



T.C.

SELÇUK ÜNİVERSİTESİ

FEN BİLİMLERİ ENSTİTÜSÜ

LOJİK FONKSİYONLARDA İZOLE MİNTERMLERİN TESPİTİ VE ETKİN BİR SADELEŞTİRME ALGORİTMASININ GELİŞTİRİLMESİ

Hakan AKAR

DOKTORA TEZİ

Bilgisayar Mühendisliği Anabilim Dalı

Temmuz-2020
KONYA
Her Hakkı Saklıdır

TEZ KABUL VE ONAYI

Hakan AKAR tarafından hazırlanan "***LOJİK FONKSİYONLARDA İZOLE MİNTERMLERİN TESPİTİ VE ETKİN BİR SADELEŞTİRME ALGORİTMASININ GELİŞTİRİLMESİ***" adlı tez çalışması 01/07/2020 tarihinde aşağıdaki jüri tarafından oy birliği / ~~oy çokluğu~~ ile Selçuk Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı'nda DOKTORA TEZİ olarak kabul edilmiştir.

Jüri Üyeleri	**İmza**
Başkan Doç. Dr. Mesut GÜNDÜZ	……..
Danışman Prof. Dr. Fatih BAŞÇİFTÇİ	……..
Üye Prof. Dr. Hakan IŞIK	……..
Üye Prof. Dr. Harun UĞUZ	……..
Üye Doç. Dr. Halife KODAZ	……..

Yukarıdaki sonucu onaylarım.

Prof. Dr. Saadettin Erhan Kesen
Enstitü Müdür V.

Bu tez çalışması TÜBİTAK tarafından 1059B141500323 nolu proje ile desteklenmiştir.

ÖZET

DOKTORA TEZİ

LOJİK FONKSİYONLARDA İZOLE MİNTERMLERİN TESPİTİ VE ETKİN BİR SADELEŞTİRME ALGORİTMASININ GELİŞTİRİLMESİ

Hakan AKAR

Selçuk Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı

Danışman: Prof. Dr. Fatih BAŞÇİFTÇİ

2020, 167 Sayfa

Jüri
Danışman: Prof. Dr. Fatih BAŞÇİFTÇİ
Prof. Dr. Hakan IŞIK
Prof. Dr. Harun UĞUZ
Doç. Dr. Mesut GÜNDÜZ
Doç. Dr. Halife KODAZ

Bilişim sektörünün hemen hemen bütün alanlarında kullanılan entegre devrelerin daha küçük, daha sade ve daha hızlı olabilmesi için lojik fonksiyonların sadeleştirilmesi büyük önem taşımaktadır. Lojik fonksiyonların sadeleştirilmesi için çok farklı teknikler ve algoritmalar geliştirilmiştir. Bu tez çalışmasında bir çeşit doğrudan örtme tekniği sunulmuştur. Ayrıca lojik fonksiyon dosyalarının içindeki izole mintermler tespit edilerek sadeleştirme işleminde iyileştirme sağlanmıştır. Sunulan sadeleştirme tekniklerinin algoritmaları hazırlanmış, çok işlemcili bilgisayarlara uyumlu parallel versiyonu geliştirilmiş ve C# programında kodlanmıştır.

Bu çalışmada; lojik fonksiyon dosyalarının ne kadar sadeleştirilebileceği, izole mintermlerin tespitinin lojik fonksiyonların sadeleştirilmesini ne kadar etkilediği, fonksiyon sadeleştirme ve izole minterm tespiti işleminin ne kadar paralel hale getirilebileceği araştırılmıştır. Araştırma sonuçlarına göre yakın sonuç kapsama algoritmasının (YSKA) sadeleştirme oranı %82,86 iken, kesin sonuç kapsama algoritmasında (KSKA) bu oran %0,34 artarak %83,20 olmuştur. YSKA ve KSKA 41 benchmarkta eşit sayıda Asal İmplikant (AI) sonucuna ulaşmış, 9 benchmarkta ise KSKA daha iyi sonuç bulmuştur. Fakat YSKA, KSKA'na göre 6,92 kat daha hızlı çalışmıştır. Araştırma sonucunda giriş değişkeni ve ON mintermi sayısının fazla olmadığı benchmarklarda YSKA ve KSKA aynı ya da çok yakın sadeleştirme yaptıkları tespit edilmiştir. Yüksek sayıda ON ve/veya OFF mintermi içeren benchmarklarda KSKA yüksek çalışma süresine rağmen daha sade sonuçlara ulaşmıştır.

Bu tez çalışmasında, izole mintermlerin tespitinin hem YSKA hemde KSKA'nın sonuç kalitelerini arttırdığı ve genel ortalamada çalışma zamanlarını düşürdüğü tespit edilmiştir. İzole mintermlerin tespiti algoritması YSKA üzerinde %2,33 sonuç kalitesi artışı, %12,68 çalışma zamanı kazanımı sağlamıştır. İzole mintermlerin tespiti algoritması KSKA üzerinde %1,09 sonuç kalitesi artışı, %6,51 çalışma zamanı kazanımı sağlamıştır. İzole mintermleri tespit edip sıralamak için geliştirilen algoritma sıralama aşamasında işlem zamanı kullanırken, YSKA ve KSKA'nın işini kolaylaştırarak sadeleştirme aşamasında zaman kazandırmaktadır.

Algoritmaların paralel programlamaya uyarlanması, izole mintermlerin tespiti ve YSKA'nda önemli iyileştirmeler (%49,37, %22,88) sunarken, KSKA'nda belirgin bir farklılık (%1,18, %0,65) oluşturmamıştır. Paralel programlamanın algoritmaların sonuç kalitesi, yani bulduğu AI sayısı üzerinde herhangi bir etkisi olmamıştır.

Anahtar Kelimeler: Fonksiyon Sadeleştirme, İzole Minterm, Lojik Fonksiyon, Minterm, Paralel Programlama.

ABSTRACT

Ph. D. THESIS

FINDING ISOLATED MINTERMS IN LOGIC FUNCTIONS AND DEVELOPING AN EFFICIENT SIMPLIFICATION ALGORITHM

Hakan AKAR

Selçuk University
Institute of Natural Sciences
Department of Computer Engineering

Advisor: Prof. Dr. Fatih BAŞÇİFTÇİ

2020, 167 Pages

Jury

Advisor: **Prof. Dr. Fatih BAŞÇİFTÇİ**
Prof. Dr. Hakan IŞIK
Prof. Dr. Harun UĞUZ
Assoc. Prof. Dr. Mesut GÜNDÜZ
Assoc. Prof. Dr. Halife KODAZ

Minimization of logic functions is very important for the development of simpler, smaller and faster integrated circuits which are used in almost every area of IT sector. Many techniques and algorithms have been developed to simplify logic functions. In this study, a new version of direct cover technique is presented. In addition, minimization process is improved by finding isolated minterms in logic function files. Algorithms for presented minimization technique are prepared and parallel computing algorithms compatible with multicore computers are developed and all these algorithms are coded in Microsoft C# program.

The following questions have been answered in this study; how much minimization is possible for logic function files, how the process of finding isolated minterms affect function minimization, what is the potential speedup in parallelization of function minimization and isolated minterm detection algorithms. Results revealed that minimization ratio of Close Results Covering Algorithm (Yakın Sonuç Kapsama Algoritması, YSKA) (82.86%) are increased by 0.34% in Exact Results Covering Algorithm (Kesin Sonuç Kapsama Algoritması, KSKA) algorithm (83.20%). Both YSKA and KSKA finds equal number of PIs in 41 benchmarks and KSKA finds better results in 9 benchmarks. However, YSKA computes PIs 6.92 times faster. It was also found that YSKA and KSKA perform the same or very close minimization results in benchmarks where input variable and ON minterms are not high. On benchmarks with a high number of ON and/or OFF minterms, KSKA achieved better results despite its high uptime.

In this thesis, it has been found that the detection of isolated minterms improves the quality of the results of both YSKA and KSKA and decreases the average computing time. The detection algorithm of isolated minterms resulted in 2.33% quality increase and 12.68% faster computing time on YSKA. The detection algorithm of isolated minterms resulted in 1.09% quality increase and 6.51% faster computing time on KSKA. While the algorithm developed for detecting and sorting the isolated minterm uses processing time in the sorting phase, it saves time in minimization phase by facilitating the work of the YSKA and KSKA.

While parallelization of algorithms has provided significant improvements in detection of isolated minterms and YSKA (49.37%, 22.88%), there was no significant improvement (1.18%, 0.65%) in KSKA algorithm. Parallel programming had no effect on the result quality of the algorithms like the number of prime implicants.

Keywords: Function Minimization, Isolated Minterm, Logic Functions, Minterm, Parallel Programming

İÇİNDEKİLER

ÖZET ..... **iv**

ABSTRACT ..... **v**

ÖNSÖZ ..... **vi**

İÇİNDEKİLER ..... **vii**

ŞEKİL LİSTESİ ..... **x**

ÇİZELGE LİSTESİ ..... **xii**

SİMGELER VE KISALTMALAR ..... **xiv**

1. GİRİŞ **1**

1.1. Entegre Devreler 2
1.2. Lojik Fonksiyonlar 2
1.2.1. Lojik fonksiyonların gösterimleri 3
1.3. Lojik Fonksiyonların Sadeleştirilmesi 9
1.3.1. Geleneksel sadeleştirme teknikleri 9
1.3.1.1. Matematik işlemleriyle sadeleştirme 9
1.3.1.2. Karnaugh haritası 10
1.3.2. Modern sadeleştirme teknikleri 12
1.3.2.1. Quine McCluskey algoritması 12
1.3.2.2. Espresso algoritması 15
1.3.2.3. İkili karar diyagramları 17
1.4. İzole Mintermler 20
1.4.1. İzole mintermin bulunması 22
1.5. Paralel Programlama 24
1.6. Tezin Amacı ve Önemi 25
1.7. Tezin Organizasyonu 27

2. KAYNAK ARAŞTIRMASI **29**

2.1. Fonksiyon Sadeleştirme Yöntemleri Kaynak Araştırması 29
2.2. Fonksiyon Sadeleştirme Uygulamaları Kaynak Araştırması 41

3. MATERYAL VE YÖNTEM **49**

3.1. Lojik Fonksiyonlar 49
3.1.1. Eksik tanımlanmış fonksiyonlar (Incompletely specified functions) 49
3.2. Bitsel Operatörler 49
3.3. Doğrudan Örtme Algoritmaları 52
3.3.1. Minterm seçme kıstasları 52
3.3.2. İmplikant seçme kriterleri 54
3.4. Petrick Yöntemi 54
3.5. Paralel Algoritmalar 56

3.6. Rekürsif Algoritmalar 57
3.7. Programlama Dili Seçimi (C#) 58

4. LOJİK FONKSİYONLARIN BİTSEL SADELEŞTİRİLMESİ 60

4.1. İzole Mintermlerin Tespiti 60
4.2. Bitsel Operatör İşlemleri 62
4.2.1. Mintermlerin gösterimi 62
4.2.2. Mintermlerin bitsel operatörler ile genişletilmesi 63
4.2.3. Birbirini kapsayan mintermlerin bitsel operatörler ile tespiti 65
4.2.4. Mintermlerin bitsel operatörler ile birbirinden çıkarılması 66
4.2.5. Fonksiyon kapsama algoritmalarında kullanılan tanımlar 67
4.3. Yakın Sonuç Kapsama Algoritması 69
4.4. Kesin Sonuç Kapsama Algoritması 73
4.5. Algoritmaların Paralel Programlamaya Uyarlanması 86
4.6. Fonksiyon Sadeleştirme Uygulaması 89
4.6.1. Kullanılan dosya formatı 89
4.6.2. Kullanıcı arayüzü 90

5. GELİŞTİRİLEN ALGORİTMALARIN ANALİZİ 95

5.1. Boolean Fonksiyonların Sağlanabilirliği ve Geçerliliği 95
5.2. Problemlerin Karmaşıklık Sınıfları 96
5.3. Algoritmaların Karmaşıklık Analizi 97
5.3.1. Algoritmaların çalışma zamanı analizi 97
5.3.2. Asimptotik gösterimler 98
5.3.3. Doğrudan örtme algoritmalarının karmaşıklığı 101
5.3.4. YSKA ve KSKA algoritmalarının karmaşıklığı 102
5.3.5. Espresso algoritmasının karmaşıklığı ve KSKA ile karşılaştırılması 107

6. UYGULAMA SONUÇLARI VE TARTIŞMA 111

6.1. Lojik Fonksiyonların Sadeleştirilmesi 111
6.1.1. YSKA ve KSKA'nın karşılaştırılması 111
6.2. İzole Mintermlerin Tespiti 114
6.2.1. İzole mintermlerin tespitinin YSKA'ya etkisi 114
6.2.2. İzole mintermlerin tespitinin KSKA'ya etkisi 117
6.3. Algoritmaların Paralel Programlamaya Uyarlanması 119
6.3.1. Yakın sonuç kapsama algoritmasının paralel programlanması 119
6.3.2. Kesin sonuç kapsama algoritmasının paralel programlanması 121
6.3.3. İzole minterm algoritmasının paralel programlanması 121
6.4. Çok Büyük Dosyalarda Sadeleştirme İşlemleri 124
6.5. Geliştirilen Programın Bellek Kullanımı 126
6.6. Tartışma 127
6.6.1. Yakın ve kesin sonuç kapsama algoritmalarının karşılaştırılması 128
6.6.2. İzole mintermlerin tespitinin sadeleştirmeye etkisi 129
6.6.3. Paralel programlamanın sadeleştirmeye etkisi 130

7. SONUÇLAR VE ÖNERİLER 132

7.1. Sonuçlar 132

7.2. Öneriler 134
KAYNAKLAR 136
ÖZGEÇMİŞ 148

ŞEKİL LİSTESİ

Şekil 1.1: 3'lü Boolean Fonksiyonu ve 3 Boyutlu Küp Gösterimi …… 3
Şekil 1.2: 1, 2, 3 ve 4 Boyutlu Boolean Fonksiyon Uzayı …… 11
Şekil 1.3: 1, 2, 3 ve 4 Değişkenli Boolean Fonksiyonu İçin KH'ler ..……..11
Şekil 1.4: ESPRESSO Algoritması Kaba Kodu …… 16
Şekil 1.5: Çoğunluk Fonksiyonunun İkili Ağaç Gösterimi ...…… 18
Şekil 1.6: Boolean Fonksiyonu İkili Ağaçtan ROBDD'ye İndirgeme ……..20
Şekil 1.7: En İyi ve En Kötü SOP İfadeler ...……21
Şekil 1.8: Pomper ve Armstrong Kaba Kodu (Pseudo Code) ...…… 22
Şekil 1.9: Besslich Algoritması Kaba Kodu (Pseudo Code) ...……..23
Şekil 1.10: Dueck ve Miller Minterm Seçme Kaba Kodu (Pseudo Code) ...…… 23
Şekil 1.11: Dueck ve Miller AI Seçme Kaba Kodu (Pseudo Code) ...…….. 24
Şekil 3.1: Paralel algoritma tasarım şeması …….. 56
Şekil 3.2: Rekürsif algoritma çalışma prensibi …… 57
Şekil 3.3: C# Projesi çalışma prensibi ……..58
Şekil 4.1: İzole mintermi gösteren karnaugh haritası ……....60
Şekil 4.2: İzole mintermlerin tespiti algoritması kaba kodu (pseudo code) …… 61
Şekil 4.3: Yakın sonuç kapsama algoritması kaba kodu ...……..... 69
Şekil 4.4: Kesin sonuç kapsama algoritması kaba kodu ...…… 73
Şekil 4.5: Kesin sonuç kapsama algoritması akış diyagramı ...……. 74
Şekil 4.6: Kayıt kontrol algoritması kaba kodu ...……..... 75
Şekil 4.7: Kayıt kontrol algoritması akış diyagramı ...…….. 76
Şekil 4.8: Amdahl Kanunu …… 86
Şekil 4.9: İzole mintermlerin paralel programlama ile tespiti……88
Şekil 4.10: Fonksiyon sadeleştirmede kullanılan girdi dosyası ...…… 90
Şekil 4.11: Fonksiyon Sadeleştirme Programı Kullanıcı Arayüzü ……...……..... 91
Şekil 4.12: Seri programlama algoritmaları kullanıcı arayüzü ……..... 92
Şekil 4.13: Paralel programlama algoritmaları kullanıcı arayüzü ……93
Şekil 5.1: Problemlerin karmaşıklık sınıfları arasındaki ilişki ……...…….....96
Şekil 5.2: Algoritmaların en iyi, ortalama ve en kötü durum grafiği …….. 98
Şekil 5.3: Θ, O ve Ω notasyonlarının grafik gösterimleri ...……...…… 100
Şekil 5.4: Yakın sonuç kapsama algoritması kaba kodu ……...……..... 102
Şekil 5.5: Kesin sonuç kapsama algoritması kaba kodu ……. 104

Şekil 5.6: İzole mintermlerin tespiti algoritması kaba kodu ……... 105
Şekil 5.7: Espresso algoritması kaba kodu ……. 108
Şekil 6.1: İzole mintermlerin YSKA'ya etkisi …… 115
Şekil 6.2: İzole mintermlerin KSKA'ya etkisi …….……... 117

ÇİZELGE LİSTESİ

Çizelge 1.1: Lojik Fonksiyon Sembolleri ve Anlamları ….…….. 4
Çizelge 1.2: 3 Değişkenli Çoğunluk Fonksiyonunun Doğruluk Tablosu …….. 7
Çizelge 1.3: Boolean Cebri Sadeleştirme Kuralları ……..... 10
Çizelge 1.4: Fonksiyon F (A, B, C, D) İçin Doğruluk Tablosu ……. 13
Çizelge 1.5: AI'ların Bulunması …...……14
Çizelge 1.6: Fonksiyon Kapsama Probleminin Çözümü ……..……14
Çizelge 3.1: Bitsel operatörlerin gösterimi ……....50
Çizelge 3.2: Algoritmaların minterm ve implikant seçme kriterleri ……. 54
Çizelge 3.3: Petrick yöntemi uygulama tablosu …….…... 55
Çizelge 4.1: Örnek mintermin izole seviyesinin hesaplanması ……62
Çizelge 4.2: Fonksiyon değerlerinin bitsel temsili …… 62
Çizelge 4.3: Bir mintermin bitsel gösterimi …….. 63
Çizelge 4.4: Bitsel operatörlerin gösterimi ……63
Çizelge 4.5: X_i'sol ve sağ bit gösterimi ……..... 64
Çizelge 4.6: X_i'nin genişletilmesiyle oluşan Z kümesi ……65
Çizelge 4.7: x_i ve y_i literallerine göre z_i değeri ……67
Çizelge 4.8: "0000" minterminin genişletilmesi ve ayıklanması …….. 70
Çizelge 4.9: "0101" minterminin genişletilmesi ve ayıklanması …….. 71
Çizelge 4.10: "1010" minterminin genişletilmesi ve ayıklanması …… 72
Çizelge 4.11: "0000" minterminin genişletilmesi ve ayıklanması …… 77
Çizelge 4.12: "0001" minterminin genişletilmesi ve ayıklanması …… 78
Çizelge 4.13: "0010" minterminin genişletilmesi ve ayıklanması …… 79
Çizelge 4.14: "0100" minterminin genişletilmesi ve ayıklanması …… 80
Çizelge 4.15: "0101" minterminin genişletilmesi ve ayıklanması …… 81
Çizelge 4.16: "0111" minterminin genişletilmesi ve ayıklanması …… 82
Çizelge 4.17: "1001" minterminin genişletilmesi ve ayıklanması …… 84
Çizelge 5.1: Geliştirilen algoritmaların karmaşıklık analizleri ……... 106
Çizelge 6.1: Yakın Sonuç ve Kesin Sonuç Kapsama Algoritmaları ……...113
Çizelge 6.2: İzole Mintermlerin Tespitinin YSKA'ya Etkisi ……. 116
Çizelge 6.3: İzole Mintermlerin Tespitinin KSKA'ya Etkisi …...…….. 118
Çizelge 6.4: Paralel Programlamanın YSKA'ya Etkisi …….. 120
Çizelge 6.5: Paralel Programlamanın KSKA'ya Etkisi …….. 122

Çizelge 6.6: Paralel Programlamanın İzole Sıralı Algoritmalara Etkisi ……. 123
Çizelge 6.7: Çok Büyük benchmarklarda İzole Sıralamanın Etkisi ……125
Çizelge 6.8: Çok Büyük benchmarklarda Paralel Programlamanın Etkisi ……..... 126
Çizelge 6.9: Çok Büyük benchmarklarda Bellek Kullanımı ……...127

SİMGELER VE KISALTMALAR

Simgeler

{0,1,*}	Boolean değişkenin tanımlama uzayı
*	Belirli olmayan değişken değeri
n	fonksiyonun değişken sayısı
$\mathbb{D}$	Boolean mintermi
$\mathbb{D}^n$	n boyutlu Boolean mintermi
α	minterm
f^{on}	Doğru sonuç veren mintermler kümesi
f^{off}	Yanlış sonuç veren mintermler kümesi
f^{dc}	DC mintermler kümesi
$x_1, x_2, \ldots, x_n$	Değişken vektörü
β	Örnek Boolean formülü
$\varphi_1, \varphi_2, \ldots, \varphi_n$	Örnek mintermler
∧	Kesişim (*VE*)
∨	Birleşim (*VEYA*)
∀	Evrensel tanımlayıcı, "Bütün…" niceleyicisi
∃	Varlıksal tanımlayıcı, "En az 1 …" niceleyicisi
$f\ (a, b, c)$	Üç değişkenli fonksiyon
¬	Tamamlayıcı, tümleyen, "*değil*" işareti
γ	Yaprak düğüm
$deger(\gamma)$	Yaprak düğümün değeri
$indeks(\gamma)$	Yaprak düğümün indeksi
$yok(\gamma)$	Yaprak düğümün 0 çocuğu
$var(\gamma)$	Yaprak düğümün 1 çocuğu

$.i$	PLA giriş değişkeni sayısı
$.o$	PLA çıkış değişkeni sayısı
$.p$	PLA minterm sayısı
$.e$	PLA dosya sonu
$\mathbb{D}_L$	$\mathbb{D}$ minterminin sol biti
$\mathbb{D}_R$	$\mathbb{D}$ minterminin sağ biti
$\oplus$	Özel *VEYA* (XOR)
$\otimes$	Minterm genişletme simgesi
$\textcircled{\scriptsize ⅡⅡ}$	Minterm kapsama simgesi
$\ominus$	Minterm çıkarma simgesi
$\mathcal{X}$	ON mintermler kümesi
$\mathcal{Y}$	OFF mintermler kümesi
$\Longrightarrow$	eğer öyle ise bağlacı
$\Leftrightarrow$	Ancak ve ancak bağlacı
$\mid$	Ya da bağlacı
O	Büyük Omicron Notasyonu
Ω	Büyük Omega Notasyonu
Θ	Büyük Teta Notasyonu

Kısaltmalar

AI	Asal İmplikant (Prime Implicant)
ASICs	Application Specific Integrated Circuits (Uygulamaya Özel Entegre Devreler)
BDD	Binary Decision Diagram (İkili Karar Diyagramı)
BEK	Bekleyen Mintermler Kümesi
CF	Clustering Factor (Kümeleme Faktörü)
CMOS	Complementary Metal Oxide Semiconductor Transistor (Tamamlayıcı Metal Oksit Yarıiletken Transistor)
CNF	Conjunctive Normal Form (Bağlayıcı Normal Form)
DC	Don't Care (Önemsiz)
DF	Discernibility Function (Fark edilebilirlik Fonksiyonu)
DNF	Disjunctive Normal Form (Toplayıcı Normal Form)
ESOP	EXOR Sum of Products (Çarpımların Özel-Veya-Toplamı)
ESCT	EXOR Sum of Complex Terms (Karmaşık Terimlerin Özel-Veya-Toplamı)
EXOR	Exclusive OR (Özel Veya)
eQMC	Enhanced QMC (Geliştirilmiş QMC)
FIB	Full Forwarding Information Base (Tam Yönlendirme Bilgi Tabanı)
IM	İşlenen Minterm
Imp	İmplikant
IW	Isolation Weight (İzolasyon Ağırlığı)
JC	Joint Computation (Birleşik Hesaplama)
JSD	Joint Support Decomposition (Ortak Destek Ayrışması)
K	Kayıt Durumu

KH	Karnough Haritası (Karnough Map)
KSKA	Kesin Sonuç Kapsama Algoritması
LSI	Large Scale Integrated (Büyük Ölçekli Devre)
MLC	Machine Learning Classifiers (Makine Öğrenme Sınıflandırıcıları)
MP	Multiprocessor (Çok İşlemcili)
NMOS	N Channel Mosfet (N Kanallı Mosfet)
NP	Non-deterministic Polynomial (Belirsiz Polinomsal)
NP-com	Nondeterministic Polynomial Complete (Belirsiz Polinomsal Tam)
NP-hard	Nondeterministic Polynomial Hard (Belirsiz Polinomsal Zor)
NRC	Neighbourhood Relative Count (Komşuluk Göreli Sayım)
OBDD	Ordered BDD (Sıralı BDD)
ON_Min	ON Kümesi Mintermleri
P	Polynomial (Polinomsal)
PLA	Programmable Lojik Arrays (Programlanabilir Lojik Diziler)
PMOS	P Channel Mosfet (P Kanallı Mosfet)
POS	Product of Sum (Toplam Terimlerinin Çarpımı)
QCA	Qualitative Comparative Analysis (Nitel Karşılaştırma Analizi)
QMC	Quine-McCluskey (Bir Çeşit Tablolama Algoritması)
QMDD	Quantum Multiple-Valued Decision Diagrams (Kuantum Çok Değerli Karar Diyagramları)
QOF	Quantum Operator Form (Kuantum Operatör Formu)
RBC	Relative Break Count (Göreceli Kesme Sayısı)
RTL	Register Transfer Level (Yazmaç Transfer Seviyesi)
SAT	Satisfiability Problem (Sağlanabilirlik Problemi)
SF	Switching Functions (Anahtarlama Fonksiyonu)

SOP	Sum of Product (Çarpım Terimlerinin Toplamı)
TLC	Threshold Logic Circuits (Eşik Mantık Devreleri)
VLSI	Very Large Scale Integrated (Çok Büyük Ölçekli Devre)
YSKA	Yakın Sonuç Kapsama Algoritması

1. GİRİŞ

Klasik (iki değerli) mantık her önermenin doğru ya da yanlış olduğunu varsaymaktadır. Bazı durumlarda önermelerin belirsiz olabileceği durumu Aristo'nun "Yorum üzerine" eserinden beri tartışılmaktadır. Klasik mantıkta önermelerin doğru, yanlış ya da belirsiz olması durumu, dijital elektronikte elektronların akış yönlerini belirlemekte de kullanılmaktadır.

Dijital elektroniğin temeli Lojik ifadelerdir. Dijital elektronik devrelerinde bulunan transistörler iletim ya da kesim durumunda bulunurlar (Agarwal ve Lang, 2009). Bu yüzden devrenin çıkışında görülen seviye ya kaynak gerilimi ya da toprak gerilimi seviyesindedir. Dijital elektronikte transistörlerin iletim durumunda olduğu durum yani kaynak gerilimi "Lojik 1" olarak ifade edilirken, transistörlerin kesim durumunda olduğu durum yani toprak gerilimi "Lojik 0" olarak ifade edilir. Teknolojinin gelişmesiyle çok büyük ölçekli entegre (Very Large Scale Integrated, VLSI) devrelerin sahip olduğu transistor sayısı Moore kuralına (Moore, 1965) göre artmasına rağmen kullanılacak alanların sabit ya da daha az olması sebebiyle kullanılan transistör sayısının azalması büyük önem taşır. Dijital devrelerde kullanılan transistörlerin iletim ya da kesim durumunda olması lojik ifadelerle tanımlanmaktadır. Dijital devre çıkışında kaynak gerilimlerini veren devre giriş değerleri lojik fonksiyonlarla ifade edilmektedir.

Lojik ifadeler tek değerli (0-1) ya da çok değerli olabilmektedir. Her iki Lojik ifadelerde günümüzde yaygın olarak kullanılmaktadır. Bilgisayar teknolojisi geliştikçe, ihtiyaç duyulan Lojik ifadeler daha karmaşık hale gelmektedir. Bu durum harcanan zamanın, kullanılan belleğin ve gerçekleştirme maliyetinin artması anlamına gelir. Lojik ifadelerin karmaşıklığı arttıkça, ifadelerin sadeleştirilmeleri daha fazla önem arzeder. Teknoloji geliştikçe dijital devrelerin kullanıldığı cihazların daha da küçülmesi, devrelerin sadeleştirilmesini mecburi kılmaktadır. 1920'lerde havası boşaltılmış elektron lambalarıyla kullanılmaya başlanılan elektronik devreler, 1950'lerde transistörlerin, 1960'larda entegrelerin bulunmasıyla daha küçük, işlevsel, hızlı ve doğru çalışır hale gelmiştir (Jenkins, 2005).

Lojik ifadeler toplamların çarpımı ve çarpımların toplamı olmak üzere iki şekilde ifade edilebilmektedir (Crama ve Hammer, 2012). Eğer farklı değerler parantez içerisinde önce toplanıp, ardından parantezler birbirleriyle çarpılıyorsa bu ifade toplamların çarpımı (Product of Sum-POS) şeklinde yazılmıştır. Lojik ifadeler birbirleriyle çarpılıp, daha

sonra bu çarpımlar toplanıyorsa, bu gösterime çarpımların toplamı (Sum of Product-SOP) denir (Altun ve Riedel, 2012).

1.1. Entegre Devreler

Jack Kilby'nin entegre devreleri (Integrated Circuit, IC) 1958 yılında keşfetmesinden beri bu alanda çok fazla gelişmeler olmuştur. Intel'in kurucu ve ortaklarından Gordon Moore, 1965 yılında yarı iletken endüstrisi için önemli bir tahminde bulunmuştur. Daha sonra Moore kanunu olarak da bilinen bu yasaya göre, tek bir entegre devreye koyulabilecek transistor sayısı her iki yılda yaklaşık olarak iki katına çıkacaktır. Yaklaşık 50 yıldır bu tahmin doğru çıkmıştır. Günümüzde birkaç milyar transistör tek bir entegre devreye koyulabilmektedir. Devre tasarımlarının daha karmaşık olmasına sebep olan bu yükseliş, araştırmacıların çözmesi gereken bir problem olarak durmaktadır. Araştırmacılar, lojik sadeleştirme ve elektronik tasarım otomasyonu (Electronic Design Automation, EDA) gibi yöntemlerle daha sade devreler tasarlamaya çalışmışlardır.

Lojik sadeleştirmenin temelinde mantık ve cebir vardır. Boolean Cebri (Boolean Algebra) lojik sadeleştirmenin özüdür. Boolean cebri ile devre tasarımını birleştiren en etkili çalışma Claude E. Shannon'nın "A Symbolic Analysis of Relay and Switching Circuits" isimli araştırmasıdır (Shannon, 1938). Shannon bu çalışmada anahtarlama devrelerinin tasarım ve analizinin boolean cebri ile yapılabileceğini ve anahtarlama devrelerinin boolean cebri problemlerini çözmek için kullanılabileceğini ispatlamıştır.

İki seviyeli SOP formundaki lojik fonksiyonların sadeleştirilmesi teorisi Willard V. Quine tarafından 1952'de sunulmuştur. SOP formundaki fonksiyonların sadeleştirilmesi 1970'lerden beri IC tasarımında geniş uygulama alanı bulmuştur.

