heuristics. In other words, to find the optimum solution for any MCM in polynomial time requires developing the combinatorial model of the MCM operation. We have succeed in this work in developing this model by enumerating all the solutions of MCM then combining these solutions resulting a cyclic multigraph called the demand graph. Actually, enumerating all the solutions is not necessary because this is equivalent to connecting the subexpressions with each others using the decomposition process described in Chapter 6. The demand graph is augmented with deadheading arcs to

make the touring on it possible. Ant colony optimization metaheuristics is proposed to search the graph for minimum solutions.

This chapter includes conclusions and suggestions for future work on each proposed CSE algorithm. It is structured as follows: Section 9.1 discusses the pattern preservation algorithm. Suggestions for future work regarding the graph encoder is given in Section 9.2. Conclusion and future work about the subexpression tree algorithm is given in Section 9.3. Section 9.4 includes conclusions about the ant colony algorithm.

9.1 PATTERN PRESERVATION ALGORITHM

A pattern preservation algorithm (PPA) was proposed in Chapter 3 by introducing a two pass search with digit set/reset feature to improve the Hartley's CSE methods. It was found that the added features improve the PPA detecting ability for overlapping patterns and common subexpression simultaneously. Comparison results with other algorithms showed that the PPA gives better or comparable results when compared with other algorithms found in the literature.

The PPA (similarly for other Hartley's table methods [Hartley, 1991, 1996]) tries to find similar patterns in the representations. The difficulty of specifying the number of common patterns in a representation increases the difficulty to constrain the logic depth. To resolve this drawback, a new number system should be developed to provide representations that can be easily partitioned into two patterns. Another suggestion is to enlarge the search space by using BSD representations. In this case, the algorithm should be modified to resolve a more complicated patterns overlapping found in BSD.

9.2 GRAPH ENCODER

The graph encoder presented in Chapter 5 can be costumized for CSE. One way to customize the graph encoder is to construct the it for each coefficient and search the individual graphs in consequence for shortest paths and have most common edges. Coefficients with $\delta = 1$ are added to the digit set to be in the edge set. If there is no coefficient with $\delta = 1$, trails are required to find the best numbers with $\delta = 1$ to be added to the digit set. To show how the customized graph encoder work, consider synthesizing the filter F_1 given in [Farahani et al., 2010]. Its coefficients are $\{25, 103, 281, 655\}$. Consider generating the BSD graph for each coefficient. Since there is no coefficient with $\delta = 1$, several trails are made to find the best digit set which is $\mathbb{D} = \{\overline{1}, 0, 1, 5, 15\}$. The shortest path for each coefficient are $\{25$: [5 15], 103 : [1 0 0 0 0 0 0 $\overline{25}$], 281 : [1 0 0 0 0 0 0 0 25], 655 : [5 0 0 0 0 0 0 15]$\}$. The resulting multiplier block of the filter is shown in Figure 9.1. However, adding digits to the digit set increase the branching factor and hence the size of the encoder graphs. The idea presented here can be developed in the future to implement a CSE method based on the graph encoder.

One way is search the encoder graphs for paths of minimum Hamming weight subexpressions with best subexpression sharing. Synthesized subexpressions are added to the digit set starting from that with adder step equals to $\delta = 1$. However, increasing

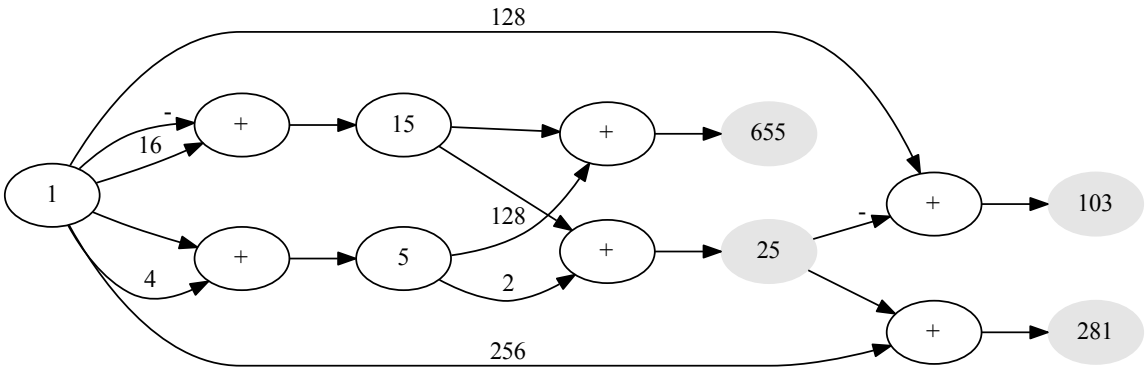


Figure 9.1: Synthesis of the filter F_1 in [Farahani et al., 2010] using the customized graph encoder results in a realization with LO = 6.

the digit set cardinality increases the encoding graph branching factor and consequently the complexity of the CSE algorithm. A suggestion is to use a multi-core platform and implementing a distributed Dijkstra's algorithm. One of choices to implement the distributed Dijkstra's algorithm is using the C++ parallel boost graph library (PBGL). The library includes the distributed Dijkstra's algorithm that finds shortest paths on a graph [Edmonds et al., 2009; Siek et al., 2002]. Another suggestion to reduce the algorithm complexity is to use a number system with adaptive radix to reduce the branching factor and therefore the size of search space.

9.3 SUBEXPRESSION TREE ALGORITHM

The results of using the subexpression tree algorithm (STA) showed that the possibility of finding MCM realizations of minimum logic depth is improved because the subexpression space became larger. Comparison with other algorithms supports this argument where the algorithm has a superior performance in realizing coefficients with wordlengths up to 16 bits. This is useful in realizing short wordlength DSP systems such as in mobile communications, where power consumption is the main concern. However, the algorithm can be modified in the future to synthesize longer wordlength coefficients. A possible choice is to use number system other than BSD by letting the digit set include the synthesized fundamentals (non-constant digit set). In this case, the search space could be reduced to the minimum Hamming weight. Another possibility to reduce the search space is by using a number system with variable radix and/or digit set.

The concept of co-prime subexpressions is introduced as a generalization for the positive odd coefficients terminology used in the literature with respect to the BSD representations. This helps to extend the work in the future to include other redundant number systems to reduce the search space. In addition, using redundant number systems provides the possibility of designing a carry free adder such as the compres-

sor. However, using the redundant adder could increase the cost of the MCM and a compromise between cost and speed should be made.

9.4 ANT COLONY OPTIMIZATION ALGORITHM

Two simulation methods were used in this work to search the demand graph, firstly a serial program in python language that works on a single process platform and secondly a parallel program that use the C++ parallel boost graph library. The serial implementation of the MMAS is designed to emulate the dynamic winter gritting. A swarm of fixed number of ants (called generation) are designed to depart from the coefficients with $\delta = 1$. The result showed that an increase in the number of ant generations increases the probability of exploring new routes and hence finding more than one minimum solutions for some MCM. Using random shuffle of output edges alongside with pheromone updating enhances the stochastic behavior of the algorithm to not stagnate at a local minimum.

The parallel implementation was carried out using the IBM Power755 cluster and the C++ parallel boost graph library. The demand graph is distributed over the cluster and a distributed BFS algorithm is used to make ants find shortest paths from the depot to the coefficients. The pheromone traces result from using the BFS help in reducing the graph size. The resulting reduced demand graph is searched again using the serial implementation described in Section 8.2. In this case, the resulting ant colony algorithm is called a MCM demand graph search using MMAS (MDM). The MDM algorithm is found to be stagnated at one of the minimum LD solutions because of dominating the heuristics value over the pheromone levels which prevent the algorithm from finding other minimum solutions. The work can be developed by mitigating the reduction ratio of demand graph and keep some of parallel arcs that could lead to other minimum solutions. However, the most novel work of proposing the demand graph and ant colony algorithm to search it was accomplished in this thesis. There are quite many directions for future research which include but not limited to:

1. Develop a new distributed directed depth search algorithm that emulate the dynamic winter gritting.

2. Tuning the reduction ratio of the demand graph by increasing the number of propagated solutions in the first reduction stage.

3. Tuning the number of eliminated parallel edges in the second reduction stage.

4. Tuning the heuristics and pheromone updating parameters.

5. Tuning the number and position of ant agents over the demand graph.

6. Compare the work with other ant system methods.

9.5 COMPARING THE PROPOSED ALGORITHMS

The MCM problem is an active research for more than two decades. Many heuristics have proposed to solve the problem. There are two potential and competitive heuristics which are the GD and the CSE methods. Our research focuses on the CSE. Three CSE algorithms are proposed in this work which are the PPA, STA, and MMAS. They are differ in the technique (the class of the CSE), the search space, and performance. This is true if we look at the three methods as unrelated separated algorithms. Actually there is a tight hierarchy research relation between them. The following research development steps shows this relation:

1. In the last few years, the research in CSE become a little sluggish because the method depends on number representations and going beyond the MHW is an adventure. We crossed to the hazardous region by using the ZDS first. The PPA is designed to search the ZDS representations tabulated in Hartleys' tables. The ZDS is an expanding of the MHW by few but important representations that provide larger space of patterns. The results were found promising as compared with the CSD and MHW which cause us jump to larger space of BSD representations.

2. The STA algorithm is developed to search the BSD. The STA itself is not just a CSE algorithm, it is actually a new style of CSE methods that incorporate the generation of representations with CSE. Other important developments accompanied the STA which are the subexpression space concept and generalizing the A-operation.

3. The relation between the coefficients and their decompositions pave the road to derive the combinatorial model of the MCM which is the demand graph. The search space is now the solutions of the MCM which is another new direction in the search space representation. The problem of MCM is now solved by finding tours of minimum length over the demand graph. This is a significant contribution in the field of CSE in that solving the MCM problem become an arc routing which has analogies in the operation research field.

4. The search showed that no matter how clever a GD or CSE method is, it may fail in finding an optimal solutions for some MCM instants without considering the whole space.

The technical differences between the three methods are shown in Table 9.1. The PPA works on Hartley's table method. The method may result in a short LD and minimum LO but the number of combinations is large. The main advantage of this method is it works on the canonical structure of the FIR filter. Therefore, the delay line is fused with the multiplier block which reduce the critical path and the pipelining cost.

The STA algorithm showed better performance than the PPA in terms of LO, LD, and run time. However, the STA is a HCSE-M method that use the transpose structure of the FIR filter. This isolate the multiplier block than the delay line which may increase

the pipelining cost. However, the STA algorithm is a genuine competitive of the best known graph methods. The algorithm solved problem scaled to 16 bit. This scale may be increased if there is a possibility to improve the implementation.

The last algorithm is the MMAS. The serial implementation of the algorithm is the one with worst run time result though it showed a better performance than PPA and STA in terms of LO and LD. However, the main objective of implementing the serial MMAS is to verify the correctness of the demand graph model and find the setting of the tunable parameters. The parallel implementation can solve the run time issue and scale the problem to search large size demand graphs. However, this work is developing now.

Table 9.1: A comparison between the PPA, STA, and MMAS algorithms.

Algorithm	**CSE Class**	**Search Space**
PPA	Hratley's HCSE	ZDS
STA	HCSE-M	BSD
MMAS	Not Classified (Arc Routing)	Demand Graph

Appendix A

GRAPH

A graph G consists of a set of vertices (nodes), $V = \{v_1, v_2, v_3, \cdots\}$, and a set of arcs (edges), $A = \{(v_i, v_j) : v_i, v_j \in V\}$.

A.1 BASIC TERMS

The following are some definitions of the graph theory [Vasudev, 2006]:

Definition A.1.1. Undirected graph G (Figure A.1) consists of a set V of vertices and a set A of arcs such that each arc in G is an unordered pair, i.e. $(v_i, v_j) = (v_j, v_i)$.

Definition A.1.2. A directed graph or digraph G (Figure A.2) consists of a set V of vertices and a set A of arcs such that each arc (v_i, v_j) is an ordered pair.

Definition A.1.3. A multigraph G (Figure A.3) is a graph where more than one arc can join tow vertices. A multidigraph G is a digraph where more than one arc can join two vertices.

For abbreviation the name graph would mean graph, digraph, multigraph, or multidigraph.

Definition A.1.4. The number of arcs incident on a vertex v_i is the degree of v_i. The in-degree of the vertex v_i is the number of arcs arriving at v_i. The out-degree of v_i is the number of arcs departing from v_i.

For example vertex 1 in Figure A.3 is with in-degree = 1, out-degree = 3, and total degree = 4.

Definition A.1.5. A graph G is symmetric if the in-dgree for each vertex v_i in G equal the out-degree of v_i.

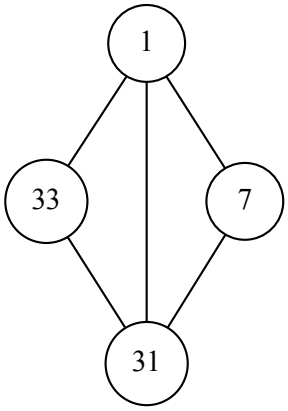


Figure A.1: An example of undirected graph G.

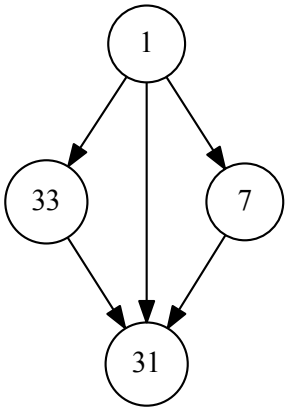


Figure A.2: An example of directed graph G.

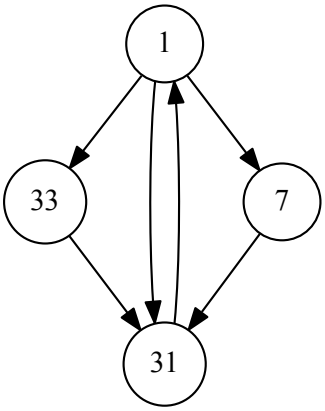


Figure A.3: An example of multidigraph G.

A.2 PATHS, CIRCUITS, AND CYCLES

A walk between vertex v_i and v_j in the graph G is a finite alternating sequence of vertices and arcs. For example, in the graph in Figure A.4 the sequence $(33, (33, 1), 1, (1, 7), 7)$ is a walk between vertex 33 and vertex 7 [Balakrishnan, 1997]. A trail in the graph G is a walk in which no arc is repeated. For example, the walk $(1, (1, 33), 33, (33, 31), 31)$ in the graph shown in Figure A.4 is a trail. A closed walk in a graph is a walk between a vertex and itself. For example, the walk $(33, (33, 1), 1, (1, 7), 7, (7, 1), 1, (1, 33), 33)$ in Figure A.4 is a closed walk. A circuit is a closed walk in which no arcs repeat. The closed walk $(1, (1, 7), 7, (7, 1), 1)$ in Figure A.4 is a circuit. A cycle is a circuit with no repeated vertices. For example, the circuit $(1, (1, 7), 7, (7, 31), 31, (31, 33), 33, (33, 1), 1)$ in Figure A.4 is a cycle.

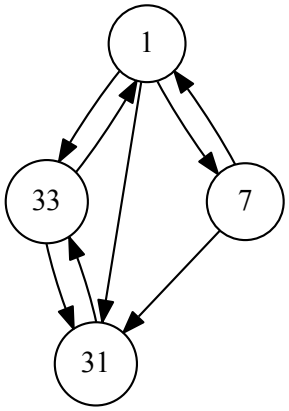


Figure A.4: A connected graph example.

A.3 CONNECTED GRAPH

A pair of vertices in a graph are connected if there is a path between them. A graph G is a connected graph if every pair of its vertices is a connected pair; otherwise, it is a disconnected graph [Balakrishnan, 1997]. An example of connected graph is shown in Figure A.4. The graph in Figure A.5 is disconnected because each vertex in the vertex set $V = \{1, 7, 31, 33\}$ is disconnected with each vertex in the vertex set $U = \{17, 43, 53\}$.