1.2. Lojik Fonksiyonlar

Günümüzde kullanılan her elektronik aygıt gibi bilgisayarlarda binary sayı sistemiyle çalışırlar. 0 ve 1'lerden oluşan bu sayı sistemi, dijital elektronik devrelerinin tasarımında kullanılır. Dijital elektronik devre tasarımının en temel öğesi, üzerinde işlem yapmamızı sağlayan, matematikçi ve filozof olan George Boole tarafından bulunan "Boolean sayı sistemi" dir (Boole, 1998). Bu sayı sistemi üzerinde matematiksel işlemler yapılmasını sağlayan bazı kurallar tanımlanmıştır. Dijital elektronikte en çok kullanılan kurallar Augustus De Morgan tarafından geliştirilen "De Morgan kuralları"dır (Morgan,

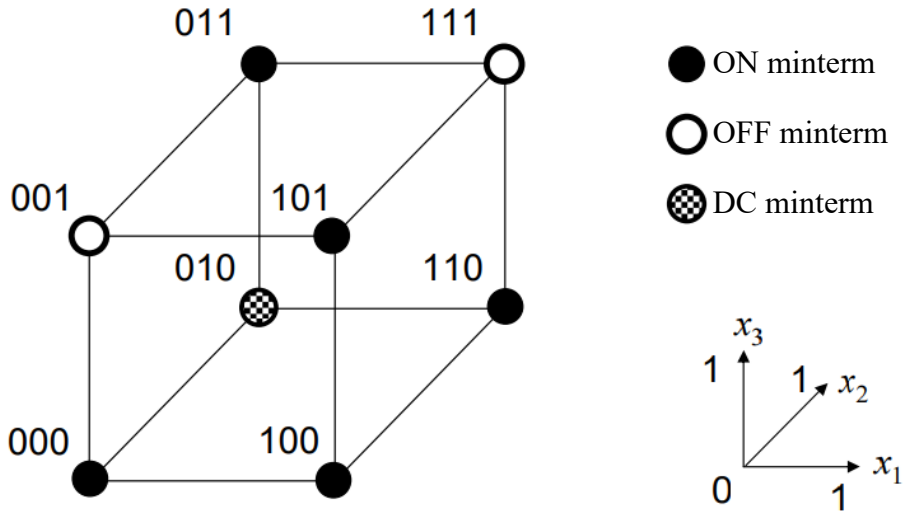
1847). Bu kuralların dijital elektronikte nasıl kullanılabileceğini matematikçi ve elektronikçi Shannon göstermiştir (Shannon, 1938).

1.2.1. Lojik fonksiyonların gösterimleri

Boolean fonksiyonları kümesi $m > 1$ olmak şartıyla, eğer f tam tanımlanmış ise $f: \mathbb{D}^n \rightarrow \mathbb{D}^m$, *eksik* tanımlanmış ise $f: \mathbb{D}^n \rightarrow \mathbb{D}_+^m$, fonksiyonel vektör veya çok çıkışlı f fonksiyonu olarak tanımlanabilir.

Tam tanımlanmış f fonksiyonu için doğru mintermler kümesi $f^{on} = \{ \alpha \in \mathbb{D}^n \mid f(\alpha) = 1 \}$, yanlış mintermler kümesi $f^{off} = \{ \alpha \in \mathbb{D}^n \mid f(\alpha) = 0 \}$ şeklinde tanımlanır. Eksik tanımlanmış f fonksiyonu için doğru ve yanlış mintermler kümesine ek olarak tanımlanmamış *DC* mintermler kümesi $f^{dc} = \{ \alpha \in \mathbb{D}^n \mid f(\alpha) = * \}$ olarak tanımlanır.

Örnek 1.1; Değişken vektörü (x_1, x_2, x_3) tarafından belirtilen bir fonksiyon, 3 boyutlu Boolean fonksiyon küpü şeklinde Şekil 1.1.'de gösterilmiştir. Köşelerdeki noktalar mintermleri göstermektedir ve Hamming uzaklığı (Hamming Distance, HD) bir olan (tek basamak farklılığı olan) mintermler bir kenar ile bağlanmıştır. Bu fonksiyona ait $f^{off} = \{001, 111\}$, $f^{on} = \{000, 011, 100, 101, 110\}$, ve $f^{dc} = \{010\}$ kümeleri, birleşik Boolean küpünde gösterilmiştir.



Şekil 1.1. 3'lü Boolean fonksiyonu ve 3 boyutlu küp gösterimi

Eğer tam belirtilmiş Boolean fonksiyonu f'in *on* mintermler kümesi f^{on}, bütün fonksiyona eşit ise f fonksiyonu $f \equiv 1$ veya $f \Leftrightarrow 1$ şeklinde gösterilir ve gereksizdir. Çünkü f fonksiyonunun bütün değerleri 1 olur.

Bütün Boolean fonksiyonları Boolean formülü olarak ifade edilebilir. Boolean formüllerinde kullanılan semboller Çizelge 1.1.'de gösterilmiştir. Boolean bağlaçları $\vee$, $\wedge$, $\neg$, $\Rightarrow$, ve $\Leftrightarrow$ sembolleridir. Bir Boolean formülü β denklem 1.1'deki kurallara göre yinelemeli olarak tanımlanabilir.

$$\beta ::= 0 \mid 1 \mid A \mid \beta_1 \vee \beta_2 \mid \beta_1 \wedge \beta_2 \mid \neg\beta_1 \mid \beta_1 \Rightarrow \beta_2 \mid \beta_1 \Leftrightarrow \beta_2, \tag{1.1}$$

"::=" işareti "eşit olabilir", "|" işareti ise "veya" demektir. Bir Boolean formülü sabit 0, sabit 1, A değişken kümesinden bir Boolean değişkeni, $\beta_1 \vee \beta_2$, $\beta_1 \wedge \beta_2$, $\neg\beta_1$, $\beta_1 \Rightarrow \beta_2$, *veya* $\beta_1 \Leftrightarrow \beta_2$ koşulunu sağlayan β_1 *ve* β_2 formüllerine eşit olabilir. Parantezleri azaltmak ve okunabilirliği arttırmak için Boolean bağlaçlarının öncelik sırası şöyledir: $\Leftrightarrow$, $\Rightarrow$, $\vee$, $\wedge$, $\neg$. Boolean formüllerinde genellikle kesişim sembolü "$\wedge$" atlanabilir.

Çizelge 1.1. Lojik fonksiyon sembolleri ve anlamları

Sembol	Adı	Anlamı
( )	Sağ ve Sol parantez	İşaretler için öncelik verir
$\neg$, `	Tamamlayıcı	Mantıksal "NOT"
$\wedge$, ·	Kesişim	Mantıksal "AND"
$\vee$, +	Birleşim	Mantıksal "OR"
$\Rightarrow$	Anlatma	Eğer …. ise … olur.
$\Leftrightarrow$, $\equiv$	Denklik	Karşılıklı birbirini gerektirir
$\exists$	Varlıksal tanımlayıcı	Şartları sağlayan bir öğenin bulunduğunu belirtir.
$\forall$	Evrensel tanımlayıcı	Bütün öğelerin şartları sağladığını belirtir.

Örnek 1.2; Denklem 1.2'de verilen bir Boolean Formülü

$$f = ((\varphi_1 \vee (\neg\varphi_2)) \vee ((\neg\varphi_1) \wedge \varphi_3)) \wedge (\varphi_1 \wedge (\neg\varphi_2)) \tag{1.2}$$

kesişim sembolleri atlanırsa denklem 1.3,

$$f = ((\varphi_1 \vee \neg\varphi_2) \vee \neg\varphi_1 \varphi_3)(\varphi_1 \neg\varphi_2) \tag{1.3}$$

bağlaç öncelik sırasına göre parantezler kaldırılırsa denklem 1.4,

$$f = (\varphi_1 \vee \neg\varphi_2 \vee \neg\, \varphi_1\, \varphi_3)\, \varphi_1\, \neg\varphi_2) \tag{1.4}$$

şeklinde sadeleşebilir fakat artık formülün orijinal formu buradan geriye türetilemez.

Bir Boolean fonksiyonu üretebilecek yeterlilikteki Boolean bağlaçlar kümesine işlevsel olarak tam (Functionally Complete, FC) denir. İşlevsel olarak tam bağlaçlar kümesi oluşturmak için yukarıda tanımlanan bütün bağlaçları kullanmak şart değildir. Örneğin, $\{\neg, \wedge\}$ ve $\{\neg, \Rightarrow\}$ kümeleri işlevsel olarak tamdır, fakat $\{\wedge, \Rightarrow\}$ kümesi işlevsel olarak tam değildir.

Bir Boolean fonksiyonunu bazı Boolean formüllerinin semantiği olarak düşünülebilir. Aynı (anlamsal) Boolean fonksiyonunu tanımlayan, farklı (sözdizimsel) Boolean formülleri vardır. Lojik sadeleştirmeyi önemli yapan Boolean fonksiyonlarının bu esnek yapısıdır.

Boolean fonksiyonları üzerindeki Boolean işlemleri, birleşim "$\cup$", kesişim "$\cap$" ve tümleme "$\neg$" gibi küme işlemleri olarak tanımlanabilir. Denklem 1.5'te temel küme işlemlerine göre fonksiyonların alt kümeleri gösterilmiştir.

$$\begin{aligned} &f = p \wedge q \;\Rightarrow\; f^{on} = p^{on} \cap q^{on},\; f^{off} = p^{off} \cup q^{off} \\ &f = p \vee q \;\Rightarrow\; f^{on} = p^{on} \cup q^{on},\; f^{off} = p^{off} \cap q^{off} \\ &f = \neg p = p' \;\Rightarrow\; f^{on} = p^{off},\; f^{off} = p^{on} \end{aligned} \tag{1.5}$$

ON kümesi, OFF kümesi ve *DC* kümesinin bütün kümeye eşit olmasına dayanarak, f fonksiyonunun *DC* kümesi türetilebilir.

Tanımlanmış Boolean formüllerinden (quantified Boolean formulas, QBFs), varlıksal ($\exists$) ve evrensel ($\forall$) tanımlayıcılar yardımıyla tanımlanmamış (quantifier-free) Boolean formülleri oluşturulabilir. QBF yazarken, tanımlayıcıların önceliğinin Boolean bağlaçlarının önceliğinden daha düşük olduğu varsayılır. Bir QBF'de, tanımlanan değişkenlere bağıl değişkenler (bound variables), tanımlanmamış değişkenlere serbest değişkenler (free variables) denir.

Örnek 1.3; QBF $\forall\varphi_1, \exists\varphi_2$ olduğu düşünülürse, Boolean formülü $p(\varphi_1, \varphi_2, \varphi_3)$ şu şekilde okunur: Bütün φ_1'ler için bazı φ_2'ler vardır. Bu durumda φ_1 ve φ_2 bağıl değişken, φ_3 ise serbest değişkendir.

Herhangi bir QBF, formül genişletmeyle tanımlayıcı elemesi yaparak, tanımlanmamış bir Boolean formülü olarak yazılabilir. Örneğin p Boolean formülü;

$$\forall\varphi.\, p(\varphi, \omega) = p(0, \omega) \wedge p(1, \omega) \tag{1.6}$$

$$\exists\varphi.\, p(\varphi, \omega) = p(0, \omega) \vee p(1, \omega) \tag{1.7}$$

şeklinde yazılabilir. Sonuç olarak, herhangi bir QBF φ'ye eşit, φ'nin sadece serbest değişkenlerini belirten bir tanımlanmamış Boolean formülü bulunur. k bağıl değişkeni olan n boyutlu bir QBF için, formül genişletme ile türetilebilecek tanımlanmamış Boolean formülünün büyüklüğü $O\,(2^n.k)$'dır. QBF'lerin tanımlanmamış Boolean formülleriyle aynı ifade gücü vardır fakat çok daha kısa ve sade ifade edilirler.

Örnek 1.4; QBF $\forall\varphi_1, \exists\varphi_2.p(\varphi_1, \varphi_2, \varphi_3)$ denklem 1.8'deki gibi yazılabilir.

$$\begin{aligned}&\forall\varphi_1.(\, p(\varphi_1, 0, \varphi_3) \vee p(\varphi_1, 1, \varphi_3)\,) \\ &= (\, \exists\varphi_2\,.\, p(0, \varphi_2,\ \varphi_3)\,) \wedge (\, \exists\varphi_2\,.\, p(1, \varphi_2,\ \varphi_3)\,) \\ &= (\, p(0, 0, \varphi_3) \vee p(0, 1, \varphi_3)\,) \wedge (\, p(1, 0, \varphi_3) \vee p(1, 1, \varphi_3)).\end{aligned} \tag{1.8}$$

$\forall\varphi_1, \exists\varphi_2.p(\varphi_1, \varphi_2, \varphi_3)$ ifadesi $\exists\varphi_2, \forall\varphi_1.p(\varphi_1, \varphi_2, \varphi_3)$ ifadesinden farklıdır. $(\exists\varphi_2,\ \forall\varphi_1.p(\varphi_1,\ \varphi_2,\ \varphi_3)) \Rightarrow (\forall\varphi_1,\ \exists\varphi_2.p(\varphi_1,\ \varphi_2,\ \varphi_3))$. Karşılaştırıldığında $\forall\varphi_1, \forall\varphi_2.p(\varphi_1, \varphi_2, \varphi_3)$ ifadesi $\forall\varphi_2, \forall\varphi_1.p(\varphi_1, \varphi_2, \varphi_3)$ ifadesine eşittir ve benzer şekilde $\exists\varphi_1, \exists\varphi_2.\, p(\varphi_1, \varphi_2, \varphi_3)$ ifadesi $\exists\varphi_2, \exists\varphi_1.p(\varphi_1, \varphi_2, \varphi_3)$ ifadesine eşittir.

Boolean formüllerinde evrensel tanımlayıcı $\forall$'nın kesişim $\wedge$ işlemine, varlıksal tanımlayıcı $\exists$'nin birleşim $\vee$ işlemine gerek duyar. Böylece herhangi bir QBF β_1 ve β_2 için denklem 1.9 ifadesi,

$$\exists\alpha\,.\,(\,\beta_1 \vee \beta_2\,) = \exists\alpha\,.\,\beta_1 \ \vee\ \exists\alpha\,.\,\beta_2 \tag{1.9}$$

buna karşılık evrensel tamınlayıcı için de denklem 1.10 ifadesi yazılabilir.

$$\forall\alpha\,.\,(\beta_1 \wedge \beta_2) = \forall\alpha\,.\,\beta_1 \wedge \forall\alpha\,.\beta_2 \tag{1.10}$$

Boolean formüllerinde evrensel tanımlayıcı ∀'nın birleşim ⋁ işlemine dağılma özelliği yoktur ve varlıksal tanımlayıcı ∃'nin kesişim ⋀ işlemine dağılma özelliği yoktur. Yani,

$$\forall\alpha\,.\,(\beta_1 \vee \beta_2) \neq \forall\alpha\,.\,\beta_1 \vee \forall\alpha\,.\beta_2 \tag{1.11}$$

$$\exists\alpha\,.\,(\beta_1 \wedge \beta_2) \neq \exists\alpha\,.\,\beta_1 \wedge \exists\alpha\,.\,\beta_2 \tag{1.12}$$

Ancak herhangi bir QBF β için denklem 1.13 ve denklem 1.14 yazılabilir.

$$\neg\forall\alpha.\beta = \exists\alpha.\neg\beta \tag{1.13}$$

$$\neg\exists\alpha.\beta = \forall\alpha.\neg\beta. \tag{1.14}$$

Çünkü evrensel tanımlayıcı ∀ ve varlıksal tanımlayıcı ∃ işaretleri, birinin tersi alınarak diğerine çevrilebilir.

Boolean fonksiyonlarını göstermek için farklı yöntemler vardır. Bir Boolean fonksiyonu, her doğruluk atamasının liste halinde verildiği doğruluk tablosu (truth table, TT) ile kapsamlı olarak gösterilebilir.

Örnek 1.5; Çizelge 1.2. {*a, b, c*} değişkenlerinin çoğunun doğru olduğu değerleri veren çoğunluk fonksiyonuna *f* = {*a, b, c*} ait doğruluk tablosunu göstermektedir.

Çizelge 1.2. 3 değişkenli çoğunluk fonksiyonunun doğruluk tablosu

a	***b***	***c***	***f***
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Doğruluk tabloları Boolean fonksiyonlarının normal (canonical) gösterimleridir. Eğer aynı doğruluk tablolarına sahipse, iki Boolean fonksiyonu birbirine eşittir. Lojik

sadeleştirme ve doğrulama uygulamalarında normal gösterimler (Canonicity) önemli bir özelliktir.

Birkaç giriş değişkenine sahip fonksiyonları göstermek için doğruluk tabloları çok uygundur. Bir doğruluk tablosunu bilgisayar kelimesi olarak saklarken, iki küçük fonksiyon üzerinde temel Boolean işlemleri yapılabilir veya 32 bitlik integer sayılardan faydalanılabilir. Fakat doğruluk tabloları çok girişli fonksiyonları göstermek için çok kullanışsızdır.

Çarpımların toplamı (Sum of Products, SOP), veya bilgisayar bilimlerindeki adıyla "Veya Bağlacı Normal Şekli" (Disjunctive Normal Form, DNF), değişkenlerin çarpımının (çarpım terimleri veya küpler) toplamından oluşan özel bir Boolean formülü şeklidir. SOP formu iki seviyeli devre tasarımının (ilk seviye AND kapısı, ikinci seviye OR kapısı) gösterimidir. İki seviyeli lojik sadeleştirmede, Boolean fonksiyonlarının SOP gösterimindeki çarpım ifadeler kümesine, Boolean fonksiyonunu kapsayan/örten (cover) denir. Bir Boolean fonksiyonunun farklı birçok kapsayanı olabilir ve bir kapsayan bir Boolean fonksiyonunu gösterebilir.

Örnek 1.6; Fonksiyon $f = \neg abc + a\neg bc + ab\neg c + abc$ ifadesi SOP formundadır. $\{\neg abc, a\neg bc, ab\neg c, abc\}$ kümesi f fonksiyonunu kapsar. Araştırmacılar genellikle SOP formundaki fonksiyon ile onu kapsayan kümeyi bir tutarlar, birbirinin yerine kullanırlar (Jiang ve Devadas, 2009).

Bütün Boolean fonksiyonları SOP formunda yazılabilir. Doğruluk tablosu gösteriminin tersine, SOP formu normal gösterim(canonical) değildir. Bu sebeple, bir Boolean fonksiyonunu en sade haliyle SOP formunda yazmak iki seviyeli lojik sadeleştirme olarak adlandırılır ve çok zordur. Problemin çözümü belirsiz polinom zamanı (Nondeterministic polynomial time, NP-Complete) gerektirir.

Toplamların çarpımı (Product of Sums, POS), veya bilgisayar bilimlerindeki adıyla "Ve Bağlacı Normal Şekli" (Conjunctive Normal Form, CNF), değişkenlerin toplamının çarpımından oluşan özel bir Boolean formülü şeklidir. POS formu iki seviyeli devre tasarımının (ilk seviye OR kapısı, ikinci seviye AND kapısı) gösterimine karşılık gelmektedir.

Örnek 1.7; Fonksiyon $f = (\neg a + b + c) . (a + \neg b + c) . (a + b + \neg c) . (a + b + c)$ ifadesi POS formundadır.

Bütün Boolean fonksiyonları POS formunda yazılabilir. SOP ve POS ifadeler birbirine çok benzemesine rağmen, devre tasarımında genellikle SOP ifadeler kullanılır. Çünkü tamamlayıcı metal oksit yarıiletken transistor (complementary metal oxide semiconductor transistor, CMOS) devre tasarımında, N kanallı Mosfet (N channel Mosfet, NMOS) devreleri, P kanallı Mosfet (P channel Mosfet, PMOS) devrelerine göre daha çok tercih edilir.

1.3. Lojik Fonksiyonların Sadeleştirilmesi

Lojik ifadeler üzerine on yıllardır çeşitli çalışmalar yapılmaktadır. Günümüzde kriptoloji, sağlık, video sıkıştırma gibi farklı alanlarda lojik sentez önemli bir yer tutmaktadır. Daha az malzeme kullanmak, daha anlaşılır ifadeler elde etmek ya da daha etkili bellek kullanmak için lojik ifadelerin sadeleştirilmesi büyük bir önem taşımaktadır. Lojik ifadelerin sadeleştirilmesi için çok çeşitli algoritmalar ve programlar geliştirilmiştir (Fiser ve ark., 2003; Başçiftçi, 2010a; Martins ve ark., 2012).

1.3.1. Geleneksel sadeleştirme teknikleri

Boolean fonksiyonlarını sadeleştirmek için uygulanan ilk yöntem matematik işlemleri olmuştur. Ardından çok boyutlu fonksiyonları iki boyutlu tablolara yerleştiren Karnaugh, hem sadeleştirme işlemini gözle görülür, anlaşılır kılmış, hemde fonksiyon sadeleştirmek için komşuluk ilişkilerine dayalı çok farklı bir yöntem önermiştir.

1.3.1.1. Matematik işlemleriyle sadeleştirme

Doğruluk tablosundaki veriler matematiksel olarak ifade edilirken POS ya da SOP formunda yazılabilir. SOP ya da POS formunda yazılmış bir fonksiyon üzerinde Boolean cebri işlemleri yapılabilir. Az sayıda giriş değişkenine sahip Boolean fonksiyonların sadeleştirilmesi için Boolean Cebri kullanılabilir. Bir fonksiyonu matematiksel yöntemlerle sadeleştirmek için, matematik problemi çözüyormuş gibi uygun olan işlem kuralları uygulanır. Çizelge 1.3.'de Boolean cebri sadeleştirme kuralları verilmiştir.

Çizelge 1.3. Boolean cebri sadeleştirme kuralları

$A + 0 = A$	$A \,.\, \neg A = 0$	$A + B \,.\, C = (A + B) \,.\, (A + C)$
$A + 1 = 1$	$\neg\neg A = A$	$A + A \,.\, B = A$
$A \,.\, 0 = 0$	$A \,.\, B = B \,.\, A$	$A \,.\, (A + B) = A$
$A \,.\, 1 = A$	$A + B = B + A$	$A + \neg A.B = A + B$
$A + A = A$	$(A \,.\, B) \,.\, C = A \,.\, (B \,.\, C)$	$A \,.\, (\neg A + B) = A \,.\, B$
$A \,.\, A = A$	$(A + B) + C = A + (B + C)$	$\neg(A + B) = \neg A \,.\, \neg B$
$A + \neg A = 1$	$A \,.\, (B + C) = A \,.\, B + A \,.\, C$	$\neg(A \,.\, B) = \neg A + \neg B$

Sadeleştirme kuralları uygulanarak fonksiyonun en sade haline ulaşılmaya çalışılır. Fakat Boolean cebri işlemleriyle yapılan sadeleştirme en sade hali olmayabilir. Boolean kurallarını kullanarak ifadeleri sadeleştirmenin temel bir sistemi yoktur. Ayrıca bilgisayar mantığına göre (programsal olarak) bir sisteme oturtmak çok zordur. Bu tekniği, sadece 4 değişkenden oluşan basit fonksiyonların sadeleştirilmesinde kullanmak için bile deneyim gerekmektedir.

Örnek 1.8; 3 değişkenli Boolean $f = \neg a.b.c + a.\neg b.c + a.b.\neg c + a.b.c$ fonksiyonunu matematiksel işlemlerle sadeleştirirken öncelikle sadeleştirmede kullanılacak "a.b.c" çarpımı çoğaltılır.

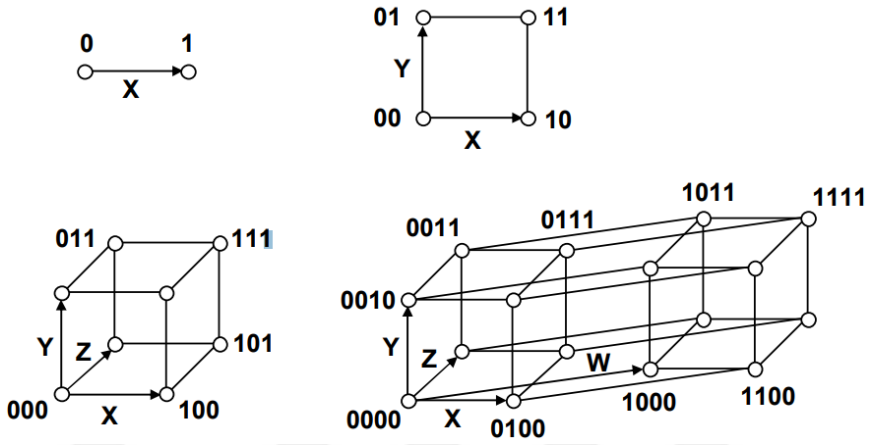
$$f = \neg a.b.c + a.\neg b.c + a.b.\neg c + a.b.c + \mathbf{a.b.c} + \mathbf{a.b.c}$$
$$= (\neg a.b.c + a.b.c) + (a.\neg b.c + \mathbf{a.b.c}) + (a.b.\neg c + \mathbf{a.b.c})$$
$$= (\neg a + a).b.c + (\neg b + b).a.c + (\neg c + c).a.b$$
$$= b.c + a.c + a.b$$

1.3.1.2. Karnaugh haritası

En çok bilinen SOP sadeleştirme yöntemi 1953 yılında Karnaugh tarafından geliştirilen görsel haritalandırma metodudur (Karnaugh, 1953). Karnaugh haritasında satırlara ve sütunlara potansiyel girişler yerleştirilir. Bunların kesiştiği hücrelere 1 ve 0'lar yerleştirilir. Bu haritadan bakarak ifadeler gruplanır. Bu görsel tablo sayesinde Lojik ifadeler toplamların çarpımı ya da çarpımların toplamı şeklinde ifade edilebilirler.

Karnaugh Haritası (Karnaugh Map, KM) Boolean uzayındaki komşuluk ilişkilerini görselleştiren güzel bir yöntem sunmaktadır. Bir KM, *n* boyutlu küpün küpteki

komşuluk ilişkileri gözetilerek iki boyutlu yüzeye yansıtılmasıdır. Şekil 1.2., 1, 2, 3 ve 4 değişkenli Boolean fonksiyon vektörlerini (X, Y, Z ve W), Şekil 1.3., bu fonksiyonlara ait KH'leri göstermektedir.



Şekil 1.2. 1, 2, 3 ve 4 boyutlu Boolean fonksiyon uzayı (NAND NOR cubes, 2009)

0	1

00	01
10	11

000	001	101	100
010	011	111	110

0000	0001	0101	0100
0010	0011	0111	0110
1010	1011	1111	1110
1000	1001	1101	1100

Şekil 1.3. 1, 2, 3 ve 4 değişkenli Boolean fonksiyonu için KH'ler

Boolean fonksiyonundaki tek bir giriş değişkenine "literal" denir. Tek veya daha fazla literalden oluşan Boolean çarpımına, çarpım terimi veya implikant (implicant) denir. Bütün implikant'lar Boolean uzayında bir köşeye veya karşılığı olan KH'de bir bölüme karşılık gelir. Asal implikantlar (AI) ise başka hiçbir implikant tarafından kapsanmayan implikantlardır. Boolean fonksiyonlarını sadeleştirmek için AI'ler aranır, çünkü asal implikantlar diğer implikantların kapladığından daha çok minterm kaplar ve daha az literal ile ifade edilebilirler.

KH yardımıyla sadeleştirme yaklaşımı az girişli fonksiyonlar için uygulama kolaylığından dolayı tercih edilebilir. Fakat çok değişkenli fonksiyonları KH'de göstermek ve KH kullanarak sadeleştirmek çok zordur. Ayrıca, KH ile sadeleştirme

yönteminde, bir fonksiyonu temsil eden en sade AI kümesini bulmak için uygulanacak kurallar açıkça tanımlanamamaktadır.

1.3.2. Modern sadeleştirme teknikleri

Sürekli gelişen dijital elektronik devreleri beraberinde karmaşık Boolean fonksiyonlarını getirmektedir. Geleneksel sadeleştirme tekniklerinin yetersiz olduğu durumlarda araştırmacılar bilgisayar yardımıyla da hesaplanabilecek teknikler geliştirmiştir. Tablolama tekniği iki seviyeli sadeleştirme dediğimiz AI oluşturma ve kapsama aşamalarını içermektedir. ESPRESSO programı sezgisel sadeleştirme yapabilmektedir. İkili karar ağaçları Boolean fonksiyon sadeleştirmede kullanılabilmektedir. Bu bölümde modern sadeleştirme teknikleri detaylı olarak incelenmiştir.

1.3.2.1. Quine McCluskey algoritması

Çok değişkenli fonksiyonları KH yardımıyla sadeleştirmek zordur. Bu sebeple 1952 ve 1956 tarihlerinde Quine (1952) ve McCluskey (1956) tarafından algoritma tabanlı tablolama tekniği geliştirilmiştir. Quine ve McCluskey tarafından geliştirilen tablolama tekniği, Karnough Haritası ile sadeleştirilmesi zor olan karmaşık fonksiyonları işleyebilmektedir. Ayrıca algoritması uygun olduğu için bilgisayar tarafından programlanması mümkündür.

Quine-McCluskey algoritması iki seviyeli Boolean sadeleştirme problemini çözmek için düzgün ve ideal bir yöntem sunmaktadır. İki seviyeli lojik sadeleştirmenin gerekli teorik zeminini Quine, W.V. (1952; 1955) hazırlamıştır. Fakat sadeleştirme işlemini otomatik hale getiren kusursuz algoritmayı McCluskey (1956) tasarlamıştır.

Quine-McCluskey yönteminin iki aşaması vardır: öncelikle bütün AI'lar üretilir, sonra fonksiyonu örten en az sayıda ve en sade AI'lar seçilir. İlk aşamada, daha az literal içeren implikantlar üretmek için fonksiyonun temel implikantları mümkün olduğu kadar tekrar birleştirilir. Sonuçta bütün AI'lar böyle üretilir. İkinci aşamada, temel implikantları kapsayan en küçük AI kümesi seçilir.

Örnek 1.9; Çizelge 1.4.'de doğruluk tablosu verilen $f(a, b, c, d)$ fonksiyonunu Quine-McCluskey algoritmasıyla sadeleştirin.

Çizelge 1.4. Fonksiyon f (a, b, c, d) için doğruluk tablosu

Minterm'ler (*a, b, c, d*)	**Fonksiyon Değeri**	**Minterm'ler (*a, b, c, d*)**	**Fonksiyon Değeri**
0000	DC	1000	0
0001	0	1001	0
0010	1	1010	0
0011	1	1011	1
0100	0	1100	1
0101	0	1101	1
0110	0	1110	DC
0111	0	1111	1

$f(a, b, c, d)$ fonksiyonu her bir giriş değişkeni kombinasyonuna göre doğru (1), yanlış (0) veya *DC* değeri alır. Asal implikantları üretmek için implikantlar mümkün olduğu kadar genişletilmelidir. Bu sebeple ON minterm kümesi ile *DC* kümesi birleştirilerek Çizelge 1.4.'ün 1. ve 3. sütununda temel implikant olarak listelenmiştir. Bu tabloda, 0'lar giriş değerlerinin tamamlayıcısını (¬), 1'ler giriş değerlerinin kendisini göstermektedir. Örneğin "1010" ifadesi "a¬bc¬d" mintermi anlamındadır. Olası bütün implikant kombinasyonlarının belirlenmesi gerekir. Komşu olmayan implikantları birleştirmek mümkün değildir. Çünkü bu implikantların birden fazla değişkenleri farklı değerler almıştır. "0000" ile "0010" mintermleri birleştirildiğinde "00X0" implikantı oluşur, aynı şekilde "0010" ile "0011" mintermleri "001X" implikantını oluşturur. Çizelge 1.5.'in ikinci sütununda oluşan ikinci derece implikantlar gösterilmiştir. Bu implikantlar tekrar kontrol edildiğinde 4 implikantı (110X, 11X0, 11X1, 111X) kapsayan "11XX" implikantının oluştuğu üçüncü sütunda görülmektedir. Eğer daha büyük bir fonksiyon olsaydı, mümkün olduğu kadar implikantların birleştirilmesi işlemi tekrar edecekti.