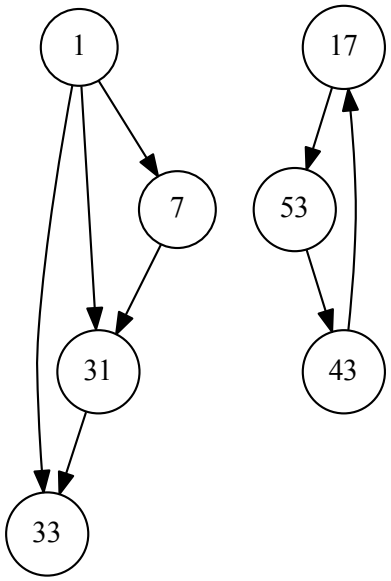


Figure A.5: An example of disconnected graph.

A.4 EULERIAN GRAPH

An Eulerian trail in a connected graph G is a trail between two distinct vertices in G that contains all the arcs of G. An Eulerian circuit in a graph G is a circuit that contains all the arcs of G. A unicursal graph is a graph that has a Eulerian trail. An Eulerian graph is a graph that has an Eulerian circuit.

For example, the graph in Figure A.6 is unicursal because it has an Eulerain trail between the vertices 1 and 7 which is $(1, (1, 15), 15, (15, 31), 31, (31, 15), 15, (15, 31), 31, (31, 7), 7, (7, 33), 33, (33, 17), 17, (17, 33), 33, (33, 17), 17, (17, 1), 1, (1, 7), 7)$. However, the graph in Figure A.6 is not an Eulerian graph [Balakrishnan, 1997]. The graph in Figure A.7 is an Eulerian graph because it has the Eulerian circuit $((1, 15), 15, (15, 31), 31, (31, 15), 15, (15, 7), 7, (7, 31), 31, (31, 17), 17, (17, 7), 7, (7, 1), 1, (1, 17,), 17, (17, 1), 1)$. That is, the circuit includes all the arcs of the graph and no arc is repeated more than once. In addition, the circuit is started with vertex 1 and ended with the same vertex.

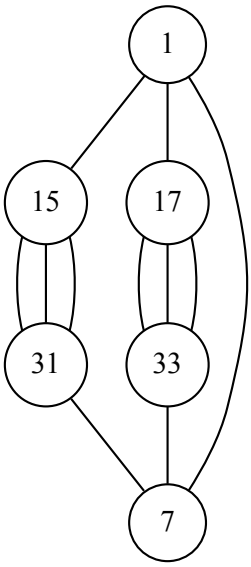


Figure A.6: An example of unicursal graph.

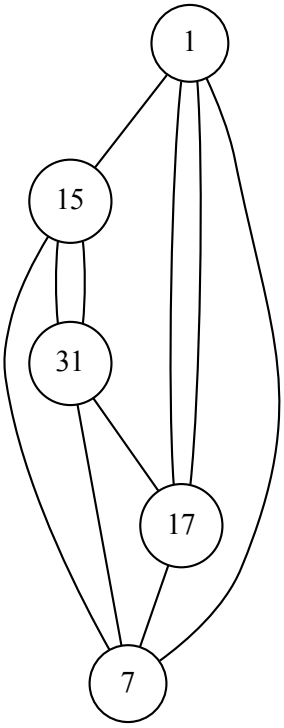


Figure A.7: An example of Eulerian graph.

Appendix B

ADDER

Computer arithmetic is based on the arithmetic operations of multiplication, division, subtraction, addition, and exponentiation. All these operations can be built using only addition. This makes the adder be the most important element in arithmetic circuits because it determines the speed, power consumption, and cost of these circuits. Since the carry propagation is inherent in the nonredundant positional number systems including binary, a binary adder, is also called carry propagate adder which slows down the speed of arithmetic circuits. There are many solutions to speed up the binary adder such as using the carry look-ahead, conditional sum, prefix , Ling, carry-select, carry-skip, and carry-save adders [Koren, 2002]. Other methods to remove the carry propagate include using redundant number system adder such as carry-save adder (CSA).

B.1 BINARY ADDER

Binary number system is well known and easy to use in computer hardware. This is because its digit set $\mathbb{D} = \{0, 1\}$ fits the two states of transistor switches (on-off). Consider adding two 4-bit binary numbers given by $(d_3d_2d_1d_0)_2$ and $(b_4b_2b_1b_0)_2$ respectively. The hardware implementation of this addition is shown in Figure B.1, where a_i and b_i are the binary bits of the inputs and c_i is the incoming carry. The function of the circuit in Figure B.1 can be expressed by the following Boolean equations:

$$w_i = a_i \oplus b_i \oplus c_i, \tag{B.1}$$

and

$$c_{i+1} = d_i b_i + t_i(d_i + b_i), \tag{B.2}$$

where $d_i \oplus b_i$ is the exclusive OR operation, w_i is the sum bit, $d_i b_i$ is the logical AND operation between the bits d_i and b_i, and $d_i + b_i$ is the logical OR operation between the two bits.

For example, consider adding two 4-bit binary numbers given by $(1101)_2$ and $(0011)_2$ as shown in Figure B.2. In this case, the addition operation results in a worst case of carry propagation because the result should wait until the end of carry ripple process. Therefore the delay increase with the wordlength, L, and this increment is in the order of $O(L)$.

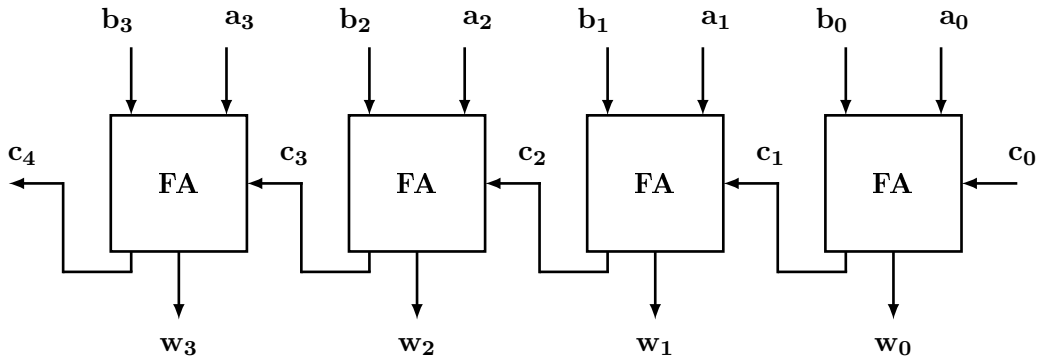


Figure B.1: 4-bit carry propagate adder.

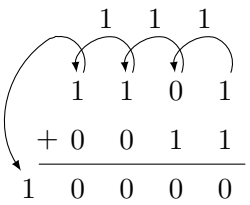


Figure B.2: Addition of two 4-bit operands.

B.2 CARRY-SAVE ADDER

The worst case of carry propagation that may result when adding two L bit binary operands is the carry ripples L stages before computing the output, where L is the operand wordlength. For three operands, carry propagation delay will be $2L$ and for k operands it will be $(k-1)L$. There are several methods to eliminate the carry propagation in multi-operand addition. The most common method is the carry save addition in which the carry propagates only in the last stage. This stage is implemented using a carry propagate adder (CPA), while other stages use carry save adders (CSA). An example of four 4-bit operand carry save addition is shown in Figure B.3. The addition of the first three operands is free of carry given the output sum digits cast in $[0,3]$. In binary, the digits in the range $[0,3]$ can be represented by using 2-bit signals (two operands). Thus, the addition of three operands using carry save format is a reduction from 3 to 2 and the corresponding CSA is called a $3:2$ counter. The name of counter is obvious from the result of conversion the first position sum from the range $[0,3]$ to the carry save digits $\{0,1,2\}$. Each digit in the position sum represents the number of ones that come to the full adder at that position. The same principle can be applied in the addition of the fourth operand, which is in this case will require another level of CSA as shown in Figure B.4. The last stage is the carry propagate adder which sums the last interim sums and the transfer digits to yield the final addition result in binary. The example of Figure B.3 shows that a carry has occurred out of the last stage.