Çizelge 1.5. AI'ların bulunması

Temel İmplikant (*a, b, c, d*)	**İkinci derece implikant**	**Üçüncü derece implikant**
0000	00X0	11XX
0010	001X	
0011	X011	
1100	110X	
1011	11X0	
1101	1X11	
1110	11X1	
1111	111X	

Çizelge 1.6. Fonksiyon kapsama probleminin çözümü

Mintermler	**Asal İmplikantlar**				
	00X0	**001X**	**X011**	**1X11**	**11XX**
0010	+	+			
0011		+	+		
1011			+	+	
1100					+
1101					+
1111				+	+

Quine-McCluskey algoritması, asal implikantlar bulunduktan sonra ikinci aşamada, fonksiyonu örten en sade AI kümesini bulmaktadır. Çizelge 1.5.'in ilk sütununda kapsanması gereken mintermler bulunmaktadır. Kapsama problemi çözülürken *DC* mintermler önemsenmemektedir. Bu sebeple *DC* mintermleri burada listelenmemiştir. *DC* mintermler sadece AI'ların üretilmesi aşamasında değerlendirilmiştir. Çizelge 1.5.'in ikinci sütununda listenen AI'lar ise ON mintermlerini kapsayabilecek kaynaklardır. Çizelge 1.6.'da satır ve sütünların kesiştiği hücrelerde o satırdaki mintermi kapsayan AI'lar + ile işaretlenmiştir.

Kapsama probleminde amaç bütün satırları en düşük maliyetli AI ile kapsamaktır. McCluskey en düşük maliyeti, içerdikleri literal sayısına bakmadan, en az sayıda AI olarak tanımlamıştır. En düşük maliyet farklı değişkenlere göre hesaplanabilir. Örneğin AI'lardaki literal sayısına göre en düşük maliyet hesaplanabilir. Kapsama problemini hızlandırmak için çeşitli sezgisel yöntemler geliştirilebilir. Mesela bir mintermi kapsayan sadece tek bir AI var ise, bu AI maliyet hesabı yapılmadan doğrudan çözüm kümesine eklenebilir. Fakat birçok AI tarafından kapsanan mintermler için karar vermek zordur. En

iyi kapsama algoritmaları geçici kararlar alarak sonucu değerlendirir ve sonra gerekirse başa dönüp farklı kararları değerlendirirler.

İki seviyeli sadeleştirmede, kapsama probleminin zorluk derecesi NP-Complete'dir. Bu sebeple, değişken sayısı arttıkça ihtimaller katlamalı olarak arttığı için yüksek değişkenlerin hesaplanmasında bu sistem çok fazla bilgisayar hafızasına ve işlem zamanına ihtiyaç duymaktadır. Bu durum algoritmayı verimsiz hale getirmektedir (Rhyne ve ark., 1977).

1.3.2.2. Espresso algoritması

Karmaşık fonksiyonlarda yaşanan zorlukları aşmak için Brayton ve arkadaşları tarafından 1984 yılında Berkeley laboratuvarlarında ESPRESSO algoritması geliştirilmiştir. ESPRESSO algoritması, MINI algoritmasının gelişmiş versiyonudur. Günümüze kadarki en meşhur ve güçlü algoritmalardan birisidir. Bu algoritmanın mantığı diğerlerinden farklıdır. Lojik fonksiyonları küplere bölmek yerine program *ON*, *OFF* ve *DC* küpleri üretmektedir. Fonksiyon değerini 1 yapan mintermlere *ON*, fonksiyon değerini 0 yapan mintermlere *OFF,* fonksiyon üzerinde etkisi olmayan mintermlere de "Dont Care" (*DC)* denir. Sadeleştirme sonucunun en sade sonuç olduğu garanti edilemese de çok yakın sonuçlar vermektedir. ESPRESSO, hafıza ve zaman açısından kendinden önce geliştirilen algoritmalardan çok daha verimlidir. Bu algoritma ile onlarca değişken ve onlarca çıkış fonksiyonu kabul edilebilir zaman ve hafıza kullanımıyla hesaplanabilmektedir.

Bu algoritmanın sürekli tekrar eden 4 aşaması vardır.

i) Genişlet (Expand),
ii) Asal implikant (AI) bul (Essential prime),
iii) Kapsa (Irredundant cover),
iv) Sadeleştir (Reduce).

Bu algoritma öncelikle asal terime ulaşana kadar mintermleri genişletir, ardından asal terimleri *DC* yapar, artık gerekmeyen küpleri siler ve ifadeleri mümkün olduğu kadar az terimle ifade edecek kadar daraltır. ESPRESSO algoritmasının kaba kodu Şekil 1.4.'de verilmiştir.

```
PROGRAM ESPRESSO (N,F,D) //Sırasıyla On, Off ve DC minterm kümeleri
BEGİN
  F = Tamlayan_Al(N+D); //On ve DC olmayan mintermler OFF kümesine
  N = Genislet(N,F); //İlk genişletme
  N = Kapsa(N,D); // İlk kapsama
  A = AI_Bul(N,D) // AI'ların bulunması
  N = N - A; // Bulunan AI'lar N'den çıkarılıyor
  D = D + A; // Bulunan AI'lar D kümesine ekleniyor
  WHILE Maliyet(N) düştüğü sürece DO //
      N = Sadeleştir(N,D); // Kalan N kümesi sadeleştiriliyor
      N = Genişlet(N,F); // N kümesi F kümesine göre genişletiliyor
      N = Kapsa(N,D); // N kümesi D kümesi ile birlikte kapsanıyor
  ENDWHILE;
  D = D - A // Bulunan AI'lar D kümesinden çıkarılıyor
  N = N + A; // Bulunan AI'lar N kümesine iade ediliyor
  N = AYIKLA(N,D,F); //AI'lar son kez ayıklanıyor
  RETURN N;
END
```

Şekil 1.4. ESPRESSO algoritması kaba kodu (pseudo code)

Çok büyük ölçekli devrelerde kesin sonuca ulaşmak çok zaman ve hafıza gerektirir. Bu durumda yakın minimum sonuç tercih edilebilir. ESPRESSO, bu gibi durumlar için geliştirilmiş tecrübesel/sezgisel algoritmalardan birisidir.

Lojik ifadelerin sadeleştirilmesinde sezgisel algoritmaların ilki IBM tarafından geliştirilen MINI programıdır (Hong ve ark., 1974). Geliştirilen program bütün asal implikantları üretmemekte, sadece istenilen bazı implikantları asal implikant yapmaktadır. Bu programın kullandığı algoritma, sadeleştirme işleminde kullanılan mintermleri birer nokta olarak algılamakta, bu noktaları birleştirerek asal implikant küpleri oluşturmaktadır.

Brown, durum makinesi sentez programı (state machine synthesis program, SMS) adını verdiği bir program geliştirmiştir. Geliştirilen program belirsiz terimleri içeren ve içermeyen iki fonksiyon oluşturmaktadır. PRESTO algoritmasını kullanan programın temelde 2 döngüsü vardır. Öncelikle giriş fonksiyonundaki her çarpım teriminden, her literali sırasıyla çıkarır. Çarpım ifadesi sonuçta hala kapsanıyorsa program literali dışarıda bırakır. Ardından çıkış fonksiyonundaki her çarpım teriminden bir çıkış literalini çıkarır, başka bir implikant tarafından o literal kapsanıyorsa, literal dışarıda bırakılır. Fakat sonuç değişiyorsa, literal yerine tekrar koyulur ve diğer literaller benzer şekilde kontrol edilir (Brown, 1981).

Kang, 1981 yılında Stanford Programmable Array Minimizer (SPAM) algoritmasını kullanan otomatik Programmable Logic Array (PLA) sentez sistemini geliştirmiştir. Bu program 72 giriş, 144 çıkış, binlerce çarpım ifadesine sahip

fonksiyonları işleyebilmiştir. Bu programın sonlu durum makinelerini tasarlamak için uygun bir araç olduğu söylenebilir. Durum diyagramı çizilebilen herhangi bir dijital kontrol devresi bu sistemle tasarlanabilmektedir (Kang ve vanCleemput, 1981).

Brayton ve arkadaşları 1981 yılında ESPRESSO-1 ve ESPRESSO-2 programlarını tanıtmışlardır. Bu programlar karmaşık lojik fonksiyonların etkin bir şekilde sadeleştirilmesinde uzun yıllar kullanılmıştır. Rudell, ESPRESSO programının çok değerli mantık fonksiyonlarının minimizasyonunda kullanılabilen yeni versiyonunu (ESPRESSO -MV) geliştirmiştir (Rudell, 1986).

1.3.2.3. İkili karar diyagramları

İkili karar diyagramları (Binary decision diagrams, BDDs) ilk olarak Lee, C.Y. (1959) tarafından tasarlanmış ve Akers, S.B. (1978) tarafından daha da geliştirilmiştir. BDD'ler Boolean fonksiyonlarının normal gösterim (canonical) formu değildir. Normal gösterim formu oluşturmak için Bryant (1986; 1992) BDD değişken sıralama üzerinde sınırlandırmalar önermiş ve birçok indirgeme kuralı sunarak indirgenmiş sıralı BDD'yi (reduced ordered BDD, ROBDD) tanıtmıştır. Birçok karar diyagramı arasından Boolean fonksiyon sadeleştirmeye en uygunu ROBDD'lerdir.

n-girişli Boolean $f(\varphi_1, \ldots\ldots, \varphi_n)$ fonksiyonunu temsil etmek için *n* seviyeli ikili karar diyagramı kullanılabilir. İkili karar diyagramı, ikili ağaç, her düğümün en fazla iki çocuğunun olduğu, iki çeşit düğüm içerir. Uç düğüm veya yaprak düğümün, $deger(\gamma) \in \{0, 1\}$ arasında bir değer alma özelliği bulunur. Ara düğümler ise seviye indeksi $indeks(\gamma) \in \{0, ..., n\}$ ve iki çocuk düğüm özelliği vardır. Çocuk düğümler 0-çocuk, $yok(\gamma) \in V$ ve 1-çocuk, $var(\gamma) \in V$ şeklinde adlandırılabilir. Eğer $indeks(\gamma) = i$ ise φ_i'ye γ düğümü için karar değişkeni denir. Bir BDD'deki bütün γ düğümleri, denklem 1.15.-1.17. de rekürsif olarak tanımlanan Boolean fonksiyonu $f[\gamma]$'ye karşılık gelir.

i. Yaprak düğüm γ için,

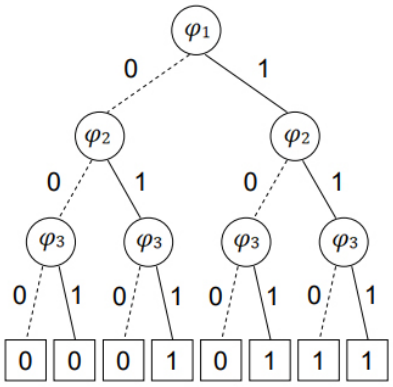
$$deger(\gamma) = 1 \Rightarrow f[\gamma]=1 \quad (1.15)$$

$$deger(\gamma) = 0 \Rightarrow f[\gamma]=0 \quad (1.16)$$

ii. $indeks(\gamma) = i$ olan ara düğüm γ için,

$$f[\gamma](\varphi_1, \ldots\ldots, \varphi_n) = \neg\, \varphi_i.f[yok(\gamma)](\varphi_1, \ldots\ldots, \varphi_n) + \varphi_i.f[var(\gamma)](\varphi_1, \ldots\ldots, \varphi_n) \quad (1.17)$$

Shannon açılımına göre f Boolean fonksiyonu $f = \varphi_i f_{\varphi i} + \neg\varphi_i f_{\varphi i}$ şeklinde yazılabilir. Bir BDD düğümünün, φ_i değişkeni tarafından kontrol edilen bir f Boolean fonksiyonu temsil ettiği varsayılırsa onun 0-çocuğu $f_{\neg\varphi i}$, ve 1-çocuğu $f_{\varphi i}$ dir. Tam bir ikili ağaçta BDD'nin yapraklarına her zaman fonksiyona göre değer verilebilir. Şekil 1.5'te görüldüğü üzere, $\varphi_1, \ldots\ldots, \varphi_n$ değişkenleri tarafından oluşturulabilecek bütün ihtimaller ikili ağaçta bir yaprak tarafından temsil edilebilir.



φ_1	φ_2	φ_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Şekil 1.5. Çoğunluk fonksiyonunun ikili ağaç gösterimi

Örnek 1.10; Çoğunluk fonksiyonunun ikili ağaç gösterimi Şekil 1.5.'te verilmiştir. Yuvarlaklar ara düğümleri, kare şekiller yaprakları, düz çizgiler 1 ihtimalini, kesik çizgiler 0 ihtimalini temsil etmektedir.

Tanım 1.1; Eğer kökten yaprağa kadar bütün düğümler aynı değişken sıralamasını takip ediyorsa bu ikili ağaç yapısına sıralı BDD (ordered BDD, OBDD) denir.

Tanım 1.2; Eğer T_1 ağacındaki herhangi bir φ düğümü $f(\varphi)=\beta$ *ise,* T_2 ağacındaki β düğümü için aşağıdaki koşulları sağlayan birebir f fonksiyonu varsa T_1 ve T_2 OBDD'leri eş biçimli (isomorphic)'tir.

i. φ ve β yaprak düğüm ise deger(φ) = deger(β) olmalı,

ii. φ ve β ara düğüm ise indeks(φ) = indeks(β), f(var(φ)) = var(β) ve f(yok(φ)) = yok(β) olmalıdır.

OBDD'ler tek kök ve ara düğümlerin çocuklarından oluşur. Eş biçimli ağaçlarda T_1 ağacının kökü, T_2 ağacının köküne, T_1 kökünün 0-çocuğu T_2 kökünün 0-çocuğuna ve benzer şekilde diğer düğümlerde de eşleyen bire bir bağlantı olmalıdır. Bu şekilde iki OBDD'nin eş biçimli olduğu doğrusal zaman karmaşıklığında kontrol edilebilir.

Tanım 1.3; (Bryant, 1986) Bir OBDD'nin hiç bir düğümünün çocukları birbirine eşit (var(φ) = yok(φ)) değilse ve iki düğümünün alt ağaçları eş biçimli değilse bu ağaca indirilmiş OBDD(reduced OBDD, ROBDD) denir.

Aşağıdaki indirgeme kuralları uygulanarak herhangi bir OBDD'den ROBDD üretilebilir.

i. Aynı değer özelliğine sahip iki yaprak birleştirilir.
ii. Aynı karar değişkenine sahip (aynı 0-çocuk, aynı 1-çocuk) iki ara düğüm birleştirilir.
iii. var(φ) = yok(φ) özelliğinde bir düğüm varsa kaldırılır. Çocuk düğüm üst düğüme bağlanır.

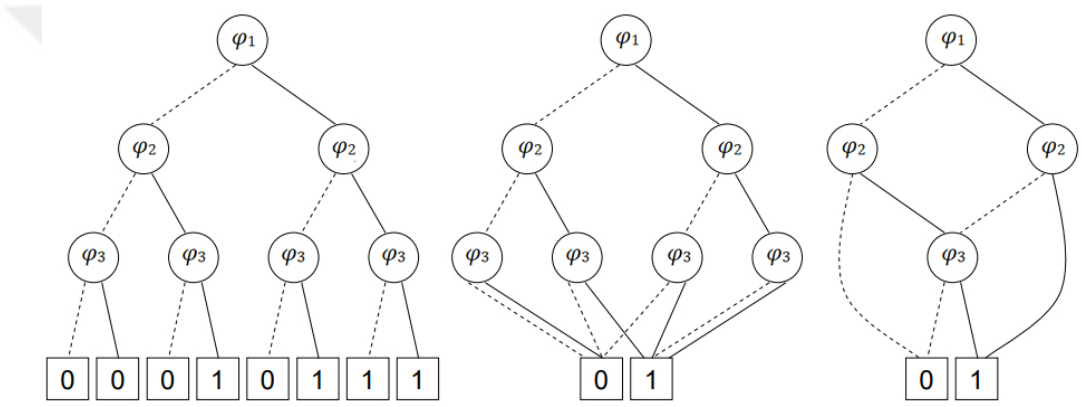
Herhangi bir OBDD üzerinde yapraklardan başlayarak yukarı düğümlere doğru yukarıdaki indirgeme adımları uygulanırsa aynı ağacın ROBDD karşılığına ulaşılır. Bu kurallar ROBDD'nin hiçbir düğümünün yapısal ve işlevsel olarak eş biçimli olmadığını ve ROBDD'nin verilen değişkenlere göre en az sayıda düğümünün olduğunu garanti eder. Ayrıca ROBDD'nin bütün düğümleri farklı Boolean fonksiyonlarını temsil eder. Aynı Boolean fonksiyonunu temsil eden iki BDD varsa, bunlar eş biçimli olmalıdır. Dolayısıyla ROBDD'ler Boolean fonksiyonların normal gösterimleridir ve her fonksiyonun eşsiz bir ROBDD'si vardır.

***Teorem 1*:** (Bryant, 1986) Herhangi bir f Boolean fonksiyonunu gösteren eşsiz bir ROBDD vardır fakat daha fazla düğüm içeren OBDD'ler olabilir.

Yukarıdaki indirgeme kuralları teoremi doğrulamaktadır. OBDD'ler sadeleştirilmediği için daha fazla düğüm içerirler ve birden fazla OBDD aynı fonksiyonu temsil edebilir. ROBDD'ler fonksiyonun en sade hali olduğu için fonksiyonu temsil eden sadece bir tane ROBDD olabilir.

Örnek 1.11; İkili ağaç yapısıyla temsil edilen boolean f fonksiyonunun indirgeme kuralları uygulanarak ROBDD'ye dönüştürülmesi Şekil 1.6'da gösterilmektedir. Öncelikle ilk indirgeme kuralı uygulanarak aynı değere sahip yapraklar birleştirilir. Toplamda sadece 2 yaprak kalır; bunlar 0 değeri ve 1 değeri olan yapraklardır. Ardından aynı karar değişkenindeki düğümlerden aynı çocuklara (0-çocuk, 1-çocuk) sahip olanlar birleştirilir. Son olarak, hem 0-çocuk hemde 1-çocuk değeri aynı olan düğümler kaldırılarak üst düğümden alttaki düğüme bağlantı verilir.

ROBDD ağaçlarda değişken sıralamasının değişmesi çok farklı ağaçların oluşmasını sağlar. Fonksiyon aynı olsa bile değişkenlerin sıralaması (ağaçtaki seviyesi) değiştiği için bağlantılar ve indirgenebilecek düğümler de değişir.

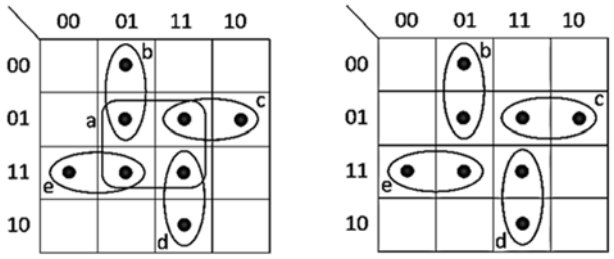


Şekil 1.6. Boolean fonksiyonu ikili ağaçtan ROBDD'ye indirgeme

1.4. İzole Mintermler

Lojik ifadelerin sadeleştirilmesi için hangi yöntem kullanılırsa kullanılsın, sadeleştirmeye nereden başlanacağı büyük önem taşımaktadır. Sadeleştirmeye göreceli olarak merkezdeki mintermlerden başlanırsa sonuç doğru çıkmayabilir, daha doğrusu en sade sonuca ulaşılamayabilir (Abd-El-Barr ve Khan, 2014). Göreceli olarak en uzak mintermler öncelikle sadeleştirilirse, merkezde bulunan mintermler de kapsanabileceğinden bunların işlenmesine gerek kalmayabilir ve en sade sonuca ulaşılabilir. Denklem 1.18'de fonksiyonun değerini 1 yapan *ON* mintermlerinin kümesi (SON), Şekil 1.7'de bu kümeye ait KH bulunmaktadır. Harita üzerindeki noktalar *ON* mintermlerini; ovaller, içerisindeki mintermleri kapsayan AI göstermektedir.

SON={0001,0101,0111,0110,1100,1101,1111,1011} (1.18)



(a) (b)

Şekil 1.7. En iyi ve en kötü SOP ifadeler

Eğer Şekil 1.7(a)'daki gibi mintermlerin kapsanmasına merkezden başlanırsa *a, b, c, d, e* AI'ları oluşur. Bu AI'ları kapsayan çözüm kümesi S={x1x1+0x01+011x+1x11+110x} denklem 1.18 için en kötü SOP ifadesidir.

Şekil 1.7(b)'deki gibi mintermlerin kapsanmasına en dışarıdaki mintermlerden başlanırsa *b, c, d, e* AI'ları oluşur. *a* AI oluşmasına gerek duyulmaz. Çözüm kümesi S={0x01+011x+1x11+110x} denklem 1.18 için en iyi SOP ifadesidir.

Yukarıdaki örnekten anlaşıldığı gibi sadeleştirmeye hangi mintermle başlanacağı en sade sonuca ulaşmak için önemlidir. Diğer mintermlere göre daha dışarıda bulunan mintermlerin kapsanması zor ve onu kapsayan AI sayısı azdır. Bu sebeple göreceli olarak dışarıda bulunan izole mintermlerin öncelikle sadeleştirilmesi gerekmektedir. Mintermlerin bulundukları yer kadar komşuluk ilişkileri de önemlidir. Bazı mintermler diğer mintermlerin ortasında bulunabilir. Bu mintermlerin herhangi bir komşusu yoksa bunlara da izole minterm denir ve öncelikle kapsanması gerekmektedir.

Doğrudan örtmede izole mintermin seçilmesi sonucu doğrudan etkilemektedir. Bu sebeple doğrudan örtme yönteminde minterm ve implikant seçiminde kullanılan 6 çeşit algoritma vardır.

- Rastgele minterm/rastgele implicant
- Pomper & Amstrong (max. AI)
- Besslich (izole MT. / max. AI)
- Dueck & Miller (max. iso. Fac./min. Relative Break Count)
- Gold
- BOOM

1.4.1. İzole mintermin bulunması

Lojik ifadelerin sadeleştirilmesinde mintermin ve AI'nın uygun seçilmesi sonucun daha sade olması ve daha kısa sürede sonucun bulunması açısından çok önemlidir. Bazı algoritmalar hem mintermi hem de AI rastgele seçerken, bazıları her ikisini de belirli kıstaslara göre seçmektedir. Pomper ve Armstrong algoritmasında minterm rastgele belirlenmekte fakat bu mintermi örten AI içerisinden en fazla mintermi örteni seçilmektedir. Bu algoritmada mintermleri örten bütün implikantlar belirlenir ve en çok mintermi önemsiz yapanı seçilir (Pomper ve Armstrong, 1981). Pomper_ve Armstrong algoritması kaba kodu Şekil 1.8'de verilmiştir.

```
PROGRAM POMPER AND ARMSTRONG
BEGIN
{ F=Giris Fonksiyonu
  S=Cozum Kumesi=Ø
  While ( F ) do
  { α= F kümesinden rastgele minterm al
    I=max_Implicant(α)
    S=S+I
    F=F-α
  }
}
END
FUNCTION MAX_IMPLICANT(α)
BEGİN
{ Imp_size = -∞
  Covered_Minterm= -∞
  For (α kapsayan her implikant I için) do
  { I_covered_min=0 veya DC yapılan minterm sayısı;
    I_size=Implikant size
    If(I_covered_min>Covered_Minterm)
    { maximal_Implicant(α)=I
      Imp_size=I_size
      Covered_Minterm=I_covered_min
    }
  }
  RETURN maximal_Implicant
}END
```

Şekil 1.8. Pomper ve Armstrong kaba kodu (pseudo code)

Besslich algoritmasında her mintermin bir ağırlığı vardır. Merkezdeki mintermlerin ağırlığı daha fazla, dışarıdaki izole mintermlerin ağırlığı ise daha azdır. Bu algoritmada mintermlerin kapsanmasına izole mintermlerden başlanır. Mintermi örten bütün AI belirlenir. Her AI kapsadığı minterm sayısı, maliyetine bölünerek AI için etkililik faktörü hesaplanır. En etkili AI seçilerek kapsama tamamlanır (Besslich, 1986). Besslich algoritması kaba kodu Şekil 1.9'de verilmiştir.

```
PROGRAM BESSLİCH
BEGIN
{ lowest_wt = ∞
  α = 00…00
  for (0 veya DC olmayan bütün minterm(β)'lar)
  { wt(β)=∑1≤m≤n(βm-γm)
    If ( wt(β) <lowest_wt )
    { α=β
      lowest_wt=wt(β)
    }
  }
  RETURN lowest_wt
}END
```

Şekil 1.9. Besslich algoritması kaba kodu (pseudo code)

Dueck ve Miller algoritmasında her minterm için izolelik faktörü belirlenir. İzolelik faktörü bir mintermin komşu sayısı ile komşularının bulunduğu yön sayısının toplanmasıyla ters orantılıdır. Mintermlerin kapsanmasına hiç komşusu olmayan yüksek izolasyon faktörlü mintermden başlanır ve düşük faktör katsayısına sahip mintermlerle devam edilir. Minterm seçme kodu Şekil 1.10.'da verilmiştir. Ardından bu mintermleri örten bütün AI'lar oluşturulur. Her AI için göreceli kesme sayısı (RBC, relative break count) hesaplanır. Bu hesaplama AI fonksiyonu ne kadar sadeleştirdiğinin bulunmasını sağlar. Geriye kalan fonksiyonu en çok sadeleştiren AI seçilir (Tirumalai ve Butler, 1988; 1991). Dueck ve Miller algoritması, AI belirleme kaba kodu Şekil 1.11.'de verilmiştir.

```
PROGRAM DUECK_and_MILLER (F)
BEGİN
{ α = 00…00
  CFmin = ∞
  While (F fonksiyonu) do
  { CF(β) = Komşu Sayısı(β)+Komşuların yön sayısı(β)
    If (CFmin>CF(β))
    { CFmin = CF(β)
      α = β
    }
    F = F − β
  }
  RETURN α
}END
```

Şekil 1.10. Dueck ve Miller minterm seçme kaba kodu (pseudo code)

```
PROGRAM DUECK_MILLER_AI_SEC (F)
BEGIN
{ rbc(I) = 0
  for(I tarafından kapsanan bütün α mintermleri) do
  { if(I implikantı α yanındaki β kapsıyorsa)
    rbc(I) = rbc(I) - 1
    if(I implikantı α yanındaki β kapsamıyorsa)
    rbc(I) = rbc(I) + 1
  }
  cur_rbc = ∞
  for(minterm α'yı kapsayan bütün I implikantları)
  do
  { if ( rbc(I) <cur_rbc )
    { eniyi = I
      cur_rbc = rbc(I)
    }
  }
  RETURN eniyi
}
```

Şekil 1.11. Dueck ve Miller AI seçme kaba kodu (pseudo code)

Yukarıda bahsedilen her 3 algoritmanın da diğerlerinden daha iyi olduğu fonksiyon sınıfları vardır. Tirumalai ve Butler'in yaptığı araştırmada hiçbir algoritma bütün fonksiyonlarda diğerlerine tam üstünlük kuramamıştır. Pomper ve Armstrong algoritması rastgele seçimden daha iyi sonuçlar vermektedir. Dueck ve Miller ile Besslich algoritmalarının her biri Pomper ve Armstrong algoritmasından biraz daha iyi sonuçlar vermiştir (Tirumalai ve Butler, 1991).

Fiser tarafından geliştirilen BOOM algoritmasında mintermleri örten AI'ların hepsi hesaplanır ve AI havuzunda biriktirilir. Bütün AI'ler havuzda biriktirildiği için hangi mintermden başlanıldığının bir önemi yoktur. AI havuzda biriktirildiği için mükerrer AI oluşmaz. AI biriktirilmesi programın daha hızlı ve daha az bellek kullanarak çalışmasını sağlar. Kapsama problemi dört aşamada aşağıdaki sırayla çözülür (Bernasconi ve ark., 2012).

i) En çok *ON* mintermini kapsayan *AI* seç,

ii) Kapsaması zor olan *ON* mintermlerini kapsayan *AI* seç,

iii) Maliyeti en düşük olanları seç (en az sayıda literal içeren),

iv) Hala çok ihtimal varsa, birisini rastgele seç.

1.5. Paralel Programlama

Günümüzün hızla gelişen teknolojisi bilgisayarlar, yazılımların ihtiyaçlarına cevap vermekte zorlanmaktadırlar. Yazılımlar her geçen gün daha fazla hafıza, daha hızlı

grafik kartı ve daha hızlı bilgisayarlara ihtiyaç duymaktadırlar. Daha fazla yarı iletken kullanarak hafıza arttırılabilir. Fakat bilgisayarların işlemcinin saat hızı fiziksel limitlere dayandığı için, saat hızını arttırmak işlemcinin yanlış işlem yapmasına ya da ısınarak yanmasına sebep olmaktadır (Denning ve Lewis, 2017). Bu sorunu çözmek için mühendisler daha fazla işlemciyi paralel olarak aynı iş üzerinde çalışacak şekilde üretmeyi başarmışlardır. Günümüzde akıllı cep telefonlarında bile dört çekirdekli işlemciler oldukça yaygın olarak kullanılmaktadır.

Multiprocessing tek bir bilgisayar sisteminde birden fazla işlemcinin kullanılabilmesine ya da bu altyapıya sahip olmasına denir. Bu sistemde tek bir bilgisayar vardır, birden fazla işlemci ortak bilgisayar kaynaklarını paylaşmaktadırlar. Bu sisteme sahip olan bilgisayarlara çok işlemcili (Multiprocessor, MP) bilgisayarlar denir. Çok işlemcili bilgisayarları komut ve veri işlemesine göre, aralarındaki iletişime göre (veriyolu veya ağ bağlantılı), hafıza yapılarına göre (paylaşılan ya da dağıtık hafıza), veya iş paylaşımlarına göre (symetric, Asymmetric) sınıflandırmak mümkündür(Flynn, 1972).

Paralel programlama aracılığıyla daha büyük problemler daha kısa sürelerde çözüme ulaştırılmakta ve performans arttırılabilmektedir. Paralel programlama yöntemi, bilimsel gelişmelerle beraber ortaya çıkan karmaşık ve büyük problemlerin çözümünde problemlerin farklı kısımlarını farklı işlemcilere bölüştürerek gereksinim duyulan hızlanma ve etkinliğin sağlanmasında yardımcı olmaktadır. Bu tez çalışmasında lojik ifadelerin sadeleştirilmesini sağlayan paralel programlanabilir bir algoritma oluşturulmuş ve C# dilinde kodlanmıştır.

1.6. Tezin Amacı ve Önemi

Bilgisayar devreleri lojik ifadelerden oluşur. Ayrıca veri sıkıştırma, şifreleme, özellik indirgeme, Conjunctive Normal Form (CNF) ve Disjunctive Normal Form (DNF) dillerindeki ifadelerin gösterimi gibi çok farklı alanlarda lojik ifadeler kullanılmaktadır. Özellikle dijital elektroniğin bulunmasından itibaren lojik fonksiyonlar ve onların sadeleştirilmesi çok büyük önem taşımaktadır. Çok değerli devreler, çok değişkenli fonksiyonlar, quantum bilgisayarlar, lojik fonksiyonlara farklı anlamlar kazandırmıştır. Lojik fonksiyonların kullanıldığı hemen hemen bütün alanlarda lojik fonksiyonların sadeleştirilmesi büyük önem taşımaktadır. Dijital elektronikte kullanılan lojik fonksiyonların daha sade olması, maliyeti düşürür, performansı arttırır ve karmaşıklığı azaltır.

Günümüzde çıkış sayısına (tek çıkışlı-çok çıkışlı), giriş değerlerine (iki değerli-çok değerli) göre farklı sadeleştirme algoritmaları geliştirilmiştir. Bütün lojik fonksiyon sadeleştirme algoritmaları, kesin ve sezgisel olmak üzere iki sonuca göre sadeleştirme yapmaktadır. Kesin sadeleştirme yönteminde giriş fonksiyonunda belirtilen bütün mintermleri eksiksiz kapsayan en sade çıkış fonksiyonuna ulaşılır. Fakat kesin sadeleştirme çok fazla zaman ve bellek kullanımı gerektirmektedir. Sezgisel sadeleştirme algoritmaları giriş fonksiyonunda belirtilen mintermleri kapsayan çıkış fonksiyonunu vermektedir. Sezgisel sadeleştirme algoritmalarında en sade sonuca ulaşıldığının garantisi olmaz fakat kesin sadeleştirme algoritmalarıyla kıyaslandığında bu algoritmalar çok daha hızlı çalışır ve daha az bellek kullanır.

Sezgisel yaklaşıma göre yapılan sadeleştirmede sonucun en sade ifade olduğundan emin olunamaz, kesin sadeleştirmede ise çok yüksek çalışma zamanı ve bellek kullanımına ihtiyaç duyulur.

Lojik ifadelerin sadeleştirilmesinde, sadeleştirmeye nereden başlanacağı büyük önem taşımaktadır. Sadeleştirmeye göreceli olarak merkezdeki mintermlerden başlanırsa en sade sonuca ulaşılamayabilir. Diğer mintermlere göre daha dışarıda bulunan ya da komşusu olmayan izole mintermlerin kapsanması zor ve onu kapsayan AI sayısı azdır. Bu sebeple göreceli olarak dışarıda bulunan izole mintermlerin öncelikle sadeleştirilmesi gerekmektedir. Göreceli olarak en uzak mintermler önce sadeleştirilirse, merkezde bulunan mintermler de kapsanabilir ve bunların sadeleştirilmesine gerek kalmadığından en sade sonuca ulaşılabilir.

Bu tezin amacı, lojik fonksiyon sadeleştirme algoritmalarını inceleyerek, izole mintermleri tespit eden minterm seçme algoritması geliştirmek, lojik fonksiyonların sadeleştirilmesi için etkili bir Yakın Sonuç Kapsama Algoritması (YSKA) geliştirmek ve makul sürelerde en sade sonuca ulaşan Kesin Sonuç Kapsama Algoritması (KSKA) geliştirmektir. Geliştirilen izole mintermlerin tespiti algoritması hem YSKA'nda hem de KSKA'nda kullanılacaktır. Bu tez çalışmasında, izole mintermler tespit edilerek sadeleştirmeye izole mintermlerden başlanması, yakın sonuç ve kesin sonuç algoritmalarını işlem zamanı, bellek kullanımı ve çıkış fonksiyonu açısından nasıl etkilediği araştırılmıştır. Lojik fonksiyonların sadeleştirilmesinde izole mintermlerin tespit edilmesinin olumlu (daha sade çıkış fonksiyonu) ve olumsuz (işlem zamanı ve bellek kullanımını arttırma) etkilerinin değerlendirilmesi için bütün algoritmalar C# dilinde kodlanarak sonuçlar incelenmiştir.

Ayrıca bu tez çalışmasında, geliştirilen izole mintermlerin tespiti ve sadeleştirme algoritmalarının paralel programlamaya ne kadar uygun olduğunun araştırılması amaçlanmıştır. Hazırlanan her 3 algoritma paralel programlama ile kodlanmış ve elde edilen sonuçlar seri programlama sonuçlarıyla işlem zamanı, kullanılan hafıza ve bulunan sonuçlar açısından karşılaştırılmıştır. Böylece paralel programlamanın lojik fonksiyonların sadeleştirilmesinde nasıl kullanılabileceğinin ve ne kadar verim alınabileceğinin tespiti amaçlanmıştır.

Lojik fonksiyonlar veri şifreleme, resim sıkıştırma, özellik seçme, devre tasarımı gibi çok farklı alanlarda, farklı amaçlarla kullanılmaktadır. Geliştirilen fonksiyon sadeleştirme algoritmaları, lojik fonksiyonlarla ilgili farklı alanlarda kullanılabilir. Bu tezde geliştirilen ve C# programında kodlanan fonksiyon sadeleştirme algoritmalarının modüler, kolay kullanılabilen, sade ve başka programlarla birleştirilebilir bir yapıda olmasına dikkat edilmiştir. Birçok farklı alanda kullanılabilecek lojik fonksiyonların sadeleştirilmesi programının daha hızlı, daha doğru ve daha az kaynak kullanarak çalışması büyük önem arzetmektedir.

1.7. Tezin Organizasyonu

Lojik fonksiyonların, izole mintermlerden başlanarak seri ve paralel algoritmalarla sadeleştirilmesini araştıran bu tez çalışması aşağıdaki gibi organize edilmiştir.

Birinci bölümde entegre devreler, lojik fonksiyonlar ve gösterimleri tanıtılmış, lojik fonksiyonların sadeleştirilmesi alanında yapılmış çalışmalar incelenmiş, izole mintermlerin önemi ve tespit etme yöntemleri araştırılmış, paralel programlama alanında kısa bir özet verilerek, tez çalışmasının amacı ve öneminden bahsedilmiştir.

İkinci bölümde lojik fonksiyon sadeleştirme yöntemleri ve lojik fonksiyon sadeleleştirme uygulamaları alanında yapılmış araştırmalar incelenmiştir. Bu bölümde ayrıca lojik fonksiyonların sadeleştirilmesine faydalı olabilecek izole mintermlerin tespiti, fonksiyon sadeleştirmede kullanılabilecek teknikler, lojik fonksiyonların karmaşıklığı ve lojik fonksiyon sadeleştirmenin kullanıldığı temel uygulama alanları araştırılmıştır.

Üçüncü bölümde kullanılan materyal ve metot anlatılmıştır. Bu bölümde lojik fonksiyonlar, bitsel operatörler, doğrudan örtme algoritmaları ve petrick yöntemi

tanıtılmıştır. Ayrıca paralel ve rekürsif algoritmalar incelenerek, hangi programlama dilinin neden seçildiği açıklanmıştır.

Dördüncü bölümde tez çalışmasında geliştirilen İzole Mintermlerin Tespiti algoritması, YSKA ve KSKA sunulmuştur. Bu algoritmaların kullandığı bitsel operatör işlemleri açıklanarak algoritmaları paralel hale getirme çalışmaları tanıtılmıştır. Ayrıca geliştirilen programın kullanıcı arayüzü ve kullanılan dosya formatı açıklanmıştır.

Beşinci Bölümde boolean fonksiyonların sağlanabilirliliği ve geçerliliği, fonksiyonların karmaşıklık sınıfları verilmiştir. Ayrıca geliştirilen izole mintermlerin tespiti, YSKA ve KSKA algoritmalarının karmaşıklık analizleri yapılmıştır.

Altıncı bölümde geliştirilen algoritma ve programlar farklı fonksiyon dosyalarıyla test edilerek sonuçlar paylaşılmıştır. Bu bölümde izole mintermlerin sadeleştirmeye etkisi incelenerek, paralel programlamanın sadeleştirme algoritmalarını hızlandırma oranı irdelenmiştir.

Yedinci bölümde tez kapsamında gerçekleştirilen çalışmaların sonuçları değerlendirilerek daha başarılı sonuçlara ulaşabilmek için önerilerde bulunulmuştur.

2. KAYNAK ARAŞTIRMASI

Bu bölümde lojik fonksiyonların sadeleştirilmesi alanında yapılmış bilimsel araştırmalar incelenmiştir. İlk bölümde lojik fonksiyonların sadeleştirilmesi amacıyla kullanılan farklı yöntemler incelenmiş, ikinci bölümde lojik fonksiyonların sadeleştirilmesiyle yapılmış farklı uygulamalar incelenmiştir.

2.1. Fonksiyon Sadeleştirme Yöntemleri Kaynak Araştırması

J. Paul Roth ve R.M. Karp (1962), kapı tipi anahtarlama devrelerinin graflar yardımıyla sentezlenmesinde kullanılabilecek yeni bir prosedür tanımlamıştır. Bu prosedür sayesinde Boolean graf tarzında ifade edilmiş her bir devre üzerinde Boolean fonksiyonu yeniden yapılandırılabilmiştir. Bu araştırmada fonksiyonel yapılandırmaya genel bir yaklaşım verilmiş ve bazı örnekler denenmiştir. Bu sayede bazı tasarım örnekleri yeniden yapılandırılarak en az maliyetli Boolean graflarına ulaşılmıştır.

Randal E. Bryant (1986), Boolean fonksiyonlarının sadeleştirilmesinde kullanılabilecek yeni bir veri türü (grafları) tanımlayarak, sadeleştirme algoritması geliştirmiştir. Yeni tanımlanan yönlü graf, Lee ve Akers tarafından geliştirilen grafa benzer olmakla birlikte, graftaki karar değişkenlerinin sıralamasında bazı kısıtlamalara gidilmiştir. Önerilen algoritmanın zaman karmaşıklığı verilen grafın büyüklüğüyle orantılıdır ve böylece graf çok büyük olmadığı sürece etkili olduğu söylenebilir.

Rudell R. L. (1989), büyük ölçekli devre tasarımında kullanılan, birlikte tam bir set oluşturan, lojik optimizasyon algoritmaları geliştirmiştir. İki seviyeli sadeleştirme, çok seviyeli yapılandırma ve teknoloji haritalandırması için etkili, optimal algoritmalar sunmuştur. Bu teknikte tanıtılan algoritmalar MIS (Çok seviyeli lojik optimizasyon sistemi, A Multiple-Level Logic Optimization System) adı verilen programda uygulanmıştır. Bu algoritmaların gerçek tasarım problemlerinde uygulanabilirliğini göstermek için tez içerisinde karmaşık dijital devre tasarımı yapmıştır.

Mailhot F. (1994), dijital devre tasarımında kullanılabilecek, iki lojik fonksiyon arasında denklik kurma algoritmaları geliştirmiş, devreler arasında gecikme optimizasyonunu ayarlayan tekniklere değinmiştir. İki lojik fonksiyon arasında denklik kuran algoritmalar ikili karar diyagramları kullanmaktadır. Bu tez çalışmasında lojik sentez ve fiziksel tasarım arasında köprü görevi gören teknoloji haritalandırması üzerinde yoğunlaşılmıştır. Geliştirilen algoritmaların mevcut diğer teknoloji haritalandırması

sistemleriyle karşılaştırıldığında çalışma zamanı ve sonuç kalitesi açısından etkili olduğu kanıtlanmıştır.

Oliver Coudert (1994), iki seviyeli Lojik sadeleştirmede Quine Mc Cluskey yöntemini tanıtmış, bu yöntemin sınırlılıklarını azaltarak, yeni bir algoritma geliştirmiştir. Bu algoritma Boolean fonksiyonlarını etkin graf yöntemiyle ifade ettiği için asal implikantların sayısıyla sınırlandırılmamıştır. Ayrıca bu algoritma aynı çalışma alanı içerisinde örtü problemini çözmekte ve alt-üst bağlantı (subset) sıralamasını tutmaktadır. Bu algoritma alt limit sınırlayıcı teoremi kullandığı için rekürsif döngü sayısını da önemli oranda düşürmektedir. Araştırmacı, bu algoritmanın (scherzo) yayınlandığı yılda bilinen bütün iyi algoritmalardan (expresso-exact, expresso-signiture) çok daha iyi sonuçlar verdiğini iddia etmektedir.

Steven M. Nowick ve David L. Dill (1995), Quine Mc Cluskey algoritmasını geliştirerek yeni bir algoritma tasarlamış ve sonuçlarını Espresso-exact algoritmasıyla karşılaştırmıştır. Tasarlanan algoritma, verilen fonksiyonu en iyi çarpımların toplamı şeklinde sadeleştirmektedir.

Wang Y. (1995), Boolean fonksiyonları için yeni bir gösterim yöntemi belirtmiş ve ikili karar ağaçları yardımıyla Boolean fonksiyonlarının sadeleştirilmesini anlatmıştır. Bu yöntem sayesinde daha verimli Boolean fonksiyon sadeleştirme algoritmaları geliştirilebileceğini belirtmiştir. Bu tez çalışmasında ayrıca bazı özel Boolean fonksiyonlarına ait sıralı ikili karar diyagramlarının karmaşıklık analizi ve graf yapıları incelenmiştir. Mevcut veri yapılarının karmaşıklığı üssel olarak artarken, geliştirilen bazı yeni veri yapılarının karmaşıklığı sabit olarak artmaktadır.

Fallah F. (1996), lojik devrelerin sadeleştirilmesi ve birleştirilmesini sağlayan bir algoritma önermiş, bu algoritmayı kullanan bir programı tanıtmıştır. Uygulama sonuçları SIS programına göre başarılı sonuçlar vermiştir.

Miller J. F. ve ark. (1996), yeni geliştirdikleri bir yöntemi Lojik ifadelerin sadeleştirilmesine uyarlamışlardır. Bu araştırmada geleneksel Boolean AND-OR ifadeleri yerine Reed-Muller mantığı olan AND-EXOR yapısı üzerinde çalışılmıştır. Araştırmacılar geliştirilen bu tekniği sıralı durum makineleri (sequential state machine) gerçekleştirmek için gereken mantık sadeleştirmelerinde kullanmışlardır.

Olivera A. L. ve ark. (1998) DC setlerini içeren ikili karar ağaçlarının sadeleştirilmesine değinmişlerdir. Geliştirilen algoritma sabit sıralı değişkenleri tam (exact) sadeleştirmiştir. Araştırmacılar bu problemin NP-Complete (Belirleyici Olmayan Polinom - Tam) olduğunu da ispat etmişlerdir. Araştırma minimum büyüklükteki ikili

karar diyagramlarının bulunmasının tam belirli olmayan (incompletely specified) sonlu durum makinelerinin azaltılması problemiyle çok benzer tarzda çözülebileceğini göstermiştir. Geliştirilen algoritma yardımıyla bir grup farklı örnek tam olarak sadeleştirilmiş, bulunan sonuçlar mevcut sezgisel algoritmalarla karşılaştırılmıştır.

Gröpl C. (1999), Boolean fonksiyonlarında ikili karar ağaç yapılarını incelemiş, sıralı ve serbest ikili karar ağaçlarının boyutları ve etkililiğine değinmiştir. Sıralı ikili karar diyagramlarında testler sabit değişken sırasına uymak zorundayken, serbest ikili karar diyagramlarında her değişkenin bir kez teste girmesinin önemli olduğunu belirtmiştir. Araştırmada ayrıca ikili karar diyagramları üzerinde silme, değiştirme ve birleştirme kurallarının etkileri ve ZBDD (Zero-Suppressed Binary Decision Diagrams), OKFDD (Ordered Kronecker Functional Decision Diagrams) gibi diğer ikili karar diyagramlarının özellikleri incelenmiştir.

Wang L. (2000), çok seviyeli lojik devrelerin sentezi ve optimizasyonunu sağlayan bir algoritma geliştirmiş, bu algoritmayı programlayarak sonuçları Espresso programıyla karşılaştırmıştır. Araştırmada fonksiyonel *DC* denilen yeni bir *DC* çeşidi tanıtılmıştır. Tek çıkışlı ve çok çıkışlı Boolean fonksiyonlarının yapılandırılmasında fonksiyonel dont care'leri oluşturan ayrı bir C programı geliştirilmiştir.

Chai L. (2000), *ON* mintermlerini kullanarak Özel-Veya Toplamlarının Çarpımı (ESOP) şeklinde verilen lojik fonksiyonları sadeleştiren bir program geliştirmiştir. Araştırmacı, birçok fonksiyon için ESOP ifadelerinin, normal SOP ifadelerinden daha az çarpım içerdiğini belirtmiş, minimum ESOP ifadelerini bulmak için sezgisel yöntemler kullanmıştır. Geliştirilen teknikler programlanarak örnekler üzerinde uygulanmış, sonuçları analiz edilmiştir.

Hlavicka J. ve Fiser P. (2000), doğruluk tablolarıyla tanımlanan fonksiyonları sadeleştirebilen yeni bir algoritma sunmuşlardır. Bu algoritmada sadece tanımlanmış çıktı değeri olan mintermler işleme alınmakta, böylece önemsiz (DC) durumlarını işleyen diğer algoritmalar gibi zaman kaybetmemektedir. Geliştirilen algoritma gücünü birkaç yüz değişken ve minterm içeren fonksiyonlar üzerinde kanıtlamıştır. Algoritmanın bilgisayar ortamında kodlanmış programı çalışma zamanının azlığı ve sonucun sadeliği açısından EXPRESSO algoritmasından daha iyi sonuçlar vermiştir.

Fiser P. ve Hlavicka J. (2001a), geliştirdikleri Boolean sadeleştirme programını (BOOM) inceleyen bir rapor yayınlamışlardır. Geliştirdikleri algoritmanın, kendilerinden önce geliştirilen bütün algoritmaların kullandığı tabandan yukarı mantığı yerine, yukarıdan aşağıya mantığıyla çalıştığını belirtmişlerdir. Bu sayede mintermlerin

içerisindeki literalleri eleyerek, implikantların varyasyonlarını arttıracağına, yeni literaller eklenerek mintermlerin varyasyonları yavaş yavaş azaltmaktadır. Orijinal Quine-McCluskey probleminde tabandan yukarı mantığıyla gidildiği için hangi mintermin önce örtüleceği sonucun sadeliğini etkilemektedir. Geliştirilen yöntemde örtü problemi final çözümünü sadece kullanılan literal sayısı açısından dolaylı olarak etkilemektedir.

Fiser P. ve Hlavicka J. (2001), daha önce geliştirmiş oldukları algoritmayı, literalleri sezgisel olarak seçerek geliştirmiş ve aradaki farkı değerlendirmişlerdir. Geliştirilen sezgisel yöntem, literallerin tekrar etme sıklığını takip ederek en uygun olabilecek literal sırasıyla eklemektedir. Bu veriyi doğrudan metot içinde kullanmak yerine, veri üzerinde bazı dönüşümler yapılmış ve bu dönüşümlerden elde edilen kazanç araştırma içerisinde değerlendirilmiştir.

Fiser P. ve Hlavicka J. (2001b), geliştirdikleri sezgisel sadeleştirme programını tanıtıp Espresso programıyla karşılaştırmışlardır. Geliştirilen sadeleştirme programı sadeleştirilecek fonksiyonun tanımını, çözüm için bir temel olarak kullanmaz. Bu algoritmada sadeleştirme prosedürü çok hızlıdır, eğer ilk bulunan sonuç ihtiyaçları karşılamazsa hemen geliştirilir ve tekrar kontrol edilir. Geliştirilen program çok farklı problemler üzerinde denenmiş ve sonuçları paylaşılmıştır.

Fiser P. (2002a), iki seviyeli Boolean sadeleştirme yöntemini tanıtmıştır. Bu yöntemde *ON* mintermlerini örten ve sonuç ifadesinde kesin olarak kullanılması gereken implikantlar önceden hesaplanmaktadır. Diğer algoritmalara göre, geliştirilen FC-Min algoritması, özellikle yüksek sayıda giriş-çıkış değişkeni olan fonksiyonların sadeleştirilmesinde etkilidir.

Fiser P. ve Hlavicka J. (2002b), daha önce geliştirdikleri "BOOM" algoritması üzerinde tek seviyeli ayrıştırma ve yeni bir Boolean sadeleştirmeyi tanıtmışlardır. Sadeleştirme giriş değişkenlerinin kullanılması için belirtilen çeşitli sınırlandırmalara uygun olarak çalışmaktadır. Bu sayede araştırmacılar giriş ve çıkış değişken sayıları özellikle belirtilen devreleri etkili bir şekilde sadeleştirmeyi başarmışlardır.

Kahramanlı Ş. ve Başçiftçi F. (2003), küp cebri işlemlerini kullanarak, yakın minimum örtme algoritması geliştirmiş ve Boolean fonksiyonlarını sadeleştirme algoritmalarının karmaşıklığını analiz etmiştir. Araştırmacılar önerilen yöntemin zaman karmaşıklığını ($O(n)$), Quine-McCluskey yönteminin zaman karmaşıklığıyla ($O(2^n)$) karşılaştırmışlardır.

Fiser P. ve Hlavicka J. (2003), "BOOM" algoritmasında implikant oluşturma paradigmasınına değinmişlerdir. Araştırmacılar kullanılan yöntemi detaylı olarak anlatmış ve kullandıkları algoritmanın kaba kodunu çalışmalarında vermişlerdir. Araştırmacılar standart MCNC benchmark devrelerinin girdi sayılarının göreceli olarak az olduğu için geliştirdikleri algoritmanın tam gücünü bu devreler üzerinde gösteremediğini, geliştirdikleri BOOM algoritmasının daha karmaşık problemlerde çok daha etkili olduğunu belirtmişlerdir.

Fiser P. ve ark. (2003), sezgisel, çok çıkışlı, iki seviyeli Boolean sadeleştirme algoritmasını tanıtmışlardır. Bu algoritma önce *ON* mintermlerinin örtüsünü bulmakta ve bunları kullanarak grup implikantlarını türetmektedir. Bu yöntemde sadece *ON* mintermleri örten gerekli implikantlar hesaplanmaktadır.

Uçar O. ve Dervişoğlu A.(Uçar ve Dervişoğlu, 2003), lojik sadeleştirmede genel örtü problemini çözen bir program geliştirerek ESPRESSO algoritmasıyla kıyaslamışlardır. İyileştirilmiş dallanma yöntemleri kullanan program, 1000x1000 mertebesindeki bir minimal problemini saniyeler içerisinde çözebilmektedir. Geliştirilen program birçok benchmark'ta en uygun, diğerlerinde de optimale çok yakın sonuçlar vermiştir.

Özcan U. ve Dervişoğlu A. (2003), asal implikantların bulunmasında ikili karar diyagramlarının kullanılmasına değinmişlerdir. Bu sayede asal implikantların bulunması için çok daha az bellek harcayarak, daha hızlı sonuca ulaşılabilmektedir. Araştırmacılar asal bileşen bulmada BDD kullanılmasının örtü probleminin çözümü için yeni seçenekler sunduğunu belirtmişlerdir.

Fiser P. ve Kubatova H. (2004b), genel örtü bulma problemine değinmişlerdir. Araştırmacılar geliştirdikleri algoritmanın örtü bulurken diğer Boolean sadeleştiricilere göre ters mantıkla çalıştığını vurgulamışlardır. Genel örtü probleminin NP-Hard (NP-Zor) olduğunu belirten araştırmacılar, problemin çözümü için bazı sezgisel yöntemleri kullanmış, bu yöntemlerin sonuç ifadesi üzerindeki kalite ve çalışma zamanı açısından etkilerini incelemişlerdir.

Färm P. (2004), "Advanced algorithms for Lojik synthesis" isimli tez çalışmasında sadeleştirme algoritmalarını inceleyerek kural tabanlı Lojik sadeleştirme tekniğini tanımlamıştır. Geleneksel lojik sentezde olduğu gibi sadece literal sayısına bakarak sadeleştirmenin kalitesini değerlendirmenin her zaman doğru olmadığını belirten araştırmacı, sonuç ifadesini değerlendirirken gecikme, enerji tüketimi gibi değerleri de göz önünde bulundurmak gerektiğini belirtmektedir. Araştırmacı, cebirsel sadeleştirme

gibi diğer yöntemlerin bazı uygulama kısıtlamalarını ortadan kaldırdığını fakat onların da daha zayıf sadeleştirme gücü olduğunu vurgulamaktadır.

Fiser P. ve Kubatova H. (2004a), önceden geliştirdikleri iki yöntemi birleştirerek BOOM-2 adını verdikleri yeni bir iki seviyeli lojik sadeleştirme yönteminden bahsetmişlerdir. Geliştirilen araç çalışma zamanı ve sonuç kalitesi açısından başarılı sonuçlar vermiştir. Bu araç çok büyük miktardaki giriş ve çıkış değişkenlerine sahip fonksiyonlara da uygulanabilmektedir.

Verma S. ve Permar K. D. (2004), Boolean fonksiyonlarının sadeleştirilmesinde gri kodlamanın kullanılması ve paralel algoritmanın geliştirilmesine değinmişlerdir. Araştırmacılar verilen mintermlerin arasındaki ilişkiyi tespit etmek için gri kodun uzaklık ve yansıma özelliklerini kullanmaktadır. Araştırmacılar verilen Boolean fonksiyonunun alt düzey sadeleştirilmesi ve asal implikantların oluşturulması için parallel bir algoritma geliştirmişlerdir. Önerilen kâğıt-kalem yöntemi ve parallel algoritma, *DC* içermeyen birkaç örnek üzerinde denenmiştir.

Ghasemzadeh M. (2005) (2005), ikili karar diyagramlarını kullanarak, tamamen belirlenmiş Boolean formüllerini değerlendiren bir algoritma sunmuştur. Araştırmada farklı ikili karar diyagramlarının olumlu ve olumsuz yönleri araştırılmış, geliştirilen algoritma C programlama dilinde Colorado üniversitesinin ikili diyagram paketi kullanılarak gerçekleşmiştir. Araştırma bulguları geliştirilen programın mevcut programlardan daha hızlı çalıştığını göstermektedir.

Başçiftçi F. (2006), lojik fonksiyonların sadeleştirilmesinde kullanılan algoritmaları incelemiş, yakın sadeleştirme yapan bir algoritma geliştirmiş, bu algoritmayı programlayarak elde edilen sonuçları ESPRESSO programıyla karşılaştırmıştır. Geliştirilen algoritma doğrudan örtme prensibine dayanmaktadır. Mevcut yöntemlerden farklı olarak bu yöntemde *ON* küpü içeren asal implikantlar, *ON* küpleri ile değil, *OFF* küpleri yardımıyla genişletilirler. Bu sayede algoritmanın karmaşıklığı önemli ölçüde azalmaktadır. Araştırma bulgularına göre geliştirilen algoritma, ESPRESSO algoritmasıyla kıyaslandığında genel olarak çok daha hızlı çalışmakta ve daha az hafıza kullanmaktadır.

Savran İ. (2006), lojik fonksiyonların minimuma yakın sadeleştirilmesinde kullanılan bir algoritmaya değinmiştir.

Kahramanlı Ş. ve ark. (2006), doğrudan örtme prensibiyle, *OFF* küplerine dayalı tek çıkışlı lojik sadeleştirme algoritması geliştirilmiş, bu algoritmanın karmaşıklığı analiz edilmiştir. Araştırmacılar verilen *ON* küpünü içeren asal implikantların belirlenmesi için,

bu küpü her seferinde bir koordinata genişletmişlerdir. Her bir genişlemenin doğruluğu genişletilen küpün, bütün *OFF* küpleriyle kesiştirilmesiyle kontrol edilmiştir. Geliştirilen algoritmanın karmaşıklığı diğer iki seviyeli sadeleştirme algoritmalarına göre daha az olmakla birlikte, sadece bazı fonksiyonlarda (*OFF* mintermlerinin sayısının *ON* mintermlerinin sayısından fazla olduğu fonksiyonlar) göreceli olarak biraz yavaş olduğu, diğer durumlarda daha doğru sonuca daha kısa zamanda ulaştığı söylenebilir.

Yılmaz B. (2007), anahtarlama fonksiyonlarının sadeleştirilmesine değinerek, *OFF* kümelerini küp cebri işlemleri kullanarak sadeleştiren bir algoritmayı tanıtmıştır. Bu algoritmada küpler genişletilirken, bütün koordinatlara aynı anda paralel olarak genişletilmiş, çalışma zamanında önemli bir kazanç sağlanmıştır. Geliştirilen algoritma C++ dilinde programlanmış, 46 adet çok girişli, tek çıkışlı fonksiyon üzerinde denenmiştir. Araştırma sonuçları ESPRESSO algoritmasıyla karşılaştırılmıştır.

Fiser P. ve ark. (2008), fazla sayıda *ON* mintermi içeren fonksiyonlar için basit ve hızlı iki seviyeli sadeleştirme algoritmasına değinmişlerdir. Geliştirilen algoritma "ternary tree" temeline dayanmaktadır ve *ON* mintermleri üzerinden gitmektedir. Çok büyük fonksiyonları çok kısa zamanda işleyebilme kapasitesine sahip olan algoritmanın sonuç kalitesi diğer algoritmalar kadar iyi değildir. Bu algoritma fonksiyonları sadeleştirmek için bir ön işlem olarak uygulanabilir. Bu işlem sayesinde daha sade hale gelen fonksiyon diğer sadeleştirme programlarıyla (EXPRESSO gibi) çok daha hızlı bir şekilde sadeleştirilebilmektedir. Böylece sonuç kalitesinde bir değişiklik olmazken, çalışma zamanı ve gereken bellek miktarı önemli ölçüde düşmektedir.

Feldman V. (2009), iki seviyeli Lojik sadeleştirmenin karmaşıklığını ve zorluk derecesini analiz etmiştir. Quine'nin bulduğu iki seviyeli lojik sadeleştirmenin Masek tarafından NP-Tam bir problem olduğuna değinen araştırmacı Boolean sadeleştirme ile ilgili farklı problemlerin zorluk derecelerini ispat etmiştir.

Lemberski I. ve Fiser P. (2009b), iki seviyeli asenkron lojik sentez metodunu tanıtmışlardır. Bu yöntem sınırlı veya sınırsız giriş zaman gecikmesine göre çalışabilmektedir. Sınırlı zaman gecikmesinde en kötü durum gecikmesine göre davranmakta, sınırsız zaman gecikmesinde girdi değerlerindeki değişimi tespit etmektedir. Bu yöntemde önce bütün girişler ve çıkışlar sıfırlanmakta, bu durum tespit edildikten sonra yeni giriş değerleri uygulanmaktadır.

Lemberski I. ve Fiser P. (2009a), çok seviyeli asenkron lojik sentez metodunu tanıtmışlardır. Bu yöntem sadeleşmiş AND-OR düğümlerinden oluşan çok seviyeli mantık ağından oluşmaktadır. Araştırmacıların daha önce sundukları iki seviyeli lojik

sentez algoritmasına dayanmaktadır. Araştırmacılar önerilen metodun uygulama karmaşıklığını EXPRESSO programıyla karşılaştırmışlardır.

El-Bakry H. M. ve Mastorakis N. (2009), Boolean fonksiyonlarının sadeleştirilmesinde kullanılan yeni bir yöntemi tanıtmışlardır. Bu yöntem karnaugh haritası ya da tablolama yöntemi gibi görsel bir tasarıma gerek duymayıp, kolayca programlanabilmektedir. Araştırmacılar geliştirilen yöntemi matlab programı yardımıyla denemişler, yaklaşık çalışma zamanını araştırma sonunda paylaşmışlardır.

Fiser P. ve Toman D. (2009), çok terimli (milyona kadar) fonksiyonları sadeleştirebilen hızlı ve etkili bir yöntemi tanıtmışlardır. ESPRESSO programından önce bu algoritmanın çalıştırılması sonuç kalitesini etkilemeden zaman tasarrufu sağladığını belirtmişlerdir.