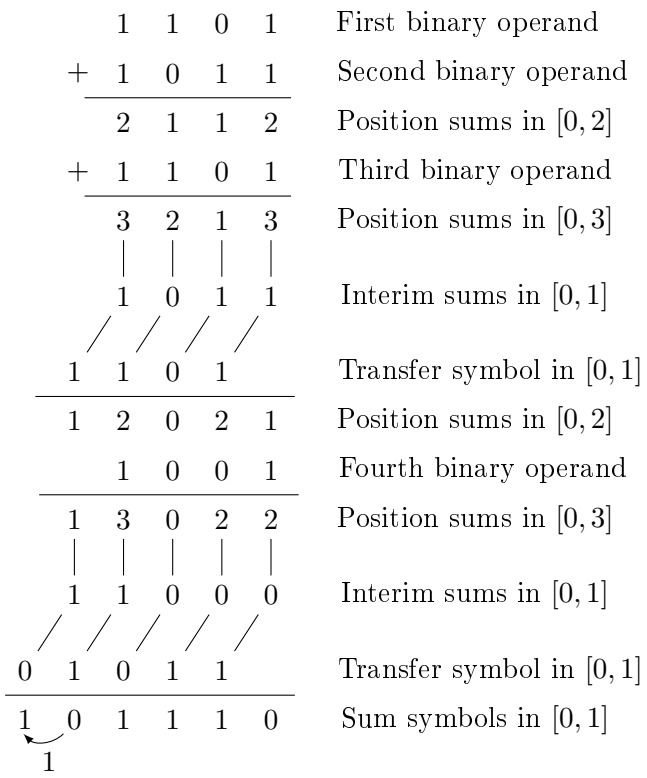


Figure B.3: Addition of four binary operands using carry save addition.

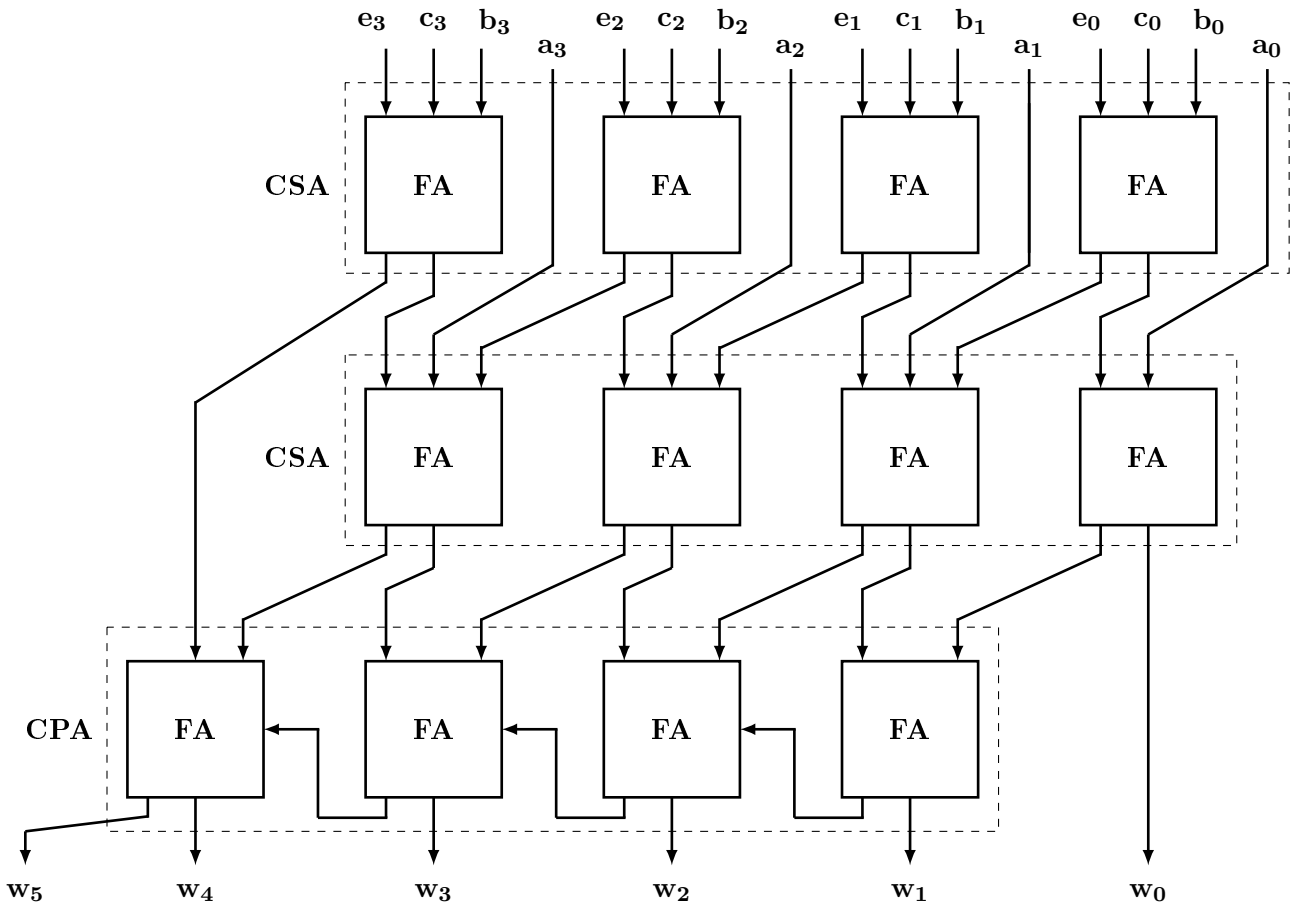


Figure B.4: Four operand carry-save adder (CSA). Note the final stage requires a carry propagate adder (CPA).

B.3 REDUNDANT ADDER

Although the CSA implies a carry free addition, the final stage of addition results in carry propagation. The speed of processing will deteriorate because of the carry propagation. This can be avoided by using the generalized signed-digit (GSD). The price of implementing redundant arithmetic is the requirement for more signals to represent the digits. For example, the BSD number system with $r = 2$ and $\mathbb{D} = \{\overline{1}, 0, 1\}$ requires two bits to encode these digits. A possible encoding of these digits is to use negative and positive bits (n, p). In this case, -1 is encoded as $(10)_2$, 0 is encoded as $(00)_2$, and 1 is encoded as $(01)_2$. The carry free addition algorithm for adding two GSD operands $\mathbf{A} = (\mathbf{d_{L-1}} \cdots \mathbf{d_0})_\mathbf{r}$ and $\mathbf{B} = (\mathbf{b_{L-1}} \cdots \mathbf{b_0})_\mathbf{r}$ is given by [Parhami, 1990],

1. Calculate the position sums $\mathbf{p_i} = \mathbf{d_i} + \mathbf{b_i}$.
2. Decompose the position sum $\mathbf{p_i}$ into a transfer digit $\mathbf{t_{i+1}}$ and an interim sum $\mathbf{e_i} = \mathbf{p_i} - \mathbf{r t_{i+1}}$.
3. Find the sum digits $\mathbf{w_i}$ from the addition of the interim sum with the coming transfer $\mathbf{w_i} = \mathbf{e_i} + \mathbf{t_i}$.

To specify the range of the interim sums and transfer digits, consider the GSD number system with a radix r and digit set in the range $[-\alpha, \beta]$. The values of the transfer digit $t_i \in [-\varepsilon, \eta]$. The permissible interim sum values should be in $[-\alpha + \varepsilon, \beta - \eta]$ to absorb the incoming transfer digit and to not produce a carry in the last step of the algorithm. Figure B.5 shows an example of carry free addition of the operands $(1\overline{2}03)_4$ and $(1\overline{1}\overline{1}2)_4$ with $r = 4$ and $\mathbb{D} = \{\overline{2}, \overline{1}, 0, 1, 2, 3\}$ or $\mathbf{w_i} \in [-2, 3]$. In this case, the transfer digit values should be bounded by the range $[-1, 1]$. The addition is performed in two steps limiting the carry propagation delay to only one position irrespective of the wordlength. The circuit diagram of the 4-digit redundant adder is shown in Figure B.6.

	1	$\overline{2}$	0	3	First operand digits $\mathbf{a_i} \in [-2, 3]$
+	1	$\overline{1}$	$\overline{1}$	2	Second operand digits $\mathbf{b_i} \in [-2, 3]$
	2	$\overline{3}$	$\overline{1}$	5	Position sums $\mathbf{p_i} \in [-4, 6]$
	2	1	$\overline{1}$	1	Interim sums $\mathbf{e_i} \in [-1, 2]$
0	$\overline{1}$	0	1		Transfer digits $\mathbf{t_i} \in [-1, 1]$
0	1	1	0	1	Sum digits $\mathbf{w_i} \in [-2, 3]$

Figure B.5: Adding two redundant 4-digit numbers with $r = 4$, $\mathbb{D} = \{\overline{2}, \overline{1}, 0, 1, 2, 3\}$, and $t_i = \{\overline{1}, 0, 1\}$.