Başçiftçi F. ve Kahramanlı Ş. (2010a), anahtarlama fonksiyonları için yakın minimum sadeleştirme algoritması geliştirilmiş, algoritmanın uygulama sonuçları Espresso algoritmasıyla karşılaştırılmıştır. Araştırmada geliştirilen algoritma C programlama dilinde kodlanmış, delphi dilinde görsel arayüzü yazılmıştır. Geliştirilen algoritmanın daha hızlı çalıştığı ve daha az bellek harcadığı tespit edilmiştir.

Başçiftçi F. ve Kahramanlı Ş. (2010b), tek çıkışlı lojik fonksiyonlar için *OFF* küplerine dayalı yakın minimum sadeleştirme algoritması tanıtılmış, asal implikantların belirlenmesine değinilmiştir. Araştırmada *ON* küplerine dayalı asal implikantların elde edilebilmesi için *OFF* küpleri kullanılmıştır. Geliştirilen algoritmada daha hızlı hesaplama yapabilmek için standart işlemler yerine, lojik işlemler kullanılmıştır.

Toman D. ve Fiser P. (2010), üçlü ağaç yapılarını (ternary tree) kullanarak çok terimli lojik fonksiyonları sadeleştirmeyi tanıtmışlardır. Bu algoritma milyona kadar çarpım ifadeleri içeren fonksiyonları işleyebilmektedir. Sadeleştirme işlemi, üçlü ağaç yapıları üzerinde temel Boolean işlemlerinin hızlı bir şekilde uygulanmasına dayanmaktadır. Bu uygulamada kısmen belirli fonksiyonların sadeleştirilmesi de desteklenmektedir.

Brown F. M.(Markham Brown, 2010), Quine'nin geliştirdiği iki seviyeli Lojik sadeleştirme yönteminin çok daha önce (1878) Hugh McColl tarafından farklı bir formatta tanımlandığına değinmiştir. Araştırmacı, McColl'un çoklu integrallerin limitleri ile ilgili problemler üzerinde çalışırken Quine tarafından geliştirilen iki seviyeli sadeleştirme yönteminin bir değişik halini kullandığını tespit etmiştir.

Başçiftçi F. ve Kahramanlı Ş. (2011), çarpımların toplamı şeklinde ifade edilmiş, tek çıkışlı Boolean fonksiyonları için *OFF* kümesi tabanlı doğrudan örten tam

sadeleştirme algoritması geliştirmişlerdir. Araştırmacılar verilen *ON* mintermlerini kapsayan asal implikantları elde etmek için *OFF* mintermlerini kullanmışlardır. Aritmetik operatörler yerine lojik operatörler kullanılarak hesaplama zamanında önemli bir kazanç sağlanmıştır. Geliştirilen algoritma ESPRESSO programından daha hızlı çalışarak daha doğru sonuçlar vermiştir.

Kahramanlı Ş. ve ark. (2011), Boolean fonksiyon yaklaşımını özellik seçmede kullanmışlardır. Böylece geleneksel fark edilebilirlik matrixi kullanan programlarla işlenemeyen veri setlerinden birçoğunu hafıza kullanımını arttırmadan işleyebilmektedir.

Nosrati M. ve Hariri M. (2011), graf veri yapılarını kullanarak Boolean fonksiyonlarını sadeleştiren sezgisel bir algoritmayı tanıtmıştır. Önerilen algoritma için graf veri yapıları, çarpımların toplamı şeklindeki ifadeler üzerinde kullanılmıştır. Araştırmada ayrıca maximum sadeleştirmeye ulaşabilmek için bazı önerilerde bulunulmuştur.

Toman D. ve Fiser P. (2011), üçlü ağaç yapıları üzerinde Boolean işlemleri yapan algoritmayı, ESPRESSO'da kullanılan algoritmayla birleştirerek yeni bir lojik sentez yöntemine değinmişlerdir. Geliştirilen algoritma milyonlarca çarpım ifadesini işleyebilmektedir. Araştırma sonunda bulunan algoritmanın performansı EXPRESSO programıyla karşılaştırılmıştır.

Başçiftçi F. ve ark. (2012), azaltılmış (indirgenmiş) *OFF* küplerine dayalı asal implikant belirleme yöntemine değinmişlerdir. Araştırmacılar lojik sadeleştirmede kullanılan küpleri hangi literalleri kaldırarak genişleteceklerini tespit etmek için küplerin pozisyonlarını tutan string kullanmayı önermektedir. Böylece sonuç kalitesini önemli ölçüde arttırmayı ve bellek karmaşıklığını 2 kat, zaman karmaşıklığını 3,5 kat azaltmayı başarmışlardır.

Bernasconi A. ve ark.(Bernasconi ve ark., 2009b), lojik sadeleştirmede *DC* mintermlerine birer ağırlık değeri atayarak, sonuç fonksiyonunda faydası olabilecek *DC* mintermlerinin kapsanmasını sağlamışlardır. Araştırmada daha önce Fiser tarafından geliştirilen BOOM algoritması kullanılmıştır. Bu algoritmaya ek olarak *DC* terimlerine ağırlık değeri atayarak daha fazla *DC* teriminin kapsanması amaçlanmıştır. Araştırma bulgularına göre geliştirilen algoritma, normal BOOM algoritmasına göre ortalama %66 daha fazla *DC* mintermini kapsamaktadır.

Sampson M. ve ark. (Sampson ve ark., 2012) , eksik belirtilen Boolean fonksiyonları için tam ESOP ifadelerinin bulunmasına yarayan bir yöntem sunmuşlardır. Bu yöntem 6 giriş değişkenine kadar fonksiyonları desteklemektedir. Araştırmacılar, 5

değişkenli fonksiyonların ağırlıklarını sıkıştırılmış bir tabloda listeleyerek hesaplama zamanını önemli oranda düşürmüşlerdir. Araştırmacılar çalışmalarının eksik tanımlanan fonksiyonların, kesin çözüm kümesini bulan ilk araştırma olduğunu düşünmektedirler.

Čepek O. ve ark. (2012), verilen bir *f* Boolean fonksiyonunun CNF gösterimi ve küme gösterimi arasındaki ilişkiyi incelemişlerdir. Araştırmacılar kapsanabilen bütün fonksiyonlar için en küçük CNF boyutunda polinomsal doğrulanabilirlik belgesi olduğunu kanıtlamışlardır. Ayrıca bütün fonksiyonların kapsanabilir olmadığını belirtmişler, kapsanabilir olmayan fonksiyon örnekleri oluşturmuşlardır.

Li X. ve ark. (Li ve ark., 2013) , kriptolojide Boolean fonksiyonları tasarlarken aynı zamanda hem dayanıklı, hem de dengeli lojik fonksiyonların oluşturulmasının zor olduğunu belirtmişlerdir. Araştırmacılar her iki durumu da sağlayan *n* değişkenli, dengeli, simetrik Boolean fonksiyonları oluşturmuşlardır.

Čepek O. ve ark. (2013), Boolean fonksiyon sınıfları hiyerarşisini tanıtmışlardır. Sınıflarda bulunan fonksiyonların asal implikantları en az belirli bir uzunlukta olmalıdır. Araştırmacılar verilen bir DNF ve hiyerarşi sınıfının polinomsal zaman içerisinde test edilebileceğini göstermişlerdir. Araştırmacılar ayrıca verilen bir DNF giriş fonksiyonunu, en kısa DNF çıkış fonksiyonuna çeviren polinomsal zamanlı bir algoritma sunmuşlardır. Araştırmada tanıtılan bu sınıf, Boolean sadeleştirme probleminin polinomsal zamanda çözülebilirliği sınıfının yeni üyesidir. Araştırmacılar hiyerarşide bulunan diğer sınıflar için Boolean sadeleştirme probleminin sabit bir katsayı ile tahmin edilebilir olduğunu göstermişlerdir.

Chikalov I. ve ark.(2013), en fazla 5 değişkenli Boolean fonksiyonlarının hesaplanmasında ikili karar ağaçlarının zaman(ağaç derinliği) ve hafıza(bağlantı sayısı) karmaşıklığı arasındaki ilişkiyi incelemiştir. Araştırmacılar en fazla 5 değişkene sahip Boolean fonksiyonlarının her birisi için, derinlik ve bağlantı sayısı açısından ideal bir karar ağacı olduğunu göstermiştir.

Creignou N. ve Daudé H. (2013), Boolean fonksiyonların hassasiyetini araştırmışlardır. Belirli giriş değerlerine sahip bir Boolean fonksiyonundaki hassasiyet seti, değişmesi halinde çıkış fonksiyonunu değiştirecek giriş değerlerinden oluşmaktadır. Araştırmacılar bir Boolean fonksiyonunun hassasiyet setindeki eleman sayısının sadece giriş değerlerinin önem düzeyine göre değiştiğini göstermişlerdir.

Lemberski I. ve ark. (2014), tek yönlü, senkron lojik ifadeleri, iki yönlü asenkron lojik ifadeler haline çevirip sadeleştirme yapmış ve sıradan yöntemle karşılaştırıp karmaşıklık analizini sunmuştur. Araştırmada ulaşılabilir ve ulaşılamaz *DC* durumlarını

içeren kısmen belirli fonksiyon setleri tanıtılmıştır. Araştırma bulguları asenkron sadeleştirmenin ortalama %20 daha hızlı sadeleştirdiğini göstermiştir.

Mizuki T. ve ark. (2014), iki değişkenli, çok değerli giriş fonksiyonlarının sadeleştirilmesinde ESOP algoritmalarının kullanıldığını fakat "karmaşık terimlerin özel-veya-toplamı" (ESCT) algoritmalarının kullanılmadığını tespit etmişlerdir. Araştırmacılar iki değişken, çok değerli giriş fonksiyonlarının sadeleştirilmesi için etkili bir sadeleştirme algoritması geliştirmiş, çıkış ifade sayısının diğer yönteme göre en az bir tane indirgenebileceğini göstermişlerdir.

Movsisyan Y. M. ve Aslanyan V. A. (2014), De Morgan fonksiyonları için DNF ve CNF gösterimlerini tanımlamışlar ve bu fonksiyonlar için fonksiyonel tamlık teoremini ispatlamıştır.

Roy S. ve Bhunia C. T. (2014), çok değişkenli giriş fonksiyonundan iki değişkenli giriş fonksiyonuna indirgeme algoritmasını sunmuştur. Araştırmada daha sade devrelerin maliyetinin düşük olduğu, ısınma sorununun daha az olduğu ve daha hızlı çalıştıkları belirtilmiştir. Geliştirilen algoritma yüksek sayıda giriş değişkenine sahip problemler için de uygulanabilir.

Roy S. ve Bhunia C. T. (2015), çok girişli lojik devrelerin iki seviyeli sadeleştirilmesinde sistematik bir yaklaşım önermişlerdir. Araştırmada dijital anahtarlama devrelerinin gösteriminde ve sadeleştirilmesinde onaltılık(hexadecimal) kodlama kullanılmıştır. Geliştirilen yöntemin yazılım uygulaması tablolama tekniğine dayanmaktadır. Yeni yaklaşım sayesinde her gruptaki 1'leri kontrol etme ihtiyacı önemli oranda azalmıştır. Sunulan teknik hem manuel hesaplama hem de bilgisayar uygulamasında kullanılabilmektedir.

Roy S. ve Bhunia C. T. (2015), lojik anahtarlama devrelerinin tarihsel gelişimini araştırmışlardır. Araştırmacılar bilgisayar uygulamaları ve elektronik alanında lojik devrelerin kullanımı ve kronolojik gelişimi incelenmiştir. Ayrıca araştırmada lojik fonksiyonların gösterilmesi ve sadeleştirilmesi detaylı olarak irdelenmiştir.

Zulehner A. ve Wille R. (Zulehner ve Wille, 2017) , kuantum çoklu değerli karar diyagramları (Quantum Multiple-Valued Decision Diagrams – QMDD) temelli yeni bir sadeleştirme yaklaşımını tanıtmışlardır. Araştırmacılar karar diyagramlarındaki yolları ve düğümleri birlikte değerlendirerek kuantum giderlerini önemli oranda düşürmüşlerdir. Deney sonuçlarına göre çalışma zamanını üç kat, kuantum giderlerini 6 kat oranında azaltmışlardır.

Amaru L. ve ark. (2017b), endüstride uygulanabilecek yeni bir lojik sentez tekniği sunmuşlardır. Araştırmacılar sunulan lojik sentez tekniğinin sonuç kalitesini arttırdığını ve yazmaç transfer seviyesi (register transfer level, RTL) sentezi ile fiziksel uygulama arasında daha doğru bir ilişki sağladığını belirtmiştir. Araştırmada yeni sentez tekniğinin nano aygıtlarda uygulanması tartışılmıştır.

Amaru L. ve ark. (2017a), çok seviyeli lojik devrelerin kesin indirgenmesini sağlayan ölçeklenebilir bir metot sunmuştur. Araştırmada çok seviyeli lojik devrelerin indirgenmesindeki doğruluk incelenmiştir. Sunulan yöntem, çizge teorisinden ve ortak destek ayrışmasından (Joint Support Decomposition, JSD) kavramlar içermektedir. Araştırma sonuçları lojik sentez alanında yeni araştırmalara çok ihtiyaç olduğunu göstermiştir.

Roy S. (2017), mintermleri oktal kodlayarak anahtarlama fonksiyonlarının sadeleştirmesinde yeni bir yaklaşım sunmuştur. Araştırmada giriş değişkenleri üçerli(oktal) olarak gruplanmıştır. Birleştirilebilecek komşu mintermlerin tespiti için oktal karşılaştırma tabloları kullanılmıştır. Geliştirilen yöntem diğer yöntemlere göre algoritma adımlarını önemli ölçüde azaltmıştır.

Bernasconi ve ark. (2018), en küçük boyutlu devrelerin sentezi için lojik fonksiyonların ayrışmasına dayanan iki yeni sistem önermişlerdir. Araştırma sonuçlarına göre önerilen yöntemlerden birisi benchmark dosyalarının %34'ünde en az %25 alan kazanımı sağlamış, diğer yöntem ise benchmark dosyalarının %27'sinde en az %22 alan kazanımı sağlamıştır.

Aksoy L. ve Altun M. (2019), anahtarlama devrelerinin lojik sentezi için gerçeklemeye dayalı yakın minimum sadeleştirme algoritması geliştirmiştir. Bu araştırma ayrıca çoklu lojik fonksiyonların gerçekleşmesi için etkili bir yöntem önermektedir. Araştırma sonuçlarına göre, geliştirilen yakın minimum sadeleştirme algoritması diğer yakın minimum sadeleştirme algoritmalarına göre daha iyi sonuçlar vermiştir.

Lee S ve ark. (2019), 3 değerli lojik devrelerin sentezine yönelik bir yöntem önermiş ve uygulamışlardır. Araştırmacılar geliştirilmiş Quine-McCluskey algoritması ile transistör sayısını minimize ederek, 3 değerli lojik devreleri optimize etmişlerdir. Önerilen yöntem devrelerin güç-gecikme çarpımını %52 ve %68 oranında geliştirmiş ve bu devrelerde ortalama %42 daha az sayıda transistör kullanmıştır.

Abd-El-Barr M. ve ark. (2020), çok değerli lojik devrelerin sentezinde doğrudan örtme algoritmaları ile sürü zekâsının karşılaştırılmasını araştırmışlardır. Araştırmada rastgele oluşturulmuş 2 değişkenli, 4 değerli fonksiyon dosyalarını kullanmışlardır.

Simulasyon sonuçlarına göre, parçacık sürüsü optimizasyonu, karınca kolonisi optimizasyonuna göre daha iyi sonuçlar vermiştir. Ayrıca sürü optimizasyonu sonuçları Espresso-MV sonuçlarından ortalama çarpım terimleri sayısı açısından daha iyi çıkmıştır.

Souza L. ve ark. (2020), kombinasyonel mantık devrelerinin tasarımında kullanılan algoritmaların analizine yönelik benchmark dosya seti geliştirmişlerdir. Araştırma sonuçları geliştirilen benchmark dosya setinin algoritmaların amaçlı sezgisel yöntemlerini analiz etmede çok başarılı olduğunu göstermiştir.

2.2. Fonksiyon Sadeleştirme Uygulamaları Kaynak Araştırması

Li P. P. (1989), çok işlemcili, message passing memory sistemlerinde çalışan lojik programlamada paralel işleme modelini araştırmıştır. Geliştirilen model veri işlemenin paralel programlamaya uyarlanmasını "AND" ve "OR" üzerinden yapmaktadır. AND kapısı paralelleştirmesi verilen lojik ifadenin literallerinin değişken akış grafiği oluşturularak uygulanmaktadır. Bu bir sıralama algoritmasıdır. OR paralelleştirmesi parçalı çözüm ifadelerine özel senkronizasyon sinyallerinin eklenmesiyle yapılmaktadır. Bu bir birleştirme algoritmasıdır.

Lu P. (1990), Boolean yöntemlerini kullanan kural tabanlı bir uzman sistem araştırmıştır. Bu sistem bazı kural setlerini minimum Boolean formlarına çevirmek için kullanılan konsensus tabanlı algoritmadan oluşmaktadır. Bu amaçla lineer olmayan etkili bir binary programlama algoritması sunulmuştur. Araştırma kapsamında Boolean işlemlerine dayanan komşuluk ilişkisi araştırma temelli tasarım optimizasyon yöntemi de geliştirilmiştir. Araştırma sonuçlarına göre bu algoritmaların sistemleri sadeleştirme ve hızlandırmada önemli bir potansiyeli bulunmaktadır.

Fiser P. ve Hlavicka J. (2001b), iki seviyeli Lojik sentez için sezgisel sadeleştirme metodunu tanıtmışlardır. Geliştirilen yöntemin lojik tasarım, yapay zeka, yazılım mühendisliği ve graf teorisi gibi alanlardaki problemlere uygulanabilir olduğunu belirtmişlerdir.

Yang C. ve Ciesielski M. (Congguang ve Ciesielski, 2002) , ikili karar diyagramlarını kullanarak Lojik ifadeleri sadeleştiren bir yöntemi tanıtmışlardır. Tanıtılan yöntem AND, OR, XOR, complex MUX gibi aritmetiksel ve Boolean bütün yapılandırma yapılarını desteklemektedir. Bu yöntem çok büyük devreleri işleyebilme kapasitesine sahiptir. Araştırma sonuçları ikili karar diyagramlarının mevcut lojik sadeleştirme yaklaşımlarına iyi bir alternatif olabileceğini göstermiştir. Bu yöntem ayrıca,

diğer geleneksel lojik sadeleştirme algoritmalarına göre çok önemli çalışma zaman avantajı sunmaktadır.

Hlavicka J. ve Fiser P. (2002), daha önce geliştirdikleri "BOOM" algoritması üzerinde tek seviyeli ayrıştırmanın etkisine değinmişlerdir. Sadeleştirme, giriş ve çıkış sayıları için belirtilen sınırlandırmaya ve giriş için yük dengeleme hedefine orantılı/uygun olarak gerçekleştirilmektedir. Geliştirilen yöntem sayesinde bine kadar giriş sayısına sahip çok büyük fonksiyonları çok kısa zamanda işleyebilmektedir. Yöntem özellikle belirsiz (DC) sayısının çok olduğu fonksiyonlar için çok daha etkili çalışmaktadır.

Tomaszewski S. P. ve ark.(Tomaszewski ve ark., 2003), Quine-McCluskey algoritmasını geliştirerek online java applet halini internette yayımlamışlardır. Hazırlanan program 4 değişkene kadar çalışabilmektedir. Araştırmacılar programın bilgisayar ve elektronik mühendisliği bölümünde okuyan öğrenciler ve bu bölümlerde ders veren profesörler için faydalı olabileceğini belirtmişlerdir.

Boyd M. J. (2005), çok büyük problemleri işleyebilen, büyük ölçekli bir işlemciyi tanıtmış, önce geliştirilmiş etkisiz alan programlama yöntemlerinden kaçınmış, büyük boyutlu parallelleştirmeyi kullanmaya çalışmıştır. Araştırmacı ayrıca tamamen paralel Boolean devre uygulamalarının karmaşıklığını analiz etmiştir.

Oral S. O. (2005), çoklu değerli mantık fonksiyonlarını kendisini oluşturan daha küçük alt fonksiyonlara bölerek fonksiyon sentezinin daha hızlı ve etkin yapılmasına değinmiştir. Araştırmada "Boolean ayrık hesaplama metodu" çoklu değerli mantık fonksiyonlarına uyarlanmıştır. Araştırmada ihtiyaç duyulduğu için Karar Çizenekleri ve İşleçleri oluşturma algoritması da geliştirilmiştir.

Fiser P. ve Kubatova H. (2006), esnek sadeleştirme algoritmasını tanıtmış ve uygulamalarından bahsetmiştir. Geliştirilen sadeleştirme algoritması, kullanıcı tarafından tanımlanan çeşitli kısıtlamalara uygulanabilecek şekilde tasarlanmıştır. Böylece kullanıcı düşük güç tasarımı, test edilebilir tasarım, modüler yapı gibi farklı durumlara uygun sadeleştirme yapabilmektedir.

Franzle M. ve ark. (2007), karmaşık Boolean yapıya sahip aritmetik sistemlerin etkili çözüm yöntemini incelemişlerdir. Araştırmacılar önerdikleri yöntemle binlerce değişkene sahip, binlerce literale sahip aşırı karmaşık Boolean yapılarını işleyebilmektedir. Bu araştırma Boolean sadeleştirmenin farklı alanlarda kullanımına güzel bir örnek teşkil etmektedir.

Cobb J. (2007), bilgisayar ağlarının çok noktalı optimizasyonunda Boolean fonksiyonlarının kullanımını incelemiş, Boolean fonksiyonlarını kullanarak ağları

sadeleştiren bir algoritma sunmuştur. Bu algoritmada aynı anda sadece iki nokta optimize edilmektedir. Sadeleştirilecek noktaların seçilmesi için akıllı sezgisel yöntemler kullanılmıştır. Geliştirilen algoritmada pencere temelli teknik kullanılmıştır. Bu tezde sunulan algoritma diğer tekniklere göre daha kısa sürede, daha az hafıza kullanarak ortalama %12 daha az literal oluşturmaktadır.

Perinkulam A.S. (2007), grafik işlemcide çalışan lojik simülatör geliştirerek aynı programın normal işlemcide çalışan versiyonuyla karşılaştırmıştır. Geliştirilen algoritma paralel olarak programlanmıştır ve Grafik kartı işlemcisi üzerinde paralel olarak çalışmaktadır.

Fiser P. ve Toman D. (2008), iki veya daha fazla fonksiyon arasında Boolean işlemleri yapabilen bir programı tanıtmışlardır. Hazırlanan program giriş fonksiyonlarını doğruluk tablosu, CNF formu ya da aritmetiksel ifade gibi çok farklı formlarda kabul etmektedir. Alınan giriş fonksiyonları üzerinde tersini alma, AND, OR, XOR gibi işlemler yapılabilmekte, CNF ve DNF fonksiyonlarını birbirlerine çevirebilmekte ve çok seviyeli Boolean ağlarını/fonksiyonlarını, iki seviyeli doğruluk tablosu şekline indirgeyebilmektedir. DNF formundaki bir fonksiyonun tersini alma gibi bazı Boolean işlemleri çok fazla zaman ve hafıza tüketebildiği için "Ternary tree" adı verilen özel bir yapı tanıtılmıştır. Geliştirilen yöntem fonksiyon sadeleştirme, SAT (Sağlanabilirlik Problemi) çözme, fonksiyon indirgeme gibi çeşitli problemler üzerinde diğer programlarla karşılaştırılmıştır.

Bernasconi A. ve ark. (2009a), lojik devre tasarımında alan küçülten yeniden yapılandırma tekniğini araştırmışlardır. Bu teknik yardımıyla sinyaller, en az anahtarlama işlemi, en yüksek performans, en küçük alan kullanarak iletilebilmektedir. Bu teknik Shannon tekniğine dayanmakta, orijinal mintermlerin arasındaki Hamming uzaklığını değiştiren fonksiyonları kullanmaktadır. Araştırma sonunda bulgular Shannon tekniği ile karşılaştırılmıştır.

Wu L. ve Qiu D. (2010), çok değerli mantık fonksiyonlarına dayalı bir otomata teorisi tanıtılmış, yeni tanımlanan bu dilin sadeleştirilmesine değinilmiştir. Araştırmada daha önce Qiu tarafından tanımlanan L-değerli sonlu otomata (L-değerli lojik) teorisi kullanılmıştır. Araştırmada sonlu uzunlukta bir kelimenin olup olmamasına bakarak dilin Mealy tipi bir dil olup olmadığının tespit edilebildiği ispat edilmiştir. Araştırmada L-değerli sonlu otomata tarafından tanınan L-değerli diller ve L-değerli düzenli diller tanımlanmıştır.

El-Bakry H. M. ve ark. (2010), modüler sinir ağları kullanarak giriş fonksiyonlarını homojen şekilde bölerek daha önce geliştirdikleri algoritmayı işlem zamanı ve hafıza ihtiyacı açısından geliştirmişlerdir. Araştırmada bu yaklaşımın *XOR* fonksiyonlarına uygulanmasına yer verilmiştir. Daha önce sunulan modüler olmayan tasarımlarla kıyaslandığında hesaplama sıralamasında ve donanım ihtiyacında ciddi oranda bir düşüş yakalanmıştır.

Çini U. (2010), çok değerli mantık devrelerinin tasarımlarını incelemiş ve yeni nesil yeniden ayarlanabilen sistemlere değinmiştir. Araştırmada çok değerli mantık uygulamaları ile birlikte işaretli sayılar ve yedekli sayı sistemleri de analiz edilmiştir. Araştırmacı çok değerli mantık devreleri kullanarak, alternatif akım modlu yapılar oluşturmuştur. Sunulan yapılar etkili çarpma-toplama işlemleri ve sonlu dürtülü filtreleme yapabilmektedir.

Başçiftçi F. ve Hatay Ö. F. (2011), uzman sistem kullanarak diyabetin 10 temel belirtisi dahil bütün ihtimallerini(2^{10}) inceleyerek doğruluk tablosu oluşturmuşlardır. Doğruluk tablosunu lojik fonksiyonların sadeleştirilmesi yöntemiyle sadeleştirerek 15 temel kurala indirgemiş ve kural tabanını oluşturmuştur. Geliştirilen sistem 768 hastanın verilerini incelemiş, ortalama %97 oranında doğru tahmin etmiştir.

Zolfaghari B. ve Sheidaeian H. (2011), lojik fonksiyon sadeleştirmeyi kullanarak, görüntü sıkıştırma yöntemi geliştirmiş, diğer sıkıştırma yöntemleriyle sıkıştırma oranlarını karşılaştırmıştır. Araştırmacılar lojik fonksiyon sadeleştirme yöntemiyle 24 bitlik renkli görüntüyü sıkıştırmışlardır. Araştırma sonuçlarına göre geliştirilen algoritma, diğer algoritmalardan ortalama %25 daha iyi bir sıkıştırma oranına ulaşmıştır.

Hacıbeyoğlu M. ve ark. (2011), Boolean geri dağıtım kuralını kullanarak, lojik tabanlı parametre azaltma yöntemini geliştirmiş, yöntemin karşılaştırmalı olarak karmaşıklık analizine değinmişlerdir. Veri madenciliğinde DF (Discernibility Function)-DNF dönüşümünün gereksiz implikantlar ürettiğine değinen araştırmacılar, Boolean ters dağıtım kuralını kullanarak, çok daha az implikant üreten bir algoritma geliştirmişlerdir. Böylece algoritma veri madenciliğinde kullanılan diğer parametre azaltma algoritmaların işleyemediği birçok veri setini işleyebilmektedir.

Cruz-Cano R. ve ark. (2012), tıbbi bilişimde molekül sıralamada lojik sadeleştirmenin nasıl kullanılabileceğine değinmişlerdir. Lojik sadeleştirme, tıbbi bilişim verileri üzerinde daha önce kullanılmasına rağmen, ilk defa bu araştırmada sınıflandırma ve kural oluşturma amacıyla kullanılmıştır. Araştırmacılar lojik sadeleştirmeye dayanan bir metot kullanarak, tıbbi bilişimdeki iki önemli problemi (molekül sıralamada işlevsel

blokların sıralaması ve protein-DNA arasındaki bağ oluşum açıklaması) çözmeye çalışmışlardır.

Borowik G. ve ark. (2012), bilgi teknolojileri alanında verinin etkili olarak temsil edilmesi amacıyla veriler üzerinde kullanılabilen lojik sadeleştirme tekniklerine değinmişlerdir. Kendi geliştirdikleri tekniği diğer veri madenciliği teknikleri ile karşılaştırmışlardır. Araştırmacılar yapay zekâda "rough set" diye geçen bir tekniği veri madenciliği alanına uyarlamış ve veri madenciliğinde özellik indirgemesi yapan yeni bir algoritma sunmuşlardır.

Rawat V. ve ark. (2012), lojik sadeleştirme kullanarak kayıpsız gri tonlu resim sıkıştırmaya değinmiştir. Yeni geliştirdikleri teknikte komşu pikselleri XOR işlemine tabi tutmuşlardır. Ardından işlenen resmi 16x16lık bloklara bölmüşlerdir. Bu blokları Boolean anahtarlama fonksiyon küplerine çevirmişlerdir. Bu küpleri de Quine-McCluskey algoritmasını kullanarak sadeleştirmiş/sıkıştırmışlardır.

Drucker A. D. (2012), sıralama algoritmaları, lojik devreler ve Turing makineleri gibi çeşitli hesaplama modellerinde birleşik hesaplamanın (Joint Computation, JC) gücünü ve etkililiğini araştırmıştır. Araştırmacı tez çalışmasında bağımsız çoklu işlemler için etkili bileşik hesaplama yöntemi geliştirmiştir. Tez çalışmasında Boolean fonksiyonlarından oluşan bir koleksiyonun birleşik hesaplama karmaşıklığı, karar ağaçları yardımıyla incelenmiştir.

Pang Y.(Pang, 2012), blok matrix yöntemi kullanarak kısmen belirlenmiş Boolean fonksiyonları için aritmetik dönüşüm oluşturma algoritması geliştirmiştir. Geliştirilen algoritma matrixi küçük küplere bölerek çarpma işlemini gerçekleştirmekte, böylece geleneksel matrix çarpımı yönteminden en az 10 kat daha hızlı çalışmaktadır.

Lukac M. ve ark. (2012), ters çevrilebilir kuantum devrelerinin minimizasyonu için yeni bir metot tanıtmışlardır. Araştırmacılar kuantum devrelerinin yeni bir gösterimi olan kuantum operatör formunu (Quantum Operatör Form, QOF) tanıtmışlardır. QOF, Reed-Muller formunun farklı kuantum kapılarını kullanabilen bir versiyonudur. Araştırmacılar sadeleştirme kural seti tanıtmış ve CNOT, CV ve CV+ ile devre tasarımında kullanılabilen bir algoritma sunmuşlardır.

Boyar J. ve ark. (2013), uygulamalı lojik sadeleştirme yöntemlerinin zorluk derecelerine ve kriptolojideki uygulamalarına değinmiştir. Araştırmacılar yeni bir kombinasyonlu lojik sadeleştirme tekniği sunmuşlardır. Bu teknik iki aşamadan oluşmaktadır. Algoritma önce devreyi daha lineer hale getirip ardından lineer devredeki

kapı sayılarını indirgemeye çalışmaktadır. Geliştirilen teknik gelişmiş şifreleme standardı (AES)'na uygulanmış ve sonuçları araştırmada paylaşılmıştır.