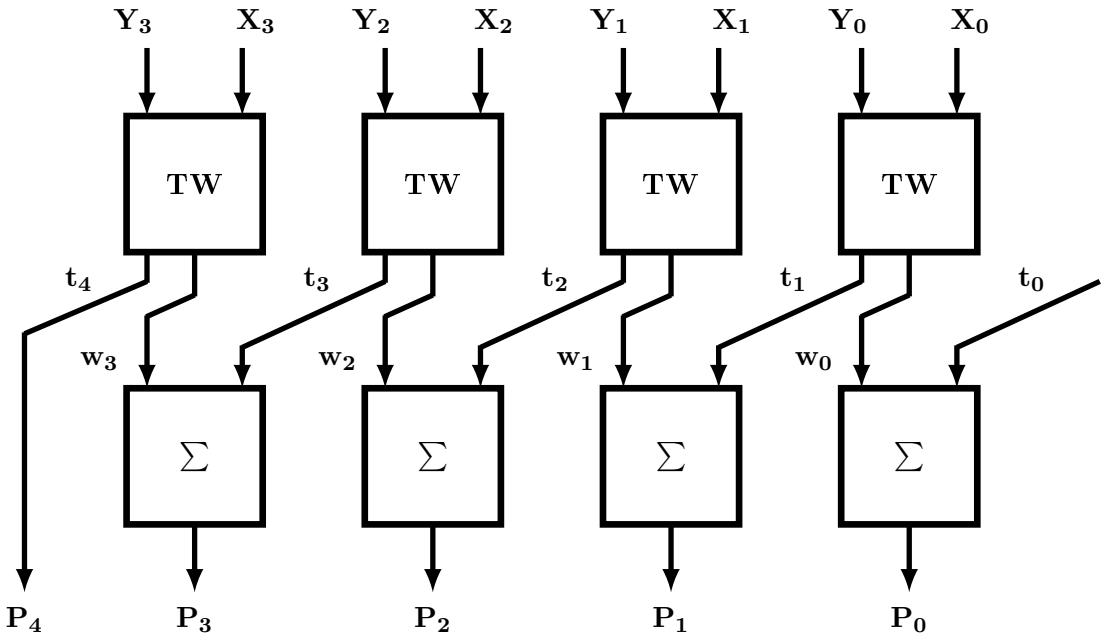


Figure B.6: An example of a 4-digit redundant adder. **TW** denotes the stage of decomposition of the position sum $\mathbf{P_i = w_i + rt_{i+1}}$.

Appendix C

ABSTRACT ALGEBRA

This appendix includes some of the fundamentals algebraic concepts that are used in the thesis.

C.1 CONGRUENCE

Definition C.1.1. Let $\mathbb{Z} = \{0, \pm 1, \pm 2, \cdots\}$ be the set of all integers, and $a, b, r \in \mathbb{Z}$, then a is said congruent to b modulo r (written $a \equiv b(\mod r)$) to mean r divides $(a-b)$, or alternatively $(a-b)$ is divisible by r [Hungerford, 1997]. Example of congruence relation is $11 \equiv 1(\mathrm{mod}\, 2)$ because $11 - 1 = 10$ is divisible by 2. Another example is $8 \equiv 2(\mathrm{mod}\, 3)$.

It is easy to show that the congruence relation is an equivalence relation because for all $a, b, r \in \mathbb{Z}$:

1. $a \equiv a(\mathrm{mod}\, r)$ (reflexive).
2. if $a \equiv b(\mathrm{mod}\, r)$, then $b \equiv a(\mathrm{mod}\, r)$ (symmetric).
3. if $a \equiv b(\mathrm{mod}\, r)$ and $b \equiv c(\mathrm{mod}\, r)$, then $a \equiv c(\mathrm{mod}\, r)$ (transitive).

Therefore, the congruence relation divides the set $\mathbb{Z}$ into equivalence (congruence) classes as shown in the next definition.

Definition C.1.2. If $a, r \in \mathbb{Z}$, then the congruence class of $a \mod r$ is

$$\overline{a} = \{b | b \in \mathbb{Z} \text{ and } b \equiv a(\mathrm{mod}\, r)\}, \tag{C.1}$$

which is the set of all integers that are congruent to $a(\mathrm{mod}\, r)$.

For example, the congruence class of 1 mod 3 is given by

$$\overline{1} = \{\cdots, -5, -2, 1, 4, 7, \cdots\},$$

i.e., the set of integers that when subtracting 1 from each, the result is divisible by 3. For example, $7 - 1 = 6$ is divisible by 3. Similarly, the classes

$$\overline{0} = \{\cdots, -6, -3, 0, 3, 6, \cdots\},$$
$$\overline{2} = \{\cdots, -4, -1, 2, 5, 8, \cdots\},$$

are congruence mod 3. The equivalence classes $\overline{0}, \overline{1},$ and $\overline{2}$ are distinct (disjoint) mod 3. All other classes are identical to one of these three distinct classes. For example, the class $\overline{4}$ is

$$\overline{4} = \{\cdots, -5, -2, 1, 4, 7, \cdots\} = \overline{1}.$$

The set that contain these three classes is denoted by $\mathbb{Z}_3$ and equals to

$$\mathbb{Z}_3 = \{\overline{0}, \overline{1}, \overline{2}\}, \tag{C.2}$$

which is also named the set of residual classes.

Definition C.1.3. The set of all congruence classes mod r (denoted $\mathbb{Z}_r$) is given by

$$\mathbb{Z}_r = \{\overline{0}, \overline{1}, \overline{2}, \cdots, \overline{r-1}\}. \tag{C.3}$$

The set $\mathbb{Z}_r$ contains exactly r distinct classes mod r $(\overline{0}, \overline{1}, \overline{2}, \cdots, \overline{r-1})$.

The sum of the classes $\overline{a}$ and $\overline{b}$ is $\overline{a} \oplus \overline{b} = \overline{a+b}$, where the symbol $\oplus$ represents the addition of residue classes. Example of addition in $\mathbb{Z}_3$ is $\overline{6} \oplus \overline{5} = \overline{6+5} = \overline{11} = \overline{2}$. The product of $\overline{a}$ and $\overline{b}$ is $\overline{a} \odot \overline{b} = \overline{a \cdot b}$. Example of classes multiplication in $\mathbb{Z}_3$ is $\overline{2} \odot \overline{5} = \overline{2 \cdot 5} = \overline{10} = \overline{1}$.

C.2 GROUPS

Definition C.2.1. A group $\mathcal{G}$ is a non empty set on which a binary operation $\circ$ is defined if for all $a, b, c \in \mathcal{G}$ the following properties hold [Jaisingh and Ayres, 2004]:

1. if $a, b \in \mathcal{G}$ then $a \circ b = c \in \mathcal{G}$. (closure for $\circ$)
2. $(a \circ b) \circ c = a \circ (b \circ c)$. (associative law)
3. There exists $u_{\mathcal{R}}$ such that $a + u_{\mathcal{R}} = u_{\mathcal{R}} + a = a$ for all $a \in \mathcal{G}$. (existence of an identity element)
4. For each $a \in \mathcal{R}$ there exists $a^{-1} \in \mathcal{R}$ such that $a \circ a^{-1} = a^{-1} \circ a = u_{\mathcal{R}}$. (existence of inverse)

The set $\mathbb{Z}$ of all integers is a group with respect to addition. The identity element is 0 and the inverse of a is $-a$ [Jaisingh and Ayres, 2004].

Definition C.2.2. A group $\mathcal{G}$ is called abelian group if for all $a, b \in \mathbb{G}$ then $a \circ b = b \circ a$ (the commutative law).

The set $\mathbb{Q}$ of all rational numbers is an abelian group with respect to multiplication. For example $3 \circ 4 = 4 \circ 3 = 12$, where $\circ$ represents the multiplication operation.

C.3 RING

Definition C.3.1. A non empty set $\mathcal{R}$ forms a ring with respect to the binary operations of addition $(+)$ and multiplication $(\cdot)$ if the following hold for any $a, b, c \in \mathcal{R}$:

1. $a+b, b+c, a+c \in \mathcal{R}$. (closure for addition)
2. $(a+b)+c = a+(b+c)$. (associative law of addition)
3. $a+b = b+a$. (commutative law of addition)
4. There exists $0_{\mathcal{R}}$ such that $a+0_{\mathcal{R}} = a$. (existence of additive identity)
5. For each $a \in \mathcal{R}$ there exists $-a \in \mathcal{R}$. (existence of additive inverse)
6. $(a \cdot b) \cdot c = a \cdot (b \cdot c)$. (associative law of multiplication)
7. $a \cdot (b+c) = a \cdot b + a \cdot b$ and $(a+b) \cdot c = a \cdot c + b \cdot c$. (distributive laws)

These axioms are the minimum requirements for an algebraic system (a set with the binary operations) to be a ring. Examples of ring are the sets $\mathbb{Z}$ and $\mathbb{Z}_r$ with the binary operations of addition and multiplication. These two rings have additional properties as shown in the next definitions.

Definition C.3.2. A ring $\mathcal{R}$ is called abelian (commutative) with respect to the multiplication operation if for $a, b \in \mathcal{R}$ the following hold:

$$a \cdot b = b \cdot a$$

Definition C.3.3. A ring $\mathcal{R}$ is called a ring with identity if it contains an element $1_{\mathbb{R}}$ with the following property:

$$a \cdot 1_{\mathbb{R}} = 1_{\mathbb{R}} \cdot a = a, \ \forall a \in \mathbb{Z}.$$

Definition C.3.4. A commutative ring $\mathbb{R}$ is called integral domain if the following hold for all $a, b \in \mathcal{R}$:

$$\text{if } a \cdot b = 0_{\mathcal{R}} \text{ then either } a = 0_{\mathcal{R}} \text{ or } b = 0_{\mathcal{R}}.$$

For example, the set $\mathbb{Z}_3$ is a commutative ring with zero element $\overline{0}$ and identity $\overline{1}$. The residual classes in $\mathbb{Z}_3 = \{\overline{0}, \overline{1}, \overline{2}\}$ have the addition and multiplication tables shown in Table C.1 and Table C.2 respectively. In addition, the set $\mathbb{Z}_3$ is an integral domain. It can be shown that all the sets $\mathbb{Z}_r$ are integral domain if r is a prime number. Consider a non prime congruence set $\mathbb{Z}_6$, the multiplication of the classes $\overline{2}$ and $\overline{3}$ equals $\overline{2} \cdot \overline{3} = \overline{6} = \overline{0}$ while $\overline{2} \neq \overline{0}$ and $\overline{3} \neq \overline{0}$.

Definition C.3.5. A subset $\mathcal{S}$ of $\mathcal{R}$ is a subring of $\mathcal{R}$ if itself forms a ring.

Table C.1: The addition table in $\mathbb{Z}_3$.

$+$	$\overline{0}$	$\overline{1}$	$\overline{2}$
$\overline{0}$	$\overline{0}$	$\overline{1}$	$\overline{2}$
$\overline{1}$	$\overline{1}$	$\overline{2}$	$\overline{0}$
$\overline{2}$	$\overline{2}$	$\overline{0}$	$\overline{1}$

Table C.2: The multiplication table in $\mathbb{Z}_3$.

$\odot$	$\overline{0}$	$\overline{1}$	$\overline{2}$
$\overline{0}$	$\overline{0}$	$\overline{0}$	$\overline{0}$
$\overline{1}$	$\overline{0}$	$\overline{1}$	$\overline{2}$
$\overline{2}$	$\overline{0}$	$\overline{2}$	$\overline{1}$

C.4 RING HOMOMORPHISM

If $\mathcal{R}$ and $\mathcal{R}'$ are rings, then a ring homomorphism is a function $f : \mathcal{R} \to \mathcal{R}'$ such that

1. $f(a+b) = f(a) + f(b)$ for all $a, b \in \mathcal{R}$.
2. $f(ab) = f(a)f(b)$ for all $a, b \in \mathcal{R}$.

REFERENCES

Aksoy, L., Costa, E., Flores, P., and Monteiro, J. (2011). Optimization of Gate-Level Area in High Throughput Multiple Constant Multiplications. *20th European Conference on Circuit Theory and Design, ECCTD 2011*, pages 588–591.

Aksoy, L., Costa, E., Flores, P., and Monteiro, J. (2012a). Optimization Algorithms for the Multiplierless Realization of Linear Transforms. *ACM Transactions on Design Automation of Electronic Systems*, 17(1).

Aksoy, L., Costa, E., Flores, P., and Monteiro, J. (2012b). Design of Low-Complexity Digital Finite Impulse Response Filters on FPGAs. In *Proceedings -Design, Automation and Test in Europe, DATE*, pages 1197–1202.

Aksoy, L., Günes, E. O., and Flores, P. (2010). Search Algorithms for the Multiple Constant Multiplications Problem: Exact and Approximate. *Microprocessors and Microsystems*, 34(5):151–162.

Al-Hasani, F., Hayes, M., and Bainbridge-Smith, A. (2011). A New Subexpression Elimination Algorithm Using Zero-Dominant Set. In *Proceedings - 2011 6th IEEE International Symposium on Electronic Design, Test and Application, DELTA 2011*, pages 45–50.

Avizienis, A. (1961). Signed-Digit Number Representation for Fast Parallel Arithmetic. *IRE Trans. Elect. Comput.*, EC-10:389–400.

Ayres, F. J. and Jaisingh, L. R. (2004). *Schaum's Outline of Theory and Problems of Abstract Algebra*. McGraw-Hill Companies, Inc., 2 edition.

Balakrishnan, V. (1997). *Schaum's Outlines Graph Theory*. McGraw-Hill Inc.

Banerjee, N., Choi, J. H., and Roy, K. (2007). A Process Variation Aware Low Power Synthesis Methodology for Fixed-Point FIR Filters. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 147–152.

Bull, D. and Horrocks, D. (1987). Reduced-Complexity Digital Filtering Structures Using Primitive Operations. *Electronics Letters*, 23(15):769–771.

Bull, D. and Horrocks, D. (1991). Primitive Operator Digital Filters. *IEE Proceedings, Part G: Circuits, Devices and Systems*, 138(3):401–412.

Bullnheimer, B., Hartl, R. F., and Strauss, C. (1999). A New Rank Based Version of the Ant System: A Computational Study. *Central European Journal of Operations Research*, 7(1):25–386.

Cappello, P. R. and Steiglitz, K. (1984). Some Complexity Issues in Digital Signal Processing. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-32(5):1037–1041.

Chang, C.-H. and Faust, M. (2010). On "A New Common Subexpression Elimination Algorithm for Realizing Low-Complexity Higher Order Digital Filters". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(5):844–848.

Colorni, A., Dorigo, M., and Maniezzo, V. (1991). Distributed Optimization by Ant Colonies. *European Conference on Artificial Life*, pages 134–142.

De Jong, K. (2006). *Evolutionary computation: A unified approach.* MIT Press, 1 edition.

Dempster, A., Dimirsoy, S., and Kale, I. (2002). Designing Multiplier Blocks with Low Logic Depth. In *Proceedings - IEEE International Symposium on Circuits and Systems*, volume 5, pages V–773–776.

Dempster, A., Macleod, M., and Gustafsson, O. (2004). Comparison of Graphical and Subexpression Elimination Methods for Design of Efficient Multipliers. *Thirty-Eighth Asilomar Conference on Signals, Systems and Computers*, 1:72–76.

Dempster, A. G. and Macleod, M. D. (1994). Constant Integer Multiplication Using Minimum Adders. *IEE Proceedings - Circuits, Devices and Systems*, 141(5):407–413.

Dempster, A. G. and Macleod, M. D. (1995). Use of Minimum-Adder Multiplier Blocks in FIR Digital Filters. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 42(9):569–577.

Dempster, A. G. and Macleod, M. D. (2004). Generation of Signed-Digit Representations for Integer Multiplication. *IEEE Signal Processing Letters*, 11(8):663–665.