Lin C. C. ve ark. (2014), eşik mantık devrelerinin (Threshold Logic Circuits, TLC) donanım maliyetini azaltmak için lojik sadeleştirme ve yeniden kablolama işlemlerinden oluşan sezgisel bir yöntem sunmuşlardır. Araştırmacılar böylece eşik mantık kapılarının eşit olup olmadığını kontrol etme işlemini önemli oranda hızlandırmışlardır.

Braun W. ve Menth M. (2014), bilgisayar ağlarındaki switch ve routerlarda bulunan yönlendirme tablolarını sıkıştırmak için lojik minimizasyon yöntemini kullanmışlardır. Araştırmacılar ağda gecikmeye sebep olmayacak kadar kabul edilebilir bir zamanda tam yönlendirme bilgi tabanını (full forwarding information base, FIB) ESPRESSO Heuristic programı yardımıyla sıkıştırmışlardır. Araştırma sonuçları FIB boyutunun %17(40.000 girdi) azaltılabileceğini fakat %1 – %2 sıkıştırma oranından fedakârlık etmek gerektiğini göstermiştir.

Du Z. ve ark. (2014), sinir ağları mimarisinde yaklaşık hesaplama sistemlerinin enerji verimliliğinin arttırılması ve hata payının indirgenmesinde yaklaşık lojik minimizasyonu kullanmışlardır. Geliştirilen sistem 65 nm. tasarım mimarisiyle simüle edilmiştir. Geliştirilen sistemin, diğer sinir ağlarıyla karşılaştırıldığında enerji harcamalarında %43 ile %62 arasında kazanım sağlandığı görülmüştür.

Du Z. ve ark. (2015), uygulamaya özel entegre devreler (Application Specific Integrated Circuits, ASICs) üzerinde, yaklaşık hesaplama sistemlerinin enerji verimliliğinin arttırılmasında yaklaşık lojik minimizasyonu kullanmışlardır. Geliştirilen sistem 65 nm. tasarım mimarisiyle denenmiştir. Sonuçlar geliştirilen sistemin diğer sinir ağlarına göre 1,78 ile 2,67 kat daha az enerji harcadığını göstermiştir.

Duşa A. ve Thiem A. (2015), sosyolog ve politik bilim adamlarının Nitel Karşılaştırmalı Analizde(Qualitative Comparative Analysis, QCA) kullandığı Quine-McCluskey(QMC) algoritmasıyla Boolean sadeleştirme işlemini incelemişlerdir. Araştırmacılar QMC algoritmasının karmaşık çıkış fonksiyonlarını işlemede yavaş kaldığını ve çok fazla hafıza harcadığını belirlemiş ve QMC algoritmasını geliştirerek eQMC (enhanced QMC) algoritmasını tanıtmışlardır. Araştırmacılar geliştirilen algoritmanın hafıza ve zaman açısından performansını değerlendirmişlerdir.

Piscitello A. ve ark. (2016), çok kiracısı olan akıllı binalar için kural seti minimizasyonu yapmışlardır. Araştırmacılar Boolean lojik devrelerinden gelen bilgi ile akıllı binaların farklılıklarını da dikkate alarak özel bir algoritma geliştirmişlerdir.

Algoritmanın etkililiği bir deney sırasında toplanan veriler üzerinde uygulanarak gösterilmiştir.

Wang H. ve Blanton R. D. S. (2016), lojik sadeleştirme kullanarak makine öğrenme sınıflandırıcıları(Machine Learning Classifiers, MLC) topluluğunun indirgenmesini sağlamışlardır. Araştırmada CANOPY adı altında yeni bir topluluk indirgeme yöntemi sunulmuştur. 20 veri setinde yapılan deneylerde CANOPY'nin mevcut topluluk indirgeme yöntemlerinden daha iyi ya da çok yakın olduğu görüşmüştür.

Macii E. ve ark. (2017), CMOS devrelerinin lojik sentezi ve ötesi isimli kitap bölümünde elektronik teknolojilerinin kronolojik gelişimini inceleyerek lojik sentezde karşılaşılabilecek konular ve zorlukları tanıtmışlardır. Araştırmacılar lojik sentez stratejilerini farklı materyal (CMOS, Silicon, etc.) ve boyutlara (1µm, 180 nm, 100 nm.) göre sınıflandırmışlardır.

Papakonstantinou G. (2017), hem teorik olarak hem de deneysel olarak mevcut ESOP sadeleştirme algoritmalarının, karmaşık terimlerin özel veya toplamı(Exclusive or Sum of Complex Terms, ESCT) ifadelerinin sadeleştirilmesinde de kullanılabileceğini kanıtlamıştır. Araştırmada ESCT ifadelerinin ESOP ifadelerine kıyasla daha genel ve ters çevrilebilir olduğu için lojik tasarım ve kuantum devrelerinde daha faydalı olacağı anlatılmıştır. Tam belirlenmiş ve tam belirlenmemiş fonksiyonlar için kesin ESCT ifadelerini ESOP ilkeleriyle bulma yöntemi ilk defa bu araştırmada sunulmuştur.

Selek M. ve ark. (2017), sıtma semptomlarına göre sıtma teşhisi koymak için, lojik indirgenmiş kurallara dayanan medikal uzman sistem tasarlamışlardır. Araştırmacılar Dünya Sağlık Örgütü raporlarına göre sıtma teşhisinde kullanılan 6 farklı veri değişkenine göre kurallar seti hazırlamış ve bu kurallar setini Boolean fonksiyonlarını sadeleştirme işlemi kullanarak indirgemiştir.

Başçiftçi F. ve Avuçlu E. (2018), kanser hücrelerinin teşhisinde indirgenmiş kurallara dayalı uzman sistem geliştirmişlerdir. Kuralların indirgenmesinde lojik fonksiyonların sadeleştirilmesini kullanmışlardır. Geliştirilen yeni sistem 4 çıkış değerine göre kanser teşhisi yapmış ve 8192 ihtimali değerlendirerek daha doğru sonuçlara ulaşmıştır.

Conceição C. ve Reis R. (2019), entegre devrelerde kullanılan transistör sayısının sadeleştirilmesi için yeni bir yöntem tanıtmışlardır. Geliştirilen yöntem bir entegre devrede kullanılacak transistör sayısını, geleneksel devre tasarım uygulamalarına göre %21, diğer lojik sadeleştirme uygulamalarına göre %4 oranında azaltmıştır. Bu sayede entegre devrelerdeki istenmeyen elektrik sızıntısını indirgemişlerdir.

Nazemi M. ve ark. (2019), boolean lojik minimizasyonu kullanarak derin sinir ağlarının gerçeklemesini sağlayan yeni bir method sunmuşlardır. Bu sayede en çok enerji tüketen bölüm olan model parametreleri için hafızaya ulaşan adım kaldırılmıştır. Geliştirilen sistem daha az işlem gücü kullanarak daha düşük gecikme süreleri elde etmiştir.

Nandan V. Ve Rao R. (2020), dijital lojik kapıların ve az enerjili AES şifreleme çekirdeklerinin minimizasyonuna yönelik bir yöntem önermişlerdir. Önerilen yöntem AND, XOR, XNOR ve NAND gibi temel kapıları kullanmaktadır. Araştırma sonuçları önerilen yöntemin hem gerekli alan miktarını hemde enerji tüketimini önemli oranda indirgediğini göstermiştir.

3. MATERYAL VE YÖNTEM

Bu bölümde lojik fonksiyonlar, bitsel operatörler, doğrudan örtme algoritmaları, Petrick yöntemi, rekürsif algoritmalar ve neden C# programının kullanıldığı açıklanmıştır.

3.1. Lojik Fonksiyonlar

Bir Boolean değişkeni bir doğruluk tablosu yâda yapılan doğruluk tanımlamasına göre, $\mathbb{D}$ ={doğru, yanlış} veya $\mathbb{D}$ ={0,1} şeklinde ikili değer alabilen bir yapıdır. Literal ise bir boolean değişkeni veya onun türevleridir. *n*-boyutlu bir boolean uzayında veya Boolean n-uzayında $\mathbb{D}^n$, en küçük elemana (veya köşe noktasına) $\alpha \in \mathbb{D}^n$, minterm denir. Minterm doğruluk tanımlaması yapılmış bir *n* boolean değişkenler vektörü olarak da ifade edilebilir.

Tamamen tanımlanmış Boolean fonksiyonu (completely specified Boolean function) $f: \mathbb{D}^n \rightarrow \mathbb{D}$, *n* giriş değişkenleriyle, bütün tanımlamaların (doğru ya da yanlış) yapıldığı fonksiyondur.

3.1.1. Eksik tanımlanmış fonksiyonlar (Incompletely specified functions)

Bazı giriş değerleri için fonksiyon çıkışının doğru ya da yanlış olmasının önemi yoktur. Bu giriş değerlerine önemsiz giriş (Dont Care, DC) denir. *DC* değerleri "2", "x" ya da " * " şeklinde gösterilir. Eksik tanımlanmış Boolean fonksiyonunu $f: \mathbb{D}^n \rightarrow \mathbb{D}_+$ tanımlamak için $\mathbb{D}$ fonksiyonunu, $\mathbb{D}_+ = \mathbb{D}\ U\ \{ * \}$ şeklinde türetebiliriz. $\mathbb{D}_+$ giriş değerleri doğru, yanlış veya *DC* olabilir. $\alpha \in \mathbb{D}^n$ olmak şartıyla, $f(\alpha) = *$ ise α mintermi için f fonksiyonunun değeri önemli değildir. α, f fonksiyonu için bir *DC* mintermidir. Eğer böyle bir tanımlama yapılmamış ise, f fonksiyonu tam tanımlanmış olarak kabul edilir.

3.2. Bitsel Operatörler

Bilgisayarlar 1 ve 0 ile ifade edilen, açık ya da kapalı durumlarda olabilen elektronik aygıtlardır. Bu sebeple bilgisayarlar ikili (binary) sayı sistemiyle çalışırlar. Bir sayı sistemi sembollerden ve bu sembollerin ifade ettiği kurallardan oluşur. Toplama çıkarma, çarpma gibi aritmetik işlemlerin benzerleri ikili sayı sisteminde de vardır.

Temel bitsel operatörler ve simgeleri Çizelge 3.1'de verilmiştir. Bu bölümde ikili sayı sisteminde kullanılan bitsel operatörler açıklanmaktadır.

Çizelge 3.1. Bitsel operatörlerin gösterimi

Bitsel Operatör	Simgesi
VE (AND)	$\wedge$
VEYA (OR)	$\vee$
Özel VEYA (XOR)	$\oplus$

Sayılar üzerindeki temel bitsel operatör işlemleri için önce sayılar ikili sisteme çevirilir ardından bu ikili sayılar işlenir. A, B ve C sayıları ikili sistemde $A=(\ldots a_2a_1a_0)$, $B=(\ldots b_2b_1b_0)$ ve $C=(\ldots c_2c_1c_0)$ olarak tanımlanırsa, Denklem 3.1- 3.3 geçerlidir.

$$A \,\&\, B = C \quad \Leftrightarrow \quad a_i \wedge b_i = c_i \quad (i \in \mathbb{N} \text{ ve } \forall i \geq 0) \tag{3.1}$$
$$A \mid B = C \quad \Leftrightarrow \quad a_i \vee b_i = c_i \quad (i \in \mathbb{N} \text{ ve } \forall i \geq 0) \tag{3.2}$$
$$A \oplus B = C \quad \Leftrightarrow \quad a_i \oplus b_i = c_i \quad (i \in \mathbb{N} \text{ ve } \forall i \geq 0) \tag{3.3}$$

Örnek 3.1; Örneğin 6 ve 12 sayıları üzerinde temel bitsel operatörleri kullanmak için öncelikle bu sayıları ikili sisteme $6 = (0110)_2$ ve $12 = (1100)_2$ olarak çevirириz.

$6 \,\&\, 12 = 4 = (0100)_2$, $\quad 6 \mid 12 = 14 = (1110)_2$, $\quad 6 \oplus 12 = 10 = (1010)_2$

Temelde denklem 3.1, 3.2 ve 3.3 kullanılarak birçok denklem üretilebilir. Ayrıca temel bitsel operatörlerin değişme, yutma, dağılma gibi özellikleri de vardır.

$$A \,\&\, B = B \,\&\, A, \quad A \mid B = B \mid A, \quad A \oplus B = B \oplus A, \tag{3.4}$$
$$A \,\&\, (B\&C)=(A\&B) \,\&\, C, \quad A \mid (B|C)=(A|B) \mid C, \quad A \oplus (B \oplus C) = (A \oplus B) \oplus C, \tag{3.5}$$
$$(A\&B) \mid C = (A|C) \,\&\, (B|C), \quad (A|B) \,\&\, C = (A\&C) \mid (B\&C), \tag{3.6}$$
$$A \oplus B = (A\&B) \oplus (A|B), \tag{3.7}$$
$$A \,\&\, 0 = 0, \quad A \mid 0 = A, \quad A \oplus 0 = A, \tag{3.8}$$
$$A \,\&\, A = A, \quad A \mid A = A, \quad A \oplus A = 0, \tag{3.9}$$
$$(A \oplus B) \,\&\, C = (A\&C) \oplus (B\&C), \tag{3.10}$$
$$(A\&B) \mid A = A, \quad (A|B) \,\&\, A = A, \tag{3.11}$$
$$A \,\&\, 11\ldots11 = A, \quad A \mid 11\ldots11 = 11\ldots11, \quad A \oplus 11\ldots11 = \neg A, \tag{3.12}$$
$$A \,\&\, \neg A = 0, \quad A \mid \neg A = 11\ldots11, \quad A \oplus \neg A = 11\ldots11, \tag{3.13}$$

$$\neg(A\&B)=\neg A \mid \neg B, \quad \neg(A|B)=\neg A \,\&\, \neg B, \neg(A\oplus B) = A \oplus \neg B = \neg A \oplus B. \qquad (3.14)$$

A ve B sayıları için A ve B'nin her basamağı $a_i \geq b_i$ koşulunu sağlıyorsa A sayısı B sayısını kapsıyor denir ve $A \supseteq B$ şeklinde ifade edilir.

$$A \supseteq B \;\Leftrightarrow\; A \,\&\, B = B \;\Leftrightarrow\; A \mid B = A \;\Leftrightarrow\; \neg A \,\&\, B = 0 \qquad (3.15)$$

Bitsel operatörler toplama çıkarma gibi aritmetiksel işlemlerde de kullanılabilir. Denklem 3.13 deki $A \mid \neg A = 11\ldots11$ prensibi kullanılarak,

$$\neg(A - B) = \neg A + B \qquad (3.16)$$

denklemine ulaşılabilir. Ayrıca bilgisayarda yapılan sola bit kaydırma işlemi o sayının 2'nin kuvvetleriyle çarpımına, sağa bit kaydırma işlemi o sayının 2'nin kuvvetlerine bölümüne eşittir. A sayısının i bit sola kaydırılması denklem 3.17'de, sağa kaydırılması denklem 3.18'de verilmiştir.

$$A \ll i = 2^i \,.\, A \qquad (3.17)$$

$$A \gg i = 2^{-i} \,.\, A \qquad (3.18)$$

$$A \ll i = A \gg -i \quad \text{ve} \quad A \gg i = A \ll -i \qquad (3.19)$$

Bir sayının 2'nin katlarına bölümünden kalanı(mod) bulmak için sayıyı bölmeye gerek yoktur. Bitsel operatörler yardımıyla bölünme sonucunda kalan denklem 3.20'de belirtildiği gibi hesaplanabilir. Denklem 3.20, mintermlerin birbirinden çıkarılmasında büyük kolaylık sağlamaktadır.

$$A \mod 2^i = A \,\&\, (2^i - 1) \qquad (3.20)$$

İkili sayı sistemindeki bir sayının en sağındaki "1" hanesini kaldırmak için denklem 3.21, en sağındaki "1" hanesini tespit etmek için denklem 3.22'den faydalanılabilir.

$$A \,\&\, (A - 1) \qquad (3.21)$$

$$A \,\&\, -A \qquad (3.22)$$

Verilerin maskelenmesi amacıyla çok amaçlı olarak kullanılabilecek sabit değerli sayılar denklem 3.23'de verilmiştir.

$$A = (\ldots 0101010101010101)_2 = (11\ldots11)/3, \quad (3.23)$$
$$B = (\ldots 0011001100110011)_2 = (11\ldots11)/5,$$
$$C = (\ldots 0000111100001111)_2 = (11\ldots11)/17.$$

3.3. Doğrudan Örtme Algoritmaları

Lojik fonksiyonların sadeleştirilmesinde kullanılan çok tanınmış yöntemlerden birisi de doğrudan örtme (direct cover) yöntemidir. Bu yöntemin bütün mintermler kapsanasıya kadar sürekli tekrar eden 4 aşaması aşağıdaki gibidir.

1. Bir minterm seçilir.
2. Mintermi kapsayan uygun bir implikant seçilir.
3. İmplikantın kapsadığı mintermler çıkarılır.
4. Bütün mintermler kapsanıncaya kadar 1-3. adımlar tekrar edilir.

Bütün doğrudan örtme algoritmaları bu temel aşamaları takip eder. Algoritmaların aralarındaki farklılık 1. ve 2. aşamada uygulanan seçim kıstaslarıdır. Promper and Armstrong (1981), Besslich (1986) ve Dueck & Miller (1987) doğrudan örtme algoritmaları için farklı seçim kıstasları önermişlerdir.

3.3.1. Minterm seçme kıstasları

Beslich bütün mintermler için, çevresindeki diğer mintermlere olan uzaklığına göre bir izolasyon ağırlığı hesaplamıştır. Herhangi bir A mintermi için diğer bütün B mintermlerine olan uzaklığı, Manhatten uzaklığı (Manhatten distance, *D)* kullanarak hesaplamıştır. Bu değerlerin toplamı A minterminin izolasyon ağırlığını vermektedir. A ve B mintermlerinin literallerinin sırasıyla $(a_n \ldots a_2 a_1 a_0)$ ve $(b_n \ldots b_2 b_1 b_0)$ olduğunu varsayarsak A ve B mintermleri arasındaki uzaklık denklem 3.24'deki gibi hesaplanır.

$$D_{(\mathrm{A,B})} = \sum_{0 \le i \le n} |\alpha_i - b_i| \tag{3.24}$$

A minterminin *B* minterminin izolasyon ağırlığına katkısını hesaplamak için bulunan uzaklık değeri *D* mintermler arası ağırlık etkisi *w* hesaplamak için kullanılır, ardından bulunan *w*'ler kullanılarak mintermin toplam izolasyon ağırlığı (Isolation Weight, *IW*) denklem 3.25'deki gibi hesaplanır. Bu yöntemde mintermlerin kapsanmasına en küçük *IW* değerine sahip mintermden başlanır.

$$w(\mathrm{A,B}) = 2^{[n(r-1)-D(\mathrm{A,B})]} \implies IW(\mathrm{A}) = \sum_{B \in Trans} Trans(B) \times w(\mathrm{A,B}) \tag{3.25}$$

Minterm seçmek için kullanılan bir diğer yöntem Dueck ve Miller (1987) tarafından sunulan kümeleme faktörü (Clustering Factor, CF)'dir. Her minterm için kümeleme faktörü denklem 3.26'daki gibi hesaplanır.

$$CF(A) = DEA_A \times (r-1) + EA_A \tag{3.26}$$

Bu denklemde DEA_A, *A* minterminin birleşebileceği 0 değerinde olmayan mintermlerin yön sayısını ve EA_A ise A minterminin birleşebileceği minterm sayısını ifade eder. Bu teknikte mintermlerin kapsanmasına en küçük kümeleme faktörü değerine sahip mintermden başlanır.

Yang ve Wang (1990) tarafından sunulan algoritma Dueck ve Miller'in sunduğu algoritmaya çok yakındır. Bu algoritmada A minterminin bağlantılı olduğu minterm sayısı CMC_A, A minterminin bağlantılı olduğu mintermlerin yön sayısı ise EDC_A olarak belirtilmiş ve A mintermine göre kümeleme faktörü $CFN = EDC_A \times (r-1) + CMC_A$ olarak hesaplanmıştır.

3.3.2. İmplikant seçme kriterleri

Bir mintermi kapsayan implikantlar bulunduktan sonra, hangi implikantın seçileceği çok önemlidir. İmplikant seçimi sonuç kalitesini ve zaman karmaşıklığını doğrudan etkiler. Bu sebeple araştırmacılar farklı implikant seçme yöntemleri önermişlerdir. İlk akla gelen seçme yöntemi implikantların kapsadığı minterm sayısına bakmaktır. En çok mintermi kapsayan (Largest Reduced to Zero, LRZ) implikant AI olarak seçilebilir.

İmplikant seçme yöntemlerinden birisi de Dueck ve Miller (1987) tarafından önerilen göreli geçiş sayısı (Relative Break Count, RBC)'dir. Bu yöntemde bitişik iki mintermin değerleri birbirinden farklı ise geçiş (Break), bütün fonksiyondaki toplamına ise geçiş sayısı (break count) denir. Göreli geçiş sayısı ise bir implikant seçildiğinde kalan fonksiyonun ne kadar sadeleştiğini yani geçiş sayısının ne kadar azaldığını tespit etmek için kullanılır.

Komşuluk göreli sayım (Neighbourhood Relative Count, NRC) yöntemi Yang ve Wang (1990) tarafından sunulmuştur. Bu yöntemde implikantların komşularla birleştirme gücünün derecesi ölçülür. Bir implikantın seçilmesi, o implikant ile komşu mintermler arasındaki bağın koparılması olarak değerlendirilir. Bu yöntemde en izole implikant yani NRC puanı en düşük olan implikant seçilir. Farklı algoritmaların minterm seçme ve implikant seçme kriterleri Çizelge 3.2'de verilmiştir.

Çizelge 3.2. Algoritmaların minterm ve implikant seçme kriterleri

Algoritma	**Minterm Seçme Kriterleri**				**İmplikant Seçme Kriterleri**		
	Rastgele	***IW***	***CF***	***CFN***	**LRZ**	**RBC**	**NRC**
Beslich		⁂			⁂		
Dueck&Miller			⁂			⁂	
Yang&Wang				⁂			⁂
Armstrong	⁂				⁂		

3.4. Petrick Yöntemi

Petrick yöntemi bir çeşit dallandırma ve sınıflandırma (branch and bound) yaklaşımını kullanarak bir fonksiyon için en iyi çözümü bulmayı amaçlar. Petrick yöntemi lojik fonksiyonların sadeleştirilmesinde en küçük SOP ifadesinin bulunmasında kullanılabilir.

Bu yöntemde öncelikle mintermleri kapsayan bütün *Imp*'ler ve bu *Imp*'lerin kapsadığı *ON_Min*'ler bulunur. *Imp*'lerin satırlara, *On_Min*'lerin sütunlara yerleştirildiği bir tablo oluşturulur. Satırlardaki *Imp*'ler etiketlenir. Bütün mintermlerin kapsanması istendiği için bir mintermi kapsayan *Imp* alternatifleri *"veya"* işlemi ile parantez içine alınır. Ardından diğer mintermler içinde aynı işlem yapılır ve bu ifadeler *"ve"* işlemine tabi tutulur. Sonuçta elde edilen POS ifadesi en küçük SOP ifadesine indirgenir (Arslan ve Sertbaş, 2002) . Çizelge 3.3'de sayılar (0, 1, …, 7) mintermleri, harfler (*A*, *B*, …, *F*) implikantları göstermektedir.

Çizelge 3.3. Petrick yöntemi uygulama tablosu

$\mathcal{X}_{(abc)}$		*0*	*1*	*2*	*5*	*6*	*7*
A	$\neg a \neg b$	+	+				
B	$\neg a \neg c$	+		+			
C	$\neg b\ c$		+		+		
D	$b \neg c$			+		+	
E	$a\ c$				+		+
F	$a\ b$					+	+

0 minterminin kapsanması için *A* veya *B* implikantının sonuç dosyasında olması gerekir. Bu durum (*A*+*B*) şeklinde gösterilir. Benzer şekilde sonuç dosyasında 1 mintermini kapsayan *A* veya *C* implikantının (*A*+*C*) olması gerekir. 2 mintermi için (*B*+*D*), 5 mintermi için (*C*+*E*), 6 mintermi için (*D*+*F*) ve 7 mintermi için (*E*+*F*) implikantının olması gerekir. Dolayısıyla bütün mintermleri kapsayan $\mathcal{X}$ fonksiyonu şöyle yazılabilir.

$$\mathcal{X} = (A+B)\,(A+C)\,(B+D)\,(C+E)\,(D+F)\,(E+F) = 1 \tag{3.27}$$

Parantez içerisindeki ifadelerin birbiriyle çarpımından sonra dağıtım kuralı ve $(A+B)\,(A+C) = A + BC$ kuralı uygulanarak ifadeler sadeleştirilir. Ardından $A + AB = A$ kuralıyla fonksiyonun en sade haline ulaşılır. Elde edilen ifadenin içinde en az sayıda implikant içeren çarpım terimi, fonksiyonun en sade ifadesidir.

$$\mathcal{X} = (A+B)\,(A+C)\,(B+D)\,(C+E)\,(D+F)\,(E+F) = 1 \tag{3.28}$$
$$\mathcal{X} = (A+BC)\,(D+BF)\,(E+CF)$$
$$\mathcal{X} = (AD+ABF+BCD+BCF)\,(E+CF)$$
$$\mathcal{X} = ADE + ABEF + BCDE + BCEF + ACDF + ABCF + BCDF + BCF$$

$\mathcal{X} = ADE + ABEF + BCDE + ACDF + BCF = 1$

F fonksiyonunun 5 çözüm kümesi vardır. En az implikant (3) içeren *ADE* ve *BCF* çözüm kümesidir. F fonksiyonu için $ADE = \{\neg a \neg b + b \neg c + ac\}$ ve $BCF = \{\neg a \neg c + \neg bc + ab\}$ minimum çözümlerdir.

3.5. Paralel Algoritmalar

Günümüzde kullanılan birçok algoritma sıralı bir şekilde çalışmaktadır. İşlemci hızları yıllardır üssel olarak artmaktadır. Fakat artık limitleri zorlayan işlemci hızlarını arttırmak çok maliyetli olmuştur. Hızı arttırmakta zorlanan üreticiler işlemcilerin sayısını arttırıp, aynı anda birden fazla bilgi işlemeyi mümkün kılmışlardır. Paralel algoritma farklı işlemcilerde farklı komutları aynı anda işleyen ve sonra bütün çıktıları tek bir sonuçta birleştiren algoritmadır. Bu algoritmalar bir problemi daha küçük alt problemlere bölerek aynı anda alt problemleri ayrı ayrı çözerler. Ardından elde edilen sonuçları biraraya getirerek istenilen sonuca ulaşırlar.

Bir problemi paralel algoritmalarla çözmek için öncelikle problem çok küçük alt problemlere ayırılır. Ardından bu problemler arasında bilgi alışverişini gerektiren iletişim bağlantıları kurgulanır. Ardından iletişim bağlantılarını minimumda tutacak problem kümeleri oluşturulur ve bu problem kümeleri haritaya yerleştirilir. Bir problemin çözümü için paralel algoritma tasarımı şekil 3.1'de verilmiştir.



Şekil 3.1. Paralel algoritma tasarım şeması

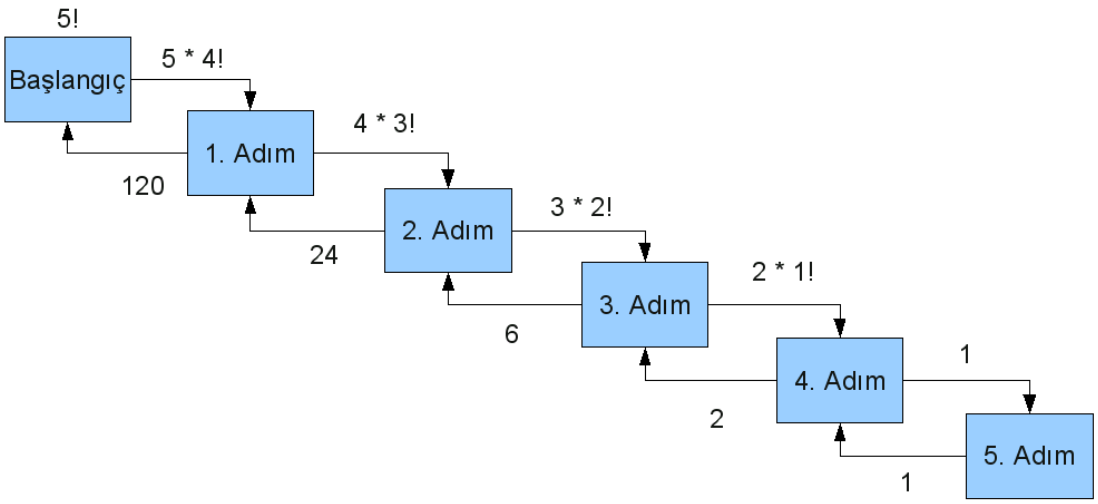
Bazı problemlerin doğası gereği, bu problemlerin çözümünde paralel algoritmaları kullanmak bazı durumlarda işe yaramayabilir. Bunun birçok sebebi olabilir. Örneğin geliştirilen paralel algoritma bilgisayarın işlemci yapısına uygun olmayabilir, problemin yapısı paralel programlamaya uygun olmayabilir ya da algoritma programa verimli bir şekilde kodlanmamış olabilir (Ivutin ve ark., 2017).

3.6. Rekürsif Algoritmalar

Bir problemi çözmek için aynı problemin daha küçük bir halini çözmekten faydalanıyorsak ve bu işlem küçük problem kendiliğinden çözülesiye kadar devam ediyorsa bu algoritmaya rekürsif algoritma denir. Rekürsif algoritmaların çalışması için, küçük alt problemlerin temel bir şarta ulaşması gerekir. Rekürsif algoritmaların iki temel kuralı vardır; problem aynı problemin daha küçük alt bileşeni olarak ifade edilebilmelidir ve en alt problemin daha fazla alt problem olmadan temel bir şarta ulaşması gerekir. Rekürsif algoritmalar hem daha anlaşılır olduğundan hemde yeni nesil bilgisayar işlemcilerinde daha verimli çalıştıklarından dolayı tercih edilmektedir.

İteratif algoritmalarda zaman karmaşıklığını hesaplamak için döngü sayısını esas alınır. Rekürsif algoritmalarda ise fonksiyonun kendisini kaç defa çağırdığı esas alınır. Dolayısıyla kendisini (n) defa çağıran bir rekürsif algoritmanın zaman karmaşıklığı O(n)'dir. Bir fonksiyon kendisini çağırdığı zaman mevcut fonksiyonun değişkenlerinin ve parametrelerinin bellekte saklanması gerekir. Bu sebeple rekürsif algoritmaların hafıza karmaşıklığı, iterative algoritmalara göre yüksektir.