Ding, Q., Hu, X., Sun, L., and Wang, Y. (2012). An Improved Ant Colony Optimization and Its Application to Vehicle Routing Problem with Time Windows. *Neurocomputing*, 98:101–107.

Doerner, K., Hartl, R., Kiechle, G., Lucka, M., and Reimann, M. (2004). Parallel Ant Systems for the Capacitated Vehicle Routing Problem. *Evolutionary Computation in Combinatorial Optimization: Lecture Notes in Computer Science*, 3004:72–83.

Dorigo, M. and Stützlea, T. (2004). *Ant Colony Optimization.* MIT Press, 1 edition.

Edmonds, N., Gregor, D., and Lumsdaine, A. (2009). Parallel boost graph library.

Eiselt, H., Gendreau, M., and Laporte, G. (1995a). Arc Routing Problems, Part I: The Chinese Postman Problem. *Operation Research*, 43(2):231–242.

Eiselt, H., Gendreau, M., and Laporte, G. (1995b). Arc Routing Problems, Part II: The Rural Postman Problem. *Operation Research*, 43(3):399–414.

Evans, J. R. and Minieka, E. (1992). *Optimization Algorithms for Networks and Graphs*. Maecel Dekker Inc., 2 edition.

Farahani, M., Guerra, E., and Colpitts, B. (2010). Efficient Implementation of FIR Filters Based on a Novel Common Subexpression Elimination Algorithm. *Canadian Conference on Electrical and Computer Engineering*, pages 1–4.

Faust, M. and Chang, C.-H. (2010). Minimal Logic Depth Adder Tree Optimization for Multiple Constant Multiplication . *2010 IEEE International Symposium on Circuits and Systems: Nano-Bio Circuit Fabrics and Systems, ISCAS 2010*, pages 457–460.

Free Software (2013). Eigen 3.2.0.

Gustafsson, O. (2007a). A Difference Based Adder Graph Heuristic for Multiple Constant Multiplication Problems. In *Proceedings - IEEE International Symposium on Circuits and Systems*, pages 1097–1100.

Gustafsson, O. (2007b). Lower Bounds for Constant Multiplication Problems. *IEEE Transactions on Circuits and Systems-II: Express Briefs*, 54(11):974–978.

Gustafsson, O. (2008). Towards Optimal Multiple Constant Multiplication: A Hypergraph Approach. *Asilomar Conference on Signals, Systems and Computers*, pages 1805–1809.

Gustafsson, O., Dempster, A. G., Johansson, K., Macleod, M. D., and Wanhammar, L. (2006). Simplified Design of Constant Coefficient Multipliers. *Circuits, Systems, and Signal Processing*, 25(2):225–251.

Gustafsson, O. and Wanhammar, L. (2002). Design of Linear-Phase FIR Filters Combining Subexpression Sharing with MILP. *Midwest Symposium on Circuits and Systems*, 3:III9–III12.

Hall, P. (2011). Power considerations for the Square Kilometre Array (SKA) radio telescope. In *General Assembly and Scientific Symposium*, pages 1–4.

Han, J.-H. and Park, I.-C. (2008). FIR Filter Synthesis Considering Multiple Adder Graphs for a Coefficient. *IEEE Transaction On Computer-Aided Design of Integrated Circuits and Systems*, 27(5):958–962.

Handa, H., Chapman, L., and Yao, X. (2005). Dynamic Salting Route Optimisation Using Evolutionary Computation. *2005 IEEE Congress on Evolutionary Computation, IEEE CEC 2005*, pages 158–165.

Hartley, R. I. (1991). Optimization of Canonic Signed Digit Multipliers for Filter Design. In *Proceedings - IEEE International Symposium on Circuits and Systems*, volume 4, pages 1992–1995.

Hartley, R. I. (1996). Subexpression Sharing in Filters Using Canonic Signed Digit Multipliers. *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, 43(10):677–688.

Ho, Y.-H. A., Lei, C.-U., Kwan, H.-K., and Wong, N. (2008). Optimal Common Subexpression Elimination of Multiple Constant Multiplications with A Logic Depth Constraint. *IEICE Trans. Fundamentals*, E91-A(12):3568–3575.

Hungerford, T. W. (1997). *Abstract Algebra: An Introduction*. Saunders College Publishing, 2 edition.

IEEESpectrum (2013). http://spectrum.ieee.org/.

Ifrah, G. (2001). *The Universal History of Computing: From the ABACUS to the Quantum Computer*. John Wiley and Sons, Inc., 1 edition.

Jaberipur, G. and Saeid, G. (2010). An Improved Maximally Redundant Digit Adder. *Computer and Electrical Engineering*, 36(3):491–502.

Jaisingh, L. R. and Ayres, F. (2004). *Schaum's Outline of Abstract Algebra*. McGraw-Hill Companies, Inc., 2 edition.

Johansson, K. (2008). *Low Power and Low Complexity Shift-and-Add Based Computations*. PhD thesis, Linköping University.

Johansson, K., Gustafsson, O., Debrunner, L. S., and Wanhammar, L. (2011). Minimum Adder Depth Multiple Constant Multiplication Algorithm for Low Power FIR Filters . In *2011 IEEE International Symposium of Circuits and Systems, ISCAS 2011*, pages 1439–1442.

Kamp, W. (2010). *Alternate Number Systems for Optimizing DSP Performance in FPGA*. PhD thesis, Canterbury University.

Kamp, W. and Bainbridge-Smith, A. (2007). Multiply Accumulate Unit Optimised for Fast Dot-Product Evaluation. *2007. International Conference on Field-Programmable Technology*, pages 349–352.

Kato, K., Takahashi, Y., and Sekine, T. (2009). A New Horizontal and Vertical Common Subexpression Elimination Method for Multiple Constant Multiplication. *2009 16th IEEE International Conference on Electronics, Circuits and Systems (ICECS 2009)*, pages 124–127.

Koren, I. (2002). *Computer Arithmetic Algorithms*. Library of Congress Cataloging-in-Publication data, 2 edition.

Kornerup, P. and Matula, D. W. (2010). *Finite Precision Number Systems and Arithmetics*. Cambridge University Press, 1 edition.

Kuo, Y.-T., Lin, T.-J., and Liu, C.-W. (2011). Complexity-Aware Quantization and Lightweight VLSI Implementation of FIR Filters. *Eurasip Journal on Advances in Signal Processing*, 2011.

Li, L. Y. and Eglese, R. W. (1996). An Interactive Algorithm for Vehicle Routeing for Winter-Gritting. *Journal of the Operational Research Society*, 47(2):217–228.

Lim, Y. C., Parker, S., and Constantinides, A. (1982). Finite Word Length FIR Filter Design Using Integer Programming Over A Discrete Coefficient Space. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-30(4):661–664.

Lim, Y. C. and Parker, S. R. (1983). Discrete Coefficient FIR Digital Filter Design Based Upon An LMS Criteria. *IEEE Transactions on Circuits and Systems*, CAS-30(10):723–739.

Mahesh, R. and Vinod, A. (2008). A New Common Subexpression Elimination Algorithm for Realizing Low-Complexity Higher Order Digital Filters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(2):217–229.

Marcos, M.-P., Boemo, E. I., and Wanhammar, L. (2002). Design of High-Speed Multiplierless Filters Using A Nonrecursive Signed Common Subexpression Algorithm. *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, 49(3):196–203.

Mathew, J., Mahesh, R., Vinod, A., and Lia, E. M.-K. (2008). Realization of Low Power High-Speed Channel Filters with Stringent Adjacent Channel Attenuation Specifications for Wireless Communication Receivers. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E91-A(9):2564–2570.

Mencl, V. (2009). Bluefern user documentation wiki.

Montemanni, R., Gambardella, L., Rizzoli, A., and Donati, A. (2002). A New Algorithm for a Dynamic Vehicle Routing Problem based on Ant Colony System. *Technical Report IDSIA-23-02, IDSIA*, pages 1–4.

Nielsen, A. M. (1997). *Number Systems and Digital Serial Arithmetic*. PhD thesis, Odense University Denmark.

Parhami, B. (1988). Carry-Free Addition of Recoded Binary Signed-Digit Numbers. *IEEE Transactions on Computers*, 37(11):1470–1476.

Parhami, B. (1990). Generalized signed-digit number systems: a unifying framework for redundant number representations. *IEEE Transactions on Computers*, 39(1):89–98.

Park, I.-C. and Kang, H.-J. (2001). Digital Filter Synthesis Based on Minimal Signed Digit Representation. In *Proceedings - 38th Design Automation Conference*, pages 468–473.

Paško, R., Schaumont, P., Derudder, V., and Vernalde, S.and Durackova, D. (1999). A New Algorithm for Elimination of Common Subexpressions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(1):58–68.

Pedemonte, M., Nesmachnow, S., and Cancela, H. (2011). A Survey on Parallel Ant Colony Optimization. *Applied Soft Computing Journal*, 11(8):5181–5197.

Phatak, D., Goff, T., and Koren, I. (2001). Constant-Time Addition and Simultaneous Format Conversion Based on Redundant Binary Representations. *IEEE Transactions on Computers*, 50(11):1267–1278.

Potkonjak, M., Srivastava, M. B., and Chandrakasan, A. P. (1996). Multiple Constant Multiplications: Efficient and Versatile Framework and Algorithms for Exploring Common Subexpression Elimination. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(2):151–165.

Rizzoli, A., Oliverio, F., Montemanni, R., and Gambardella, L. (2004). Ant Colony Optimization for Real-World Vehicle Routing Problems: From Theory to Applications. *Dalle Molle Institute for Artificial Intelligence Research, IDSIA*, pages 1–50.

Roberts, F. S. and Tesman, B. (2009). *Applied Combinatorics*. Taylor and Francis group, 2 edition.

Samadi, P. and Ahmadi, M. (2007). Common Subexpression Elimination for Digital Filters Using Genetic Algorithm. In *Proceedings of the IEEE International Conference on Electronics, Circuits, and Systems*, pages 246–249.

Samueli, H. (1989). An Improved Search Algorithm for the Design of Multiplierless FIR Filters with Powers-of-Two Coefficients. *IEEE Transactions on Circuits and Systems*, 36(7):1044–1047.

Sarwar, A. (1997). Cmos power consumption and c_pd calculation. Technical Report SCAA035B, Texas Instruments.

ScienceDaily (2009). http://www.sciencedaily.com/.

Shi, D. and Yu, Y. J. (2010). Low-Complexity Linear Phase FIR Filters in Cascade Form. *2010 IEEE International Symposium on Circuits and Systems. ISCAS 2010*, pages 177–180.

Shi, D. and Yu, Y. J. (2011). Design of Linear Phase FIR Filters With High Probability of Achieving Minimum Number of Adderse. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 58(1):126–136.

Siek, J., Lee, L.-Q., and Lumsdaine, A. (2002). *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 1 edition.

Silvia, M. and Irene, L. (2004). An Ant Colony Algorithm for the Capacitated Vehicle Routing. *Electronic Notes in Discrete Mathematics*, 18:181–186.

SKA (2013). http://www.skatelescope.org/. The Sequare Kilometer Array.

Smitha, K. and Vinod, A. (2007). A New Binary Common Subexpression Elimination Method for Implementing Low Complexity FIR Filters. In *Proceedings - IEEE International Symposium on Circuits and Systems*, pages 2327–2330.

Steven, S. M. (1997). *Advanced Compiler Design and Implementation.* A Harcourt Science and Technology Company, Harcourt Place, 32 Jamestown Road, London, NWI 7BY, United Kingdom.

Stüetzle, T. and Hoos, H. (1997). The Max-Min Ant System and Local Search for Combinatorial Optimization Problems. *2nd International conference on metaheuritics - MIC 97.*

Takahashi, Y., Sekine, T., and Yokoyama, M. (2008). A Comparison of Multiplierless Multiple Constant Multiplication using Common Subexpression Elimination Method. *2008 IEEE International 51st Midwest Symposium on Circuits and Systems, MWSCAS*, pages 298–301.

Takahashi, Y., Takahashi, K., and Yokoyama, M. (2004). Synthesis of Multiplierless FIR Filter by Efficient Sharing of Horizontal and Vertical Common Subexpression Elimination. *The 2004 International Technical Conference on Circuits/Systems and Communications (ITC-CSCC2004)*, pages (7C2L–4–1)–(7C2L–4–4).

Thong, J. and Nicolici, N. (2009). Time-Efficient Single Constant Multiplication Based on Overlapping Digit Patterns . *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(9):1353–1357.

Thong, J. and Nicolici, N. (2011). An Optimal and Practical Approach to Single Constant Multiplication. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(9):1373–1386.

Tsao, Y.-C. and Choi, K. (2010). A Simplified Flow for Synthesizing Digital FIR Filters Based on Common Subexpression Elimination. *2010 International SoC Design Conference*, pages 174–177.

Vasudev, C. (2006). *Graph Theory with Applications.* New Age International (P) Ltd., 1 edition.

Vinod, A., Lia, E., Maskell, D. L., and Meher, P. (2010). An Improved Common Subexpression Elimination Method for Reducing Logic Operators in FIR Filter Implementations Without Increasing Logic Depth. *Integration, the VLSI journal*, 43(1):124–135.

Vinod, A., Lia, E.-K., and Premkumar, A.B.and Lau, C. (2003). FIR Filter Implementation by Efficient Sharing of Horizontal and Vertical Common Subexpressions. *Electronics Letters*, 39(2):251–253.

Vinod, A. and Lia, E. M.-K. (2005). On the Implementation of Efficient Channel Filters for Wideband Receivers by Optimizing Common Subexpression Elimination Methods. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(2):295–304.

Voronenko, Y. and Püschel, M. (2007). Multiplierless Multiple Constant Multiplication. *ACM Transactions on Algorithms*, 3(2).

Wai-K, i. and Michael, N. (2006). *Markov Chains: Models, Algorithms and Applications.* Springer Science and Business Media, Inc, United States of America.

Walukiewicz, S. (1991). *Integer Programming.* Kluwer Academic Publishers, 1 edition.

Wang, C.-X., Haider, F., Gao, X., You, X.-H., Yang, Y., Yuan, D., Aggoune, H., Haas, H., Fletcher, S., and Hepsaydir, E. (2014). Cellular architecture and key technologies for 5G wireless communication networks. *Communications Magazine, IEEE*, 52(2):122–130.

Wang, Y. and Roy, K. (2005). CSDC: A New Complexity Reduction Technique for Multiplierless Implementation of Digital FIR Filters. *IEEE Transactions on Circuits and Systems*, 52(9):1845–1853.

Yao, C.-Y., Chen, H.-H., Lin, T.-F., Chien, C.-J., and Hsu, C.-T. (2004). A Novel Common Subexpression Elimination Method for Synthesizing Fixed-Point FIR Filters. *IEEE Transactions on Circuits and Systems-I: Regular Papers*, 51(11):2215–2221.

Yao, C.-Y. and Sha, C.-L. (2010). Fixed-Point FIR Filter Design and Implementation in the Expanding Subexpression Space. *2010 IEEE International Symposium on Circuits and Systems. ISCAS 2010*, pages 185–188.

Yu, B., Yang, Z.-Z., and Xie, J.-X. (2011). A Parallel Improved Ant Colony Optimization for Multi-Depot Vehicle Routing Problem. *Journal of the Operational Research Society*, 62(1):183–188.

Yu, L., Li, M., Yang, Y., and Yao, B. (2008a). An Improved Ant Colony Optimization for Vehicle Routing Problem. In *Proceedings of the 8th International Conference of Chinese Logistics and Transportation Professionals - Logistics: The Emerging Frontiers of Transportation and Development in China*, pages 3360–3366.

Yu, Y. J. and Lim, Y. C. (2009). Optimization of FIR Filters in Subexpression Space with Constrained Adder Depth. In *ISPA 2009 - Proceedings of the 6th International Symposium on Image and Signal Processing and Analysis*, pages 766–769.

Yu, Y. J., Shi, D., and Lim, Y. C. (2008b). Subexpression Encoded Extrapolated Impulse Response FIR Filter with Perfect Residual Compensation. *2008 IEEE International Symposium on Circuits and Systemss. ISCAS 2008*, pages 2446–2449.