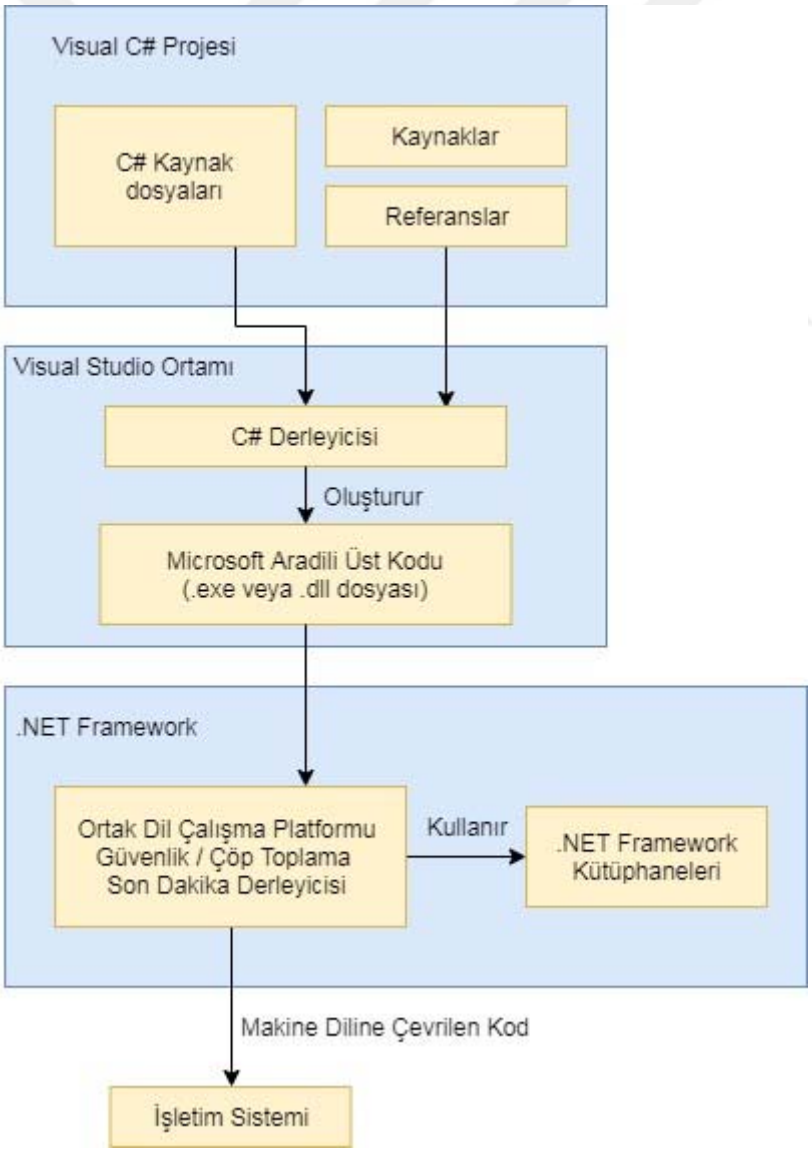
Rekürsif fonksiyonlar kendini çağırıyorsa doğrudan özyinelemeli, çağırdığı fonksiyon kendini çağırıyorsa dolaylı özyinelemeli olarak tanımlanırlar. Rekürsif algoritmalar dallanma (branching) algoritmasının bir türüdür. Şekil 3.2'de rekürsif algoritmaların çalışma prensibi verilmiştir.



Şekil 3.2. Rekürsif algoritma çalışma prensibi (Çebi, 2006)

3.7. Programlama Dili Seçimi (C#)

Hazırlanan programın kullanım arayüzünün kolay anlaşılır olması ve başka programlarla uyumlu çalışabilmesi açısından Microsoft Forms destekleyen bir programlama diline ihtiyaç duyulmuştur. Ayrıca hazırlanan programda bellek kullanımı önemli bir kıstas olduğu için geliştirilen platformun çöp toplayıcı (garbage collector) özelliğinin olması gerekmektedir. Geliştirilen algoritmanın mutlaka nesne yönelimli (object oriented) ve çok çekirdekli işlemci gücünü kullanabilen, paralel programlamaya uygun bir programda kodlanması gerekmektedir. Bütün bu ihtiyaçlar düşünüldüğünde Visual C# programı ön plana çıkmaktadır. Bir C# projesinin çalışma prensibi Şekil 3.3'de verilmiştir.



Şekil 3.3. C# Projesi çalışma prensibi (Microsoft, 2015)

Yeni nesil bir programlama dili olan C#'ı her ne kadar Microsoft geliştirmiş olsa da Avrupa Bilgisayar Üreticileri Birliği (ECMA) ve Uluslararası Standartlar Teşkilâtı (ISO) tarafından standartları belirlenmiştir. Programlama mantığı Java gibi tamamen nesneye yönelik olarak tasarlanmış olmasına rağmen kod söz dizimi C++ diline benzemektedir. C# orta seviyeli bir programlama dili olarak, makine diline ve insan algısına eşit mesafede olduğu söylenebilir. C# uygulamalarının geliştirilmesi ve çalıştırılması için .NET Framework'üne ihtiyaç duyulmaktadır. Visual Studio ortamında geliştirilen uygulamalar, Windows 7 ve üzeri işletim sistemlerinde doğrudan çalışırken, daha eski işletim sistemlerinde uygulamayı çalıştırmak için .Net Framework kurulması gerekir.

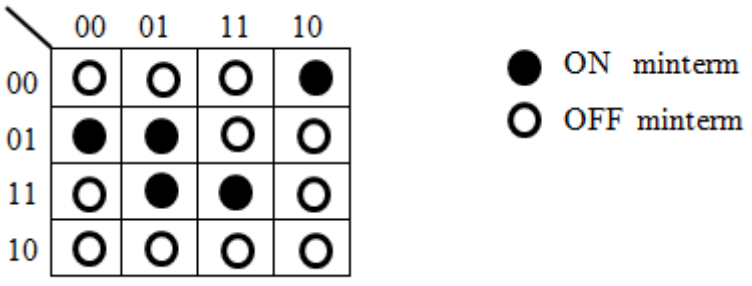
C# programlama dili, çöp toplama, nesneler arası miras / kalıtım, soyutlama, çok biçimlilik ve sarmalama özellikleri olan nesne yönelimli programlama dilidir. Bu dilin amacı programcının üretkenliğini arttırmaktır (Goyal, 2014). İlk defa 1999 yılında nesne yönelimli ve otomatik çöp toplama özellikleri ile tanıtılan C#, 2012 yılında asenkron programlama özelliği ile güncellenmiştir. Bu sayede kuyruktaki komutları paralel bir şekilde işleyebilmekte, çoklu treadler oluşturarak programların daha hızlı çalışmasını sağlamaktadır.

4. LOJİK FONKSİYONLARIN BİTSEL SADELEŞTİRİLMESİ

Bu bölümde araştırmacı lojik fonksiyonların bitsel operatörler yardımıyla sadeleştirilmesini anlatmıştır. Bununla birlikte, en sade sonuca ulaşmak için giriş fonksiyonlarında bulunan izole mintermleri tespit eden ve sadeleştirmeye izole mintermlerden başlayan Kesin Sonuç Kapsama Algoritması ve Yakın Sonuç Kapsama Algoritması (YSKA) tanıtılmıştır. Bu algoritmalarda kullanılan bitsel operatör işlemleri açıklanmış, bitsel operatörlerin mintermlerin genişletilmesi, kapsanması ve çıkarılmasında nasıl kullanılabileceği gösterilmiştir. Sadeleştirme algoritmalarının paralel programlamaya ne kadar uygun olduğunun tespit edilebilmesi için, geliştirilen algoritmalar ayrıca paralel hale getirilmiştir. Geliştirilen bütün algoritmalar C# dilinde kodlanmıştır. Kesin veya yakın sonuç bulan, izole veya rastgele minterm yöntemini kullanan, seri veya paralel programlanmış toplam 8 farklı algoritma geliştirilmiş ve kodlanmıştır.

4.1. İzole Mintermlerin Tespiti

Lojik fonksiyonların iki seviyeli sadeleştirilmesinin ilk aşamasında bir minterm seçilir ve onu örten AI'ler bulunur. Aynı algoritma, mintermleri farklı sıralanmış fonksiyonu farklı ölçüde sadeleştirebilir. Sadeleştirmeye başlamak için doğru mintermin seçilmesi çok önemlidir (Dueck, 1988). En sade sonuç fonksiyonuna ulaşmak için sadeleştirmeye, kapsaması zor olan mintermlerden başlanmalıdır.



Şekil 4.1. İzole mintermi gösteren karnaugh haritası

Komşuları arasında ON mintermi olmayan veya çok fazla OFF mintermi olan mintermlere izole minterm denir (Başçiftçi, 2014). Şekil 4.1 de Karnaugh Haritası verilen fonksiyon dosyasında "0010" minterminin bütün komşuları OFF mintermlerinden

oluşmuş ve OFF mintermleri tarafından tamamen izole edilmiştir. Bu sebeple "0010" mintermini kapsayan bir üst çarpan bulunamaz. Verilen fonksiyon dosyası için bu minterme izole minterm denir. Benzer şekilde "0100" ve "1111" mintermlerinin 3 komşusu OFF mintermi, sadece 1 komşuları ON mintermidir. Fakat "0101" ve "1101" mintermlerinin 2 komşusu OFF mintermi, 2 komşusu ise ON mintermidir. Birlikte değerlendirildiğinde "0100" ve "1111" mintermleri, "0101" ve "1101" mintermlerine kıyasla daha izole durumdadır.

İzole mintermleri tespit etmek için geliştirilen algoritmalar ON mintermlerini kullanmaktadır. Bu araştırmada ise OFF mintermlerin yardımıyla tespit edilmiştir. İzole mintermlerin tespit edilmesi için lojik komşuluk ilişkilerine dayanan bir algoritma geliştirilmiştir. İzole mintermler diğer ON mintermlerinden uzakta, OFF mintermleriyle komşu olan mintermlerdir. Birbirine komşu olan iki mintermin sadece tek bir literali farklı olmalıdır. Geliştirilen algoritma ON mintermlerini OFF mintermleri ile karşılaştırmakta ve komşuluk ilişkilerine göre mintermlere değer atamaktadır. Şekil 4.2'de izole mintermlerin tespiti algoritması kabakodu verilmiştir.

```
PROGRAM İZOLE_MİNTERM (N,F,D) //Sırasıyla On, Off ve DC kümeleri
BEGİN
  FOR all "on" in N DO //Bütün ON mintermleri için hesapla
      Sn = 0 ;     // İlk değeri sıfırla
      FOR all "off" in F DO //Bütün OFF kümesiyle karşılaştır
           g = 0; //
           gNum = 0;
           g = on ⊕ off;//on ve off mintermini XOR yap
           gNum = g içindeki "1" sayısı g;//Komşu sayısını bul
           Sn = Sn + gNum;
  SORT N (S);  //S kümesine uygun N kümesini sırala
  RETURN N;
END
```

Şekil 4.2. İzole mintermlerin tespiti algoritması kaba kodu (pseudo code)

Bu algoritma bütün ON mintermlerini herbir OFF mintermi ile karşılaştırır. Karşılaştırma işleminde iki minterme XOR işlemi uygulanır. Ardından oluşan sayıdaki "1" değerleri sayılarak, mintermlerin kaç literalinin farklı olduğu tespit edilir. Örneğin "0001" minterminin, $\mathbb{D}_{off}$={0000, 0010, 0110, 0111, 1010, 1011, 1100, 1110} kümesine göre izole olma seviyesi Çizelge 4.1'de hesaplanmıştır.

Çizelge 4.1. Örnek mintermin izole seviyesinin hesaplanması

ON Minterm	OFF Minterm	ON $\oplus$ OFF	Farklı Literal Sayısı
0001	0000	0001	1
0001	0010	0011	2
0001	0110	0111	3
0001	0111	0110	2
0001	1010	1011	3
0001	1011	1010	2
0001	1100	1101	3
0001	1110	1111	4
		İzole Seviyesi:	**20**

4.2. Bitsel Operatör İşlemleri

Bu bölümde fonksiyon sadeleştirmede kullanılan bitsel gösterim ve bitsel operatörler tanıtılmıştır. Ayrıca, geliştirilen algoritmalarda kullanılan minterm genişletme, minterm kapsama ve mintermlerin birbirinden çıkarılması işlemleri gösterilmiştir.

4.2.1. Mintermlerin gösterimi

Lojik fonksiyonların iki seviyeli sadeleştirilmesinin bilgisayar ortamında daha hızlı yapılabilmesi için bitsel operatörler kullanılmıştır. Lojik fonksiyonların sadeleştirilmesi sırasında ve sonrasında alabileceği *DC* değeri { * } vardır. Bölüm 1.2.1'de anlatıldığı gibi eksik tanımlanmış Boolean fonksiyonunu f: $\mathbb{D}^n \rightarrow \mathbb{D}_+$ tanımlamak için $\mathbb{D}$ fonksiyonunu, $\mathbb{D}_+ = \mathbb{D} \; U \; \{ * \}$ şeklinde türetebiliriz. $\mathbb{D}_+$ değerleri doğru, yanlış veya *DC* olabilir. $\alpha \in \mathbb{D}^n$ olmak şartıyla, $f(\alpha) = *$ ise α mintermi için f fonksiyonunun değeri önemli değildir. α girişi f fonksiyonu için bir *DC* değeridir. 3 değer (0, 1, *) alabilen bir girişi temsil etmek için en az 2 tane ikili sayıya (binary) ihtiyacımız vardır. Bu sebeple fonksiyon literalleri iki binary sayı çifti ile temsil edilecektir. Fonksiyon literallerinin bitsel temsili Çizelge 4.2'de verilmiştir.

Çizelge 4.2. Fonksiyon değerlerinin bitsel temsili

Literal Değerleri	Bitsel Temsili
Lojik "0" değeri	01
Lojik "1" değeri	10
Lojik *DC* değeri	11

Örnek 3.1; $\mathbb{D}$="1*0*1" minterminin bitsel gösterimi Çizelge 4.3'de verilmiştir. 5 literalden oluşan $\mathbb{D}$ mintermi, iki tane 5 bitlik sayı ile temsil edilmektedir. $\mathbb{D}$="1*0*1" mintermi sol taraftaki bitleri tutan değer $\mathbb{D}_L$="11011", sağ taraftaki bitleri tutan değer $\mathbb{D}_R$="01110" olarak gösterilecektir.

Çizelge 4.3. Bir mintermin bitsel gösterimi

$\mathbb{D}$ = **"1*0*1"**	**1**	*****	**0**	*****	**1**
Literallerin bitsel temsili	10	11	01	11	10
Sol taraftaki bit ($\mathbb{D}_L$)	1	1	0	1	1
Sağ taraftaki bit ($\mathbb{D}_R$)	0	1	1	1	0

Bu tez çalışmasında kullanılan bitsel operatörler ve simgeleri Çizelge 4.4'de verilmiştir.

Çizelge 4.4. Bitsel operatörlerin gösterimi

Bitsel Operatör	**Simgesi**
VE (AND)	$\wedge$
VEYA (OR)	$\vee$
Özel VEYA (XOR)	$\oplus$
Genişletme	$\otimes$
Kapsama	$\textcircled{\scriptsize ⫴}$
Çıkarma	$\ominus$

4.2.2. Mintermlerin bitsel operatörler ile genişletilmesi

Bir ON mintermi ($\mathcal{X}$), genişletilmek için OFF mintermleri ($\mathcal{Y}$) ile bitsel olarak karşılaştırılır. Karşılaştırma sonucunda aynı olan literaller "*" değeri üretirken, farklı olan literaller OFF minterminde bulunan literal değerini taşır. Ortaya çıkan $\mathcal{Z}$ mintermi, $\mathcal{X}$ minterminin, $\mathcal{Y}$ mintermi ile genişletilmiş halidir. Genişletme işlemi "$\otimes$" simgesi ile gösterilmiştir.

$$\mathcal{Z} = \mathcal{X} \otimes \mathcal{Y} \tag{4.1}$$
$$\mathcal{Z}_i = \mathcal{X}_i \otimes \mathcal{Y}_i$$

$\mathcal{Z}$ minterminin herhangi bir *i* literalini bulmak için $x_i \in \mathcal{X} \wedge y_i \in \mathcal{Y}$ olmak şartıyla

$$z_i : \{ x_i = y_i \Rightarrow z_i = * \mid x_i \neq y_i \Rightarrow z_i = y_i \} \quad (4.2)$$

$\mathcal{X}$ minterminin sol ve sağ bitlerini tutan değerler $\mathcal{X}_L$ ve $\mathcal{X}_R$, $\mathcal{Y}$ minterminin sol ve sağ bitlerini tutan değerler $\mathcal{Y}_L$ ve $\mathcal{Y}_R$ olsun. Yukarıda verilen formüller bitsel operatörler yardımıyla Denklem 4.3' teki gibi uygulanabilir.

$$\mathcal{Z}_L = \neg\mathcal{X}_L \vee \mathcal{Y}_L \quad (4.3)$$
$$\mathcal{Z}_R = \neg\mathcal{X}_R \vee \mathcal{Y}_R$$
$$\mathcal{Z} = \mathcal{Z}_L + \mathcal{Z}_R$$

Örnek 3.2; $\mathcal{X}_i$=0010 mintermini $\mathcal{Y}$={0001, 0011, 1010, 1100} kümesiyle bitsel operatörleri kullanarak genişletelim. Çizelge 4.5'de gösterildiği gibi verilen mintermler sağ ve sol bitler olarak ayırılır, ardından Denklem 4.3'de verilen formüller kullanılarak genişletilir.

Çizelge 4.5. $\mathcal{X}_i$'nin sol ve sağ bit gösterimi

$\mathcal{X}_i$	**0**	**0**	**1**	**0**
$\mathcal{X}_i$'nin sol biti	0	0	1	0
$\mathcal{X}_i$'nin sağ biti	1	1	0	1

$\mathcal{Y}_1$=0001 minterminin sol biti (0001), sağ biti (1110) değerindedir.
$\mathcal{Z}_L = \neg\mathcal{X}_L \vee \mathcal{Y}_L$ = 1101 ∨ 0001 = 1101
$\mathcal{Z}_R = \neg\mathcal{X}_R \vee \mathcal{Y}_R$ = 0010 ∨ 1110 = 1110
$\mathcal{Z} = \mathcal{Z}_L + \mathcal{Z}_R = 1101 + 1110 = **01$

$\mathcal{Y}_2$=0011 minterminin sol biti (0011), sağ biti (1100) değerindedir.
$\mathcal{Z}_L = \neg\mathcal{X}_L \vee \mathcal{Y}_L$ = 1101 ∨ 0011 = 1111
$\mathcal{Z}_R = \neg\mathcal{X}_R \vee \mathcal{Y}_R$ = 0010 ∨ 1100 = 1110
$\mathcal{Z} = \mathcal{Z}_L + \mathcal{Z}_R = 1111 + 1110 = ***1$

$\mathcal{Y}_3$ ve $\mathcal{Y}_4$ içinde ayrı ayrı hesaplamalar yapılır ve $\mathcal{X}_i$ minterminin $\mathcal{Y}$ kümesiyle genişletilmesinden oluşan $\mathcal{Z}$ kümesi Çizelge 4.6' da gösterilmiştir.

Çizelge 4.6. X_i'nin genişletilmesiyle oluşan Z kümesi

X_i	Y	Z
0010	0001	**01
0010	0011	***1
0010	1010	1***
0010	1100	110*

4.2.3. Birbirini kapsayan mintermlerin bitsel operatörler ile tespiti

X ve Y mintermlerinin birbirini kapsayıp kapsamadığını kontrol etmek için iki mintermin kesişiminden oluşan Z minterminden yararlanılır.

$$Z = X \textcircled{\scriptsize ||} Y \tag{4.4}$$
$$Z_L = X_L \wedge Y_L$$
$$Z_R = X_R \wedge Y_R$$

Bazen X ve Y mintermlerinin aynı literalleri zıt değerler (0 ve 1) almış olabilir. Aynı literalleri zıt değer alan X ve Y mintermleri birbirini kapsayamaz. Dolayısıyla oluşan Z mintermi geçerli değildir. Z minterminin geçerli olup olmadığını anlamak için denklem 4.5 de verilen işlemin sonucu kontrol edilir. Eğer sonuç 0 ise oluşan Z mintermi geçerlidir.

$$P = (Z_L \vee Z_R) \oplus 111...111 \tag{4.5}$$
$$P = 0 \Rightarrow \exists\, Z \mid P \neq 0 \Rightarrow Z \in \emptyset$$

$$(Z_L = X_L \wedge Z_R = X_R) \Rightarrow Z = X, \tag{4.6}$$
$$(Z_L = Y_L \wedge Z_R = Y_R) \Rightarrow Z = Y,$$

$$\text{Eğer } Z = X \Rightarrow Y \supset X, \tag{4.7}$$
$$\text{Eğer } Z = Y \Rightarrow X \supset Y$$
$$\text{Eğer } Z \neq X \text{ ve } Z \neq Y \Rightarrow X \not\supset Y, Y \not\supset X$$

Denklem 4.6 Z'nin X veya Y'ye eşit olup olmadığını kontrol eder. Eğer iki mintermin hem sağ, hemde sol bitlerini tutan değerler birbirine eşit ise bu iki minterm birbirine eşittir. Denklem 4.7'de hangi mintermin diğerini kapsadığı ifade edilmiştir. Eğer

kesişim sonucu oluşan $\mathcal{Z} = \mathcal{Y}$ ise $\mathcal{X}$ mintermi $\mathcal{Y}$ mintermini kapsar, $\mathcal{Z} = \mathcal{X}$ ise $\mathcal{Y}$ mintermi $\mathcal{X}$ mintermini kapsar. Eğer $\mathcal{Z}$ mintermi her iki minterme de eşit değilse, iki minterm de birbirini kapsamaz.

Örnek 3.3; Bitsel operatörler kullanarak $\mathcal{X} = 0*0*$ mintermi ile $\mathcal{Y} = 0100$ minterminin birbirini kapsama durumunu inceleyelim. Öncelikle mintermler sol ve sağ bit olarak ayırılır, ardından denklem 4.4'e göre $\mathcal{Z}$ mintermi oluşturulur.

$\mathcal{X} = 0*0* \Rightarrow \mathcal{X}_{\mathrm{L}} = 0101,\ \mathcal{X}_{\mathrm{R}} = 1111$
$\mathcal{Y} = 0100 \Rightarrow \mathcal{Y}_{\mathrm{L}} = 0100,\ \mathcal{Y}_{\mathrm{R}} = 1011$
$\mathcal{Z} = \mathcal{X} \textcircled{\shortparallel} \mathcal{Y}$
$\mathcal{Z}_L = \mathcal{X}_L \wedge \mathcal{Y}_L = 0101 \wedge 0100 = 0100$
$\mathcal{Z}_R = \mathcal{X}_R \wedge \mathcal{Y}_R = 1111 \wedge 1011 = 1011$
$\mathcal{Z} = 0100$

$\mathcal{Z}$ minterminin geçerli olup olmadığı Denklem 4.5'e göre kontrol edilir.

$\mathcal{P} = (\mathcal{Z}_L \vee \mathcal{Z}_R) \oplus 111\ldots111 = (0100 \vee 1011) \oplus 1111 = 1111 \oplus 1111 = 0$
$\mathcal{P} = 0 \Rightarrow \exists\ \mathcal{Z}$ vardır ve geçerlidir.
$(\mathcal{Z}_L = \mathcal{Y}_L \wedge \mathcal{Z}_R = \mathcal{Y}_R) \Rightarrow \mathcal{Z} = \mathcal{Y}$ olduğundan $\mathcal{Z} = \mathcal{Y} \Rightarrow \mathcal{X} \supset \mathcal{Y}$.

$\mathcal{X} = 0*0*$ mintermi, $\mathcal{Y} = 0100$ mintermini kapsar.

4.2.4. Mintermlerin bitsel operatörler ile birbirinden çıkarılması

Bir $\mathcal{X}$ minterminden $\mathcal{Y}$ mintermini çıkarmak ($\ominus$) için öncelikle bu iki mintermin kesişmesi gerekmektedir. Birbiriyle kesişmeyen, ortak bir literalleri olmayan mintermler birbirinden çıkarılamaz. Başka bir ifadeyle çıkarma sonucu oluşacak minterm, çıkarılan minterm ile aynı olur. $x_i \in \mathcal{X}$, $y_i \in \mathcal{Y}$ olmak şartıyla,

$f(z_i) : (x_i = \neg y_i \implies \mathcal{X} \wedge \mathcal{Y} = \emptyset \iff \mathcal{X} \ominus \mathcal{Y} = \mathcal{X})$ (4.8)

DC literalleri ifade edilirken kullanılan *"*"* ifadesi hem *"0"*, hemde *"1"* değerini temsil eder. Bu sebeple *"*"* literalinden *"0"* çıkarılırsa *"1"*, *"1"* çıkarılırsa *"0"* literali kalır. $x_i \in \mathcal{X}$, x_i = * olmak şartıyla,

$$f(z_i): (y_i = 0 \Longrightarrow x_i \ominus y_i = 1 \mid y_i = 1 \Longrightarrow x_i \ominus y_i = 0) \quad (4.9)$$

$\mathcal{X}$ ve $\mathcal{Y}$ literallerinin birbirine eşit olduğu durumlarda $\mathcal{X}$ literal değerleri korunur. Herhangi bir işlem yapılmaz. $x_i \in \mathcal{X}$, $y_i \in \mathcal{Y}$ olmak şartıyla,

$$f(z_i): (x_i = y_i \mid x_i = \{0,1\} \text{ ve } y_i = * \implies \mathcal{X} \ominus \mathcal{Y} = \mathcal{X}) \quad (4.10)$$

Denklem 4.8, 4.9 ve 4.10'da açıklanan $x_i \ominus y_i = z_i$ işlemi sonucunda oluşan z_i değerleri Çizelge 4.7'de özetlenmiştir. Çıkarma işleminin mümkün olmadığı ihtimaller $\emptyset$ simgesi ile gösterilmiştir.

Çizelge 4.7. x_i ve y_i literallerine göre z_i değeri

x_i	y_i	z_i
0	0	0
0	1	$\emptyset$
0	*	0
1	0	$\emptyset$
1	1	1
1	*	1
*	0	1
*	1	0
*	*	*

4.2.5. Fonksiyon kapsama algoritmalarında kullanılan tanımlar

Bu bölümde lojik fonksiyonların kapsanması amacıyla geliştirilen algoritmalar açıklanmıştır. Geliştirilen algoritmalarda kullanılan temel tanımlar ve açıklayıcı örnekler aşağıdaki gibidir.

Tanım 3.1; İki lojik fonksiyon $\mathcal{X}$ ve $\mathcal{Y}$ için, eğer $\mathcal{Y}_i$ = 1 olan bütün mintermler için $\mathcal{X}_i$=1 ise, $\mathcal{X}$ fonksiyonu $\mathcal{Y}$ fonksiyonunu **içerir** ve $\mathcal{X} \geq \mathcal{Y}$ olarak gösterilir.

Örnek 3.5; $\mathcal{X}_{(a,b,c)}$ = {¬abc, a¬bc, abc, ab¬c}, ve $\mathcal{Y}_{(a,b,c)}$ = {¬abc, abc} lojik fonksiyonları olsun. $\mathcal{Y}$'nin bütün mintermleri $\mathcal{X}$ fonksiyonunda bulunduğu için $\mathcal{X}$ fonksiyonu $\mathcal{Y}$ fonksiyonunu içerir, $\mathcal{X} \geq \mathcal{Y}$'dir.

Tanım 3.2; Eğer $\mathcal{X}$ fonksiyonu bir çarpım terimi x_i içeriyorsa, x_i $\mathcal{X}$ fonksiyonunun bir **implikant**ıdır (Imp). x_i bir minterm ise, x_i'ye $\mathcal{X}$ fonksiyonunun **minterm**i denir.

Tanım 3.3; Eğer *A* çarpımındaki bütün literaller *B* çarpımında da bulunuyorsa *A* *B*'nin **üst çarpım**ıdır.

Örnek 3.6; *A*= ($x_1.x_2$) ve *B*= ($x_1.x_2.x_4$) aynı fonksiyonun iki çarpım terimi olsun. *A*'nın bütün çarpımları (x_1 x_2), *B* de bulunduğu için *A*, *B*'nin üst çarpımıdır.

Tanım 3.4; *A* çarpım terimi, $\mathcal{X}$ fonksiyonunun bir implikantı olsun. Eğer $\mathcal{X}$ fonksiyonunun diğer implikantlarının hiçbiri, *A*'nın üst çarpımları değilse, $\mathcal{X}$ fonksiyonu için *A* **asal implikant**dır (*AI*).

Örnek 3.7; $\mathcal{X}_{(a,b,c)} = \{\neg a \neg bc, \neg abc, a\neg bc, abc, ab\neg c\}$ olsun. Fonksiyonun sadeleşmiş hali $\mathcal{X}_{(a,b,c)} = \{ c, ab, \neg bc \}$ implikantları olsun. İlk implikant "c" literali, "$\neg bc$" implikantında bulunduğu için "c" implikantı, "$\neg bc$" implikantının üst çarpımıdır. "c" Asal İmplikantdır ve "$\neg bc$"'yi kapsar.

Tanım 3.5; $\mathcal{X}$ fonksiyonunu temsil eden AI'lardan oluşan bir SOP ifadesi düşünelim. Eğer AI'lardan herhangi birisini çıkardığımızda fonksiyon değişiyorsa, bu SOP ifadesine **kapsama fonksiyonu** denir.

Tanım 3.6; Alternatif kapsama fonksiyonları içerisinde, en az sayıda çarpım terimi içerene **en küçük kapsama fonksiyonu** denir. En küçük kapsama fonksiyonları arasından en az sayıda literal içerene **kesin sonuç kapsama fonksiyonu** denir.

Tanım 3.7; $\mathcal{X}$ fonksiyonunun mintermlerinden biri x_i olsun. $\mathcal{X}$ fonksiyonunun AI'larından sadece birisi x_i'yi kapsıyorsa, x_i'ye **izole minterm** denir. x_i'yi kapsayan AI'ya **Asal İmplikant** (AI) denir (Sasao, 1999).

4.3. Yakın Sonuç Kapsama Algoritması

Bu tez çalışmasında bilgisayar bilimlerinin temel konusu olan lojik fonksiyonlar için iki seviyeli sadeleştirme algoritması geliştirilmiştir. SOP ifadelerini kesin sonuç sadeleştirmenin karmaşıklığı üssel olduğu için en fazla 100'e yakın çarpım terimi içeren fonksiyonlar sadeleştirilebilmektedir. Bu yüzden daha hızlı çalışan ve daha az bellek kullanan pratik sadeleştirme uygulamalarına ihtiyaç duyulmaktadır.

Yakın sonuç bulan kapsama algoritmalarında, çıkan sonucun fonksiyonun en sade hali olduğundan emin olamasak da hafıza ve zaman açısından büyük kazanç sağlamaktadır. Bu algoritmalar hızlı ve etkili çalışabilmek için farklı buluşsal yöntemler uygulamaktadır. Bu sebeple bütün lojik fonksiyonlarda en iyi olan algoritma yoktur. Farklı fonksiyon dosyaları farklı buluşsal yöntemlerle/algoritmalarla daha sade hale gelebilmektedir. Geliştirilen yakın sonuç kapsama algoritması Şekil 4.3'de verilmiştir.

```
PROGRAM YAKIN_SONUC (X, Y) //Sırasıyla ON ve OFF minterm kümeleri
BEGIN
  WHILE (X) DO
     A = X1; // X Kümesinin ilk elemanını A mintermine ata
     P = GENISLET(A, Y); // A mintermini OFF kümesiyle genislet
     R = AYIKLA (P); //P kümesinden kapsananları ayıkla
     AS = TUMLEYEN_AL(1-R); // Tamkümeden R kümesini çıkar
     FOR Q in AS DO // İmplikantlar(AS) arasından en iyisini bul
           S = SAYISI(X -Q); // ON kümesini sadeleştirme oranı
           IF (S<S2)   // eğer daha çok kapsarsa AI olur.
                 S2 = S;
                 AI = Q;
     ENDFOR;
     EAI = EAI + AI; // Bulunan AI sonuç kümesine ekleniyor
     X = X - AI; // X kümesinden kapsananlar çıkarılıyor
  ENDWHILE;
  RETURN EAI;
END
```

Şekil 4.3. Yakın sonuç kapsama algoritması kaba kodu

Yakın sonuç kapsama algoritması temelde 2 bölümden oluşmaktadır. Öncelikle bir mintermi kapsayan implikantlar oluşturulmakta, ardından en çok minterm kapsayan

implikant seçilmektedir. İmplikantların tespit edilmesi için bir minterm diğer OFF mintermleri ile karşılaştırılarak genişletilir. Genişletilen kümede gereksiz olan mintermler ayıklanarak sadece asal olanlar tespit edilir. Ardından OFF mintermlerle genişletilen ve asal hale getirilen küme, tamkümeden (**…**) çıkarılır. Böylece mintermi örten implikantlar kümesi belirlenmiş olur. Ardından ikinci aşamada implikantların kapsadığı ON mintermleri sayılır. En çok minterm kapsayan implikant AI olarak seçilir. AI'nın kapsadığı ON mintermleri çıkarılarak sadeleştirme işlemi ON Mintermler kümesi boş olasıya kadar devam eder.

Örnek 3.8; On mintermler kümesi $\mathcal{X}_{(abcd)}$ = {0, 2, 4, 5, 6, 10, 11, 13}, OFF mintermler kümesi $\mathcal{Y}_{(abcd)}$ = {1, 3, 7, 8, 9, 12, 14, 15} şeklinde verilen F fonksiyonunu yakın sonuç kapsama algoritmasını kullanarak sadeleştirelim.

$\mathcal{X}$ = {0000, 0010, 0100, 0101, 0110, 1010, 1011, 1101}
$\mathcal{Y}$ = {0001, 0011, 0111, 1000, 1001, 1100, 1110, 1111}

Sadeleştirmeye $\mathcal{X}$ kümesinin ilk elemanı olan "0000" minterminden başlanıyor. "0000" minterminin $\mathcal{Y}$ kümesiyle genişletilmesi Çizelge 4.8'de verilmiştir.

Çizelge 4.8. "0000" minterminin genişletilmesi ve ayıklanması

$\mathcal{Y}$	$\mathcal{P}$		R
0001	***1	Üst Çarpan	***1
0011	**11	***1 tarafından kapsandı	
0111	*111	***1 tarafından kapsandı	
1000	1***	Üst Çarpan	1***
1001	1**1	1*** tarafından kapsandı	
1100	11**	1*** tarafından kapsandı	
1110	111*	1*** tarafından kapsandı	
1111	1111	1*** tarafından kapsandı	

"0000" mintermini $\mathcal{Y}$ kümesiyle genişletmek için Denklem 4.2'de verilen formülden yararlanılır. Aynı değerdeki literaller yerine *, farklı değerdeki literaller yerine $\mathcal{Y}$ minterminin literali kullanılır. Genişletilmiş $\mathcal{P}$ kümesi elemanlarının birbirini kapsama durumunu Denklem 4.6 ve 4.7'de verilen formüllere göre kontrol edilir. İşlemler sonucunda genişletilmiş ve ayıklanmış iki elemanlı R = {***1, 1***} kümesi elde edilir. Ardından tamkümeden çıkarılarak R'nin tümleyeni alınır.

$\mathrm{AS} = \{**** \ominus R\} = \{(**** \ominus ***1) \ominus 1***\}$

$\mathrm{AS} = \{***0 \ominus 1***\} = \{0**0\}$

Tek implikant bulunduğu için doğrudan AI kümesine eklenir ve yeni bulunan AI'nın kapsadığı mintermleri $\mathcal{X}$ kümesinden çıkarılır.

AI = {0**0}

$\mathcal{X}_2 = \mathcal{X}_1 \ominus 0**0 = \{0101, 1010, 1011, 1101\}$

Sıradaki 0101 minterminin genişletilmesi Çizelge 4.9'da verilmiştir.

Çizelge 4.9. "0101" minterminin genişletilmesi ve ayıklanması

$\mathcal{Y}$	$\mathcal{P}$		R
0001	*0**	Üst Çarpan	*0**
0011	*01*	*0** tarafından kapsandı	
0111	**1*	Üst Çarpan	**1*
1000	10*0	*0** tarafından kapsandı	
1001	10**	*0** tarafından kapsandı	
1100	1**0	Üst Çarpan	1**0
1110	1*10	1**0 tarafından kapsandı	
1111	1*1*	**1* tarafından kapsandı	

$R = \{*0**, **1*, 1**0\}$

$\mathrm{AS} = \{**** \ominus R\} = \{((**** \ominus *0**) \ominus **1*) \ominus 1**0\}$

$\mathrm{AS} = \{((*1**) \ominus **1*) \ominus 1**0\}$

$\mathrm{AS} = \{(*10*) \ominus 1**0\}$

$\mathrm{AS} = \{010*, *101\}$

0101 mintermini kapsayan iki implikant bulundu. Hangi implikantın AI olacağına karar vermek için X kümesini ne kadar sadeleştirdiği kontrol edilir.

$\mathcal{X}_{3.1} = \mathcal{X}_2 \ominus \mathrm{AS}_1 = \mathcal{X}_2 \ominus 010* = \{1010, 1011, 1101\}$

$\mathcal{X}_{3.2} = \mathcal{X}_2 \ominus \mathrm{AS}_2 = \mathcal{X}_2 \ominus *101 = \{1010, 1011\}$

$\mathcal{X}_{3.1} \supset \mathcal{X}_{3.2}$ olduğundan, daha geniş kapsayan AS_2 implikantı AI olarak seçilir.

AI = {0**0, *101}

$\mathcal{X}_3 = \mathcal{X}_2 \ominus$ *101 = {1010, 1011}

Sıradaki 1010 minterminin genişletilmesi Çizelge 4.10'da verilmiştir.

Çizelge 4.10. "1010" minterminin genişletilmesi ve ayıklanması

$\mathcal{Y}$	$\mathcal{P}$		R
0001	0*01	Üst Çarpan	0*01
0011	0**1	Üst Çarpan	0**1
0111	01*1	0**1 tarafından kapsandı	
1000	**0*	Üst Çarpan	**0*
1001	**01	**0* tarafından kapsandı	
1100	*10*	**0* tarafından kapsandı	
1110	*1**	Üst Çarpan	*1**
1111	*1*1	*1** tarafından kapsandı	

R = {0*01, 0**1, **0*, *1**} = {0**1, **0*, *1**}

AS = {**** $\ominus R$} = {((**** $\ominus$ 0**1) $\ominus$ **0*) $\ominus$ *1**}

AS = {((1***, ***0) $\ominus$ **0*) $\ominus$ *1**}

AS = {(1*1*, **10) $\ominus$ *1**}

AS = {101*, *010}

1010 mintermini kapsayan iki implikant bulundu. Hangi implikantın AI olacağına karar vermek için X kümesini ne kadar sadeleştirdiği kontrol edilir.

$\mathcal{X}_{4.1} = \mathcal{X}_3 \ominus \text{AS}_1 = \mathcal{X}_3 \ominus$ 101* = { } = $\emptyset$

$\mathcal{X}_{4.2} = \mathcal{X}_3 \ominus \text{AS}_2 = \mathcal{X}_3 \ominus$ *010 = {1011}

$\mathcal{X}_{4.2} \supset \mathcal{X}_{4.1}$ olduğundan dolayı fonksiyonu daha çok sadeleştiren AS_1 implikantı AI olarak seçilir.

AI = {0**0, *101, 101*}

$\mathcal{X}_4 = \mathcal{X}_3 \ominus$ 101* = $\emptyset$

$\mathcal{X}_{(abcd)}$ = {0**0, *101, 101*} = {¬a¬d, b¬cd, a¬bc}

Sonuç olarak yakın sonuç kapsama algoritmasına göre 8 mintermden oluşan $\mathcal{X}_{(abcd)}$ fonksiyonu 3 AI tarafından kapsanmıştır.

4.4. Kesin Sonuç Kapsama Algoritması

Kesin sonuç kapsama algoritmasının ilk aşaması olan implikantların oluşturulması bölümü yakın sonuç kapsama algoritması ile aynıdır. İki algoritma arasındaki fark ikinci aşama olan AI'nın tespit edilmesi kısmıdır.

Bir mintermi kapsayan implikantlar arasından hangisinin seçileceği karmaşıklığı yüksek bir problemdir. Kesin sonuç bulan sadeleştirme algoritmalarının zaman ve hafıza karmaşıklığının yüksek olmasının temel sebebi AI'nın tespit edilme zorluğudur. Geliştirilen kesin sonuç kapsama algoritmasının kaba kodu Şekil 4.4'de verilmiştir.

```
PROGRAM KESIN_SONUC (X, Y) //Sırasıyla ON ve OFF minterm kümeleri
BEGIN
   FOR IM IN (X) DO
       IM = X1; // X Kümesinin elemanı İşlenen Minterm (IM) yap
       P = GENISLET(IM, Y); // A mintermini OFF kümesiyle genislet
       R = AYIKLA (P); //P kümesinden kapsananları ayıkla
       AS= TUMLEYEN_AL(1-R);//Tamkümeden R kümesini çıkar=AI Adaylar
       IF (ELEMAN_SAY(Q)==1) // Tek Imp varsa AI ekle
             AI = AI + AS1); // Bulunan AI sonuç kümesine ekleniyor
             X = X - AS(1); // X kümesinden kapsananlar çıkarılıyor
             Kayıt_Kontrol(AS(1)); Kayıt_Kontrol Prosedürü çağırılıyor

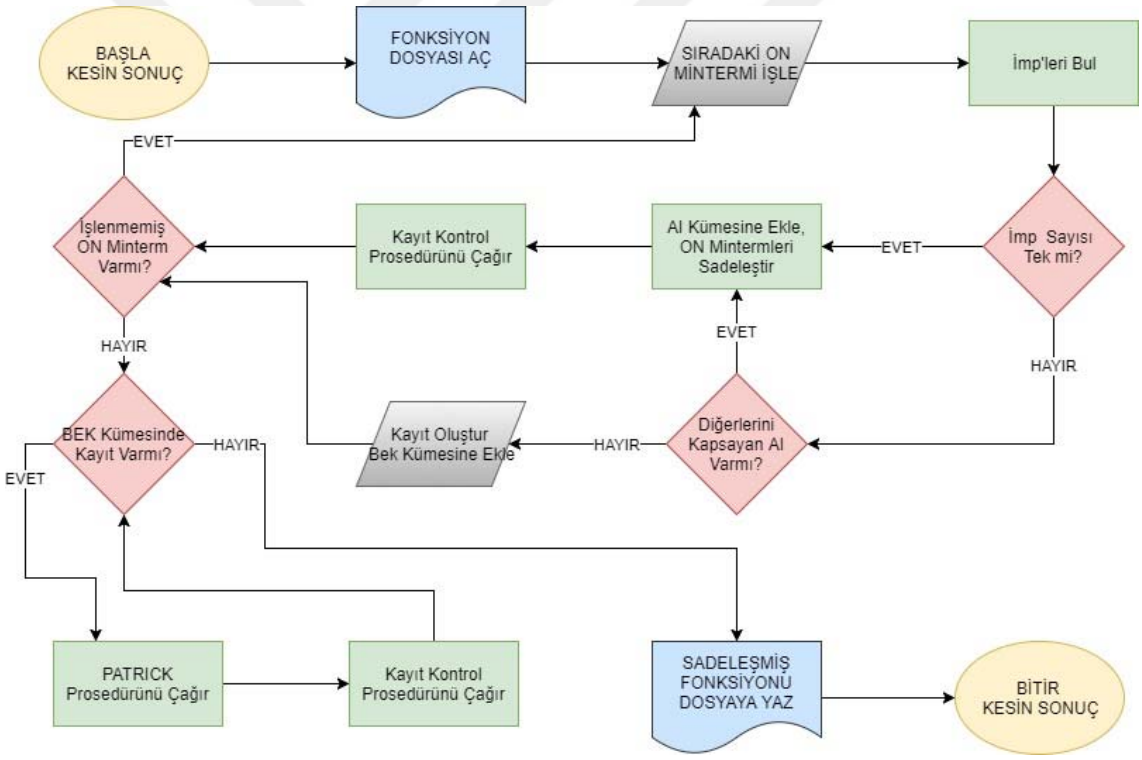
       ELSE  // Birdençok Imp varsa aşağıdaki işlemleri uygula
             FOR all q IN (AS) DO //Diğer Imp.leri kapsayan AI varmı?
                   Karsilastir (q, (all q in AS));
             IF q ⊃ (all q in Q)     // Diğer Imp.leri kapsayan AI varsa
                   AI = AI + q; // Bulunan AI ekleniyor
                   X = X - q; // kapsananlar çıkarılıyor
                   Kayıt_Kontrol(q); Kayıt_Kontrol çağırılıyor
             ELSE          //Diğer Imp.leri kapsayan AI yoksa
                   K = IM + AS;   //Yeni Kayıt(K)=Min+Imp.'s oluştur.
                   BEK = BEK + K;  //K'yı Bekleyenler Kümesine Ekle
   ENDFOR;

   IF (BEK) DO  //Bekleyenler Kümesinde Eleman varsa
       PETRICK (BEK);   //Petrick prosedürü ile minimum örtme yap.
   ENDIF;

   RETURN AI;
END
```

Şekil 4.4. Kesin sonuç kapsama algoritması kaba kodu

Kesin sonuç kapsama algoritması 3 ana bölümden oluşmaktadır. İlk bölümde kapsaması zor izole mintermler, tek implikant'la kapsanarak *AI* kümesine kaydedilir. Birden çok implikantı olan mintermlerde diğer implikantları kapsayan *Üst çarpım* ifadesi olup olmadığı kontrol edilir. *Üst çarpım* implikantı olan mintermlerde kapsanarak, *AI* kümesine kaydedilir. Eğer bir mintermin hem çok implikantı varsa, hemde implikantlarının hiçbiri *Üst çarpım* implikant özelliği taşımıyorsa, bu durumda mintermin işlenmesi ertelenir. *İşlenen Minterm* (*IM*) ve bulunan implikantları (Imp_1, Imp_2,…Imp_n) içeren *Kayıt (K)* oluşturulur ve *Bekleyenler Kümesi* (*BEK*)'e *K* eklenir. Ardından diğer mintermlerle aynı işleme devam edilir. Böylece kapsaması kolay olan mintermler ilk aşamada işlenir ve algoritmanın karmaşıklığını arttıran 2. ve 3. bölümlerde bu mintermlerle ilgilenmeye gerek kalmaz. Kesin sonuç kapsama algoritmasının akış diyagramı Şekil 4.5'da verilmiştir.



Şekil 4.5. Kesin sonuç kapsama algoritması akış diyagramı

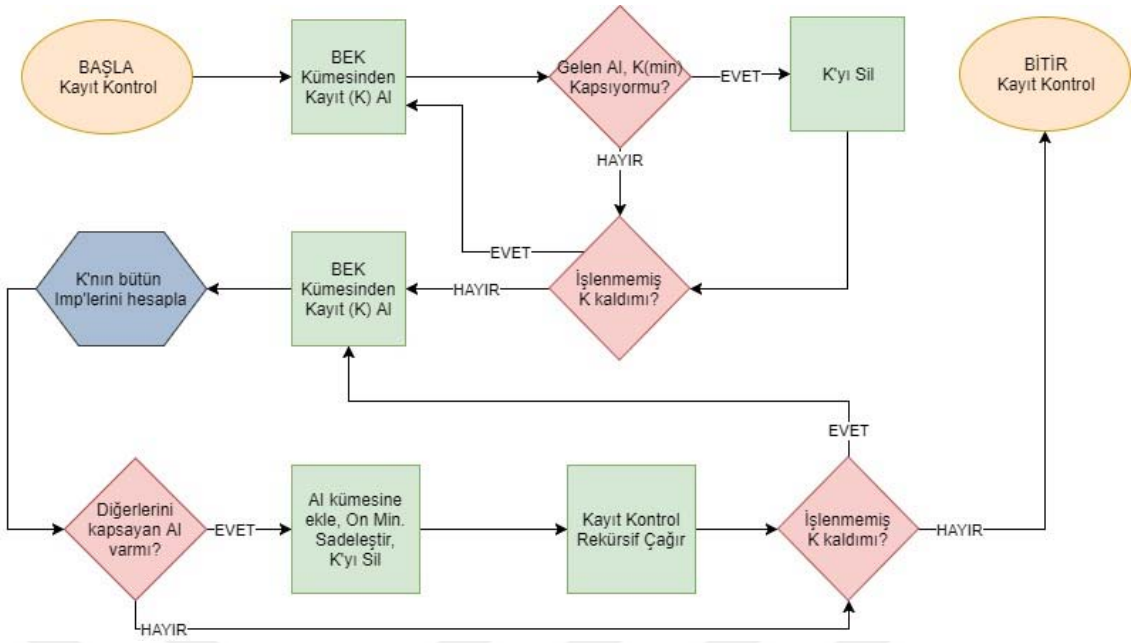
Kesin sonuç kapsama algoritmasının 2. bölümünde kapsaması zor olan mintermler için oluşturulmuş *BEK* kümesi işlenir. Bu bölüm her AI tanımlandığında otomatik olarak tetiklenir. Çünkü bir AI tanımlandığında *BEK* kümesinde bulunan *Kayıt*ların kapsanma durumu değişebilir.

Bu algoritmada öncelikle tanımlanan AI'nın, *BEK* kümesinde bulunan *K*'ların mintermleri kapsayıp kapsamadığı kontrol edilir. Eğer kapsıyorsa geçerli *K*, *BEK* kümesinden silinir. Eğer kapsamıyorsa *K* girdilerinin implikantları tekrar kontrol edilir. Çünkü kapsanan On Minterm Kümesinin son durumuna göre bazı implikantlar *Üst Çarpım* durumuna gelmiş olabilir. Eğer yeni bir AI tespit edilirse, AI kümesine eklenir, On Minterm kümesi sadeleştirilir ve programın ikinci bölümü yeni bulunan AI'ya göre tekrar çağırılır. Dolayısıyla bu algoritma kendini tekrar çağırdığı için rekürsif bir algoritmadır. *BEK* kümesini işlediği için zaman ve hafıza karmaşıklığı yüksek olan bu bölümün rekürsif olması büyük avantaj sağlamaktadır. *Kayıt* kontrol algoritmasının kaba kodu Şekil 4.6'de, akış diyagramı Şekil 4.7'de verilmiştir.

```
PROSEDÜR KAYIT_KONTROL(AI) //AI: Yeni tanımlanan Asal Implikant
BEGIN
   FOR ALL K IN (BEK) DO
       IF AI ⊃ K(min)
             Sil K;
   END FOR;

   FOR K IN (BEK) DO
       FOR ALL Impᵢ IN (K_Imp) DO
             IF Impᵢ ⊃ (all Imp in K) // Diğer Imp.leri kapsayan AI varsa
                   AI = AI + Impᵢ; // Bulunan AI ekleniyor
                   X = X - Impᵢ; // kapsananlar çıkarılıyor
                   Kayıt_Kontrol(Impᵢ); Kayıt_Kontrol çağırılıyor
                   END FOR;
       END FOR;
   ENDFOR;
END
```

Şekil 4.6. *Kayıt* kontrol algoritması kaba kodu



Şekil 4.7. *Kayıt* kontrol algoritması akış diyagramı

Bazı mintermlerin bekletilip her *AI* tanımlanmasından sonra tekrar tekrar kontrol edilmesine rağmen kapsama yapılamıyorsa bu mintermler için *Petrick* yöntemi kullanılır. Bu yöntemde mintermi kapsayan bütün *Imp*'ler ve bu *Imp*'lerin kapsadığı *ON_Min*'ler bulunur. *Imp*'lerin satırlara, *On_Min*'lerin sütunlara yerleştirildiği bir tablo oluşturulur. Satırlardaki *Imp*'ler etiketlenir. Bütün mintermlerin kapsanması istendiği için bir mintermi kapsayan *Imp* alternatifleri *"veya"* işlemi ile parantez içine alınır. Ardından diğer mintermler içinde aynı işlem yapılır ve bu ifadeler *"ve"* işlemine tabi tutulur. Sonuçta elde edilen POS ifadesi en küçük SOP ifadesine indirgenir (Arslan ve Sertbaş, 2002).

Örnek 3.9; On mintermler kümesi $\mathcal{X}_{(abcd)}$ = {0, 1, 2, 4, 5, 7, 9, 12}, OFF mintermler kümesi $\mathcal{Y}_{(abcd)}$ = {3, 6, 8, 10, 11, 13, 14, 15} şeklinde verilen F fonksiyonunu kesin sonuç kapsama algoritmasını kullanarak sadeleştirelim.

$\mathcal{X}_1$ = {0000, 0001, 0010, 0100, 0101, 0111, 1001, 1100}
$\mathcal{Y}$ = {0011, 0110, 1000, 1010, 1011, 1101, 1110, 1111}

Sadeleştirmeye $\mathcal{X}_1$ kümesinin ilk elemanı olan "0000" minterminden başlanır. "0000" minterminin $\mathcal{Y}$ kümesiyle genişletilmesi Çizelge 4.11'de verilmiştir.

Çizelge 4.11. "0000" minterminin genişletilmesi ve ayıklanması

$\mathcal{Y}$	$\mathcal{P}_1$		R_1
0011	**11	Üst Çarpan	**11
0110	*11*	Üst Çarpan	*11*
1000	1***	Üst Çarpan	1***
1010	1*1*	1*** tarafından kapsandı	
1011	1*11	1*** tarafından kapsandı	
1101	11*1	1*** tarafından kapsandı	
1110	111*	1*** tarafından kapsandı	
1111	1111	1*** tarafından kapsandı	

"0000" mintermini $\mathcal{Y}$ kümesiyle genişletmek için Denklem 4.2'de verilen formülden yararlanılır. Aynı değerdeki literaller yerine *, farklı değerdeki literaller yerine $\mathcal{Y}$ minterminin literali kullanılır. Genişletilmiş $\mathcal{P}_1$ kümesi elemanlarının birbirini kapsama durumunu Denklem 4.6 ve 4.7'de verilen formüllere göre kontrol edilir. İşlemler sonucunda genişletilmiş ve ayıklanmış üç elemanlı R_1 = {**11, *11*, 1***} kümesi elde edilir. Ardından tamkümeden çıkararak R_1'nin tümleyeni alınır.

$AS_1 = \{**** \ominus R_1\} = \{((**** \ominus **11) \ominus *11*) \ominus 1***\}$
$AS_1 = \{(**0*, ***0) \ominus *11*) \ominus 1***\}$
$AS_1 = \{(**0*, *0*0, **00) \ominus 1***\} = \{0*0*, 00*0, 0*00\}$

Aynı mintermi kapsayan üç implikant {0*0*, 00*0, 0*00} bulunduğu için aralarında diğer implikantların kapsadığı ON mintermlerini kapsayan bir implikant olup olmadığı kontrol edilir.

$\mathcal{X}_{2.1} = \mathcal{X}_1 \ominus 0*0* = \{\mathbf{0010}, 0111, 1001, 1100\}$
$\mathcal{X}_{2.2} = \mathcal{X}_1 \ominus 00*0 = \{0001, \mathbf{0100}, 0101, 0111, 1001, 1100\}$
$\mathcal{X}_{2.3} = \mathcal{X}_1 \ominus 0*00 = \{0001, \mathbf{0010}, 0101, 0111, 1001, 1100\}$

$\mathcal{X}_{2.1}$ kümesi, $\mathcal{X}_{2.3}$'ün Üst Çarpanıdır ve $\mathcal{X}_{2.3}$'ün kapsadığı bütün ON Mintermlerini kapsar. Bu sebeple 0*00 AS kümesinden çıkarılır. Fakat $\mathcal{X}_{2.1}$ ve $\mathcal{X}_{2.2}$'nin içerikleri farklı olduğu için karar vermek mümkün değildir. Bu sebeple 0000 mintermi için bir *Kayıt(K)* oluşturulur ve *Bekleyenler Kümesi* (*BEK*)'e eklenir.

K_1= {0000, (0*0*, 00*0)}

BEK = {K_1}

$\mathcal{X}_2$ = {~~0000~~, 0001, 0010, 0100, 0101, 0111, 1001, 1100}

Kapsama işlemine sıradaki 0001 mintermi ile devam edilir. Çizelge 4.12'de gösterildiği gibi genişletilmiş ve ayıklanmış üç elemanlı R_2 = {**1*, 1**0, 11**} kümesi elde edilir. Ardından tamkümeden çıkararak R_2'nin tümleyeni alınır.

Çizelge 4.12. "0001" minterminin genişletilmesi ve ayıklanması

y	$\mathcal{P}_2$		R_2
0011	**1*	Üst Çarpan	**1*
0110	*110	**1* tarafından kapsandı	
1000	1**0	Üst Çarpan	1**0
1010	1*10	1**0 tarafından kapsandı	
1011	1*1*	**1* tarafından kapsandı	
1101	11**	Üst Çarpan	11**
1110	1110	11** tarafından kapsandı	
1111	111*	**1* tarafından kapsandı	

$AS_2 = \{**** \ominus R_2\} = \{((**** \ominus **1*) \ominus 1**0) \ominus 11**\}$

$AS_2 = \{(**0* \ominus 1**0) \ominus 11**\}$

$AS_2 = \{(0*0*, **01) \ominus 11**\} = \{0*0*, 000*, 0*01, *001\}$

Aynı mintermi kapsayan dört implikant bulunduğu için aralarında diğer implikantların kapsadığı ON mintermlerini kapsayan bir implikant olup olmadığı kontrol edilir.

$\mathcal{X}_{3.1} = \mathcal{X}_2 \ominus 0*0* = \{0010, 0111, \mathbf{1001}, 1100\}$

$\mathcal{X}_{3.2} = \mathcal{X}_2 \ominus 000* = \{0010, 0100, 0101, 0111, 1001, 1100\}$

$\mathcal{X}_{3.3} = \mathcal{X}_2 \ominus 0*01 = \{0000, 0010, 0100, 0111, 1001, 1100\}$

$\mathcal{X}_{3.4} = \mathcal{X}_2 \ominus *001 = \{\mathbf{0000}, 0010, \mathbf{0100}, \mathbf{0101}, 0111, 1100\}$

$\mathcal{X}_{3.1}$ kümesi, $\mathcal{X}_{3.2}$ ve $\mathcal{X}_{3.3}$'ün Üst Çarpanıdır ve $\mathcal{X}_{3.2}$ ve $\mathcal{X}_{3.3}$'ün kapsadığı bütün ON Mintermlerini kapsar. Bu sebeple 0*00 AS kümesinden çıkarılır. Fakat $\mathcal{X}_{3.1}$ ve $\mathcal{X}_{3.4}$'ün içerikleri farklı olduğu için karar vermek mümkün değildir. Bu sebeple 0001 mintermi için bir *K* oluşturulur ve *BEK*'e eklenir.

K_2= {0001, (0*0*, *001)}

BEK = {K_1, K_2}

$\mathcal{X}_3$ = {~~0000~~, ~~0001~~, 0010, 0100, 0101, 0111, 1001, 1100}

Kapsama işlemine sıradaki 0010 mintermi ile devam edilir. Çizelge 4.13'de gösterildiği gibi genişletilmiş ve ayıklanmış dört elemanlı R_3 = {***1, *1**, 1*0*, 1***} kümesi elde edilir. Ardından tamkümeden çıkararak R_3'ün tümleyeni alınır.

Çizelge 4.13. "0010" minterminin genişletilmesi ve ayıklanması

y	$\mathcal{P}_3$		R_3
0011	***1	Üst Çarpan	***1
0110	*1**	Üst Çarpan	*1**
1000	1*0*	Üst Çarpan	1*0*
1010	1***	Üst Çarpan	1***
1011	1**1	***1 tarafından kapsandı	
1101	1101	***1 tarafından kapsandı	
1110	11**	*1** tarafından kapsandı	
1111	11*1	***1 tarafından kapsandı	

$AS_3 = \{**** \ominus R_3\} = \{(((**** \ominus ***1) \ominus *1**) \ominus 1*0*) \ominus 1***\}$

$AS_3 = \{((***0 \ominus *1**) \ominus 1*0*) \ominus 1***\}$

$AS_3 = \{*0*0 \ominus 1*0*) \ominus 1***\}$

$AS_3 = \{(00*0, *010) \ominus 1***\} = \{00*0, 0010\} = \{00*0\}$

Aynı mintermi kapsayan tek implikant bulunduğu için 00*0 Asal İmplikant (AI)'dır ve AI kümesine eklenir. Ardından $\mathcal{X}$ kümesi yeni tanımlanan AI'ya göre sadeleştirilir ve yeni bir AI tanımlandığı için *BEK* kümesindeki K'ların durumu kontrol edilir.

AI = {00*0}

$\mathcal{X}_4 = \mathcal{X}_3 - 00*0$ = {~~0000~~, ~~0001~~, 0010, 0100, 0101, 0111, 1001, 1100} − 00*0

$\mathcal{X}_4$ = {~~0001~~, 0100, 0101, 0111, 1001, 1100}

BEK = {K_1, K_2}

i) K_1'in kontrol edilmesi

K_1= {0000, (0*0*, 00*0)}

K_1'in mintermi yeni bulunan AI tarafından kapsandığı için K_1 silinir.

$K_{1(min)} \ominus$ 00*0 = $\emptyset$ olduğundan

BEK= *BEK* – K_1 = {K_2}

ii) K_2'in kontrol edilmesi

K_2= {0001, (0*0*, *001)}

$K_{2(min)} \ominus$ 00*0 = 0001 $\ominus$ 00*0 = 0001 olduğundan yeni bulunan AI(00*0), K_2'nin mintermini (0001) kapsamaz. Ardından $K_{2(Imp)}$ implikantları $\mathcal{X}$ kümesine göre kontrol edilir.

$\mathcal{X}_A = \mathcal{X}_4 \ominus$ 0*0* = {0111, **1001**, 1100}

$\mathcal{X}_B = \mathcal{X}_4 \ominus$ *001 = {**0100**, **0101**, 0111, 1100}

$\mathcal{X}_A \not\subset \mathcal{X}_B$ veya $\mathcal{X}_A \not\supset \mathcal{X}_B$ veya $\mathcal{X}_A \not\equiv \mathcal{X}_B$ olduğundan dolayı K_2 değişmeden kalır.

BEK = {K_2}

K_2 = {0001, (0*0*, *001)}

$\mathcal{X}_4$ = {~~0001~~, 0100, 0101, 0111, 1001, 1100}

Kapsama işlemine sıradaki 0100 mintermi ile devam edilir. Çizelge 4.14'de gösterildiği gibi genişletilmiş ve ayıklanmış üç elemanlı R_4 = {**1*, 10**, 1**1} kümesi elde edilir. Ardından tamkümeden çıkarılarak R_4'ün tümleyeni alınır.

Çizelge 4.14. "0100" minterminin genişletilmesi ve ayıklanması

y	P_4		R_4
0011	*011	Üst Çarpan	*011
0110	**1*	Üst Çarpan (***011'in üst çarpanı**)	**1*
1000	10**	Üst Çarpan	10**
1010	101*	10** tarafından kapsandı	
1011	1011	10** tarafından kapsandı	
1101	1**1	Üst Çarpan	1**1
1110	1*1*	**1* tarafından kapsandı	
1111	1*11	**1* tarafından kapsandı	

$AS_4 = \{**** \ominus R_4\} = \{((**** \ominus **1*) \ominus 10**) \ominus 1**1\}$

$AS_4 = \{(**0* \ominus 10**) \ominus 1**1\}$

$AS_4 = \{(0*0*, *10*) \ominus 1**1\}$

$AS_4 = \{0*0*, 0*00, 010*, *100\}$

$\mathcal{X}_{4.1} = \mathcal{X}_4 \ominus 0*0* = \{0111, 1001, 1100\}$

$\mathcal{X}_{4.2} = \mathcal{X}_4 \ominus 0*00 = \{\mathbf{0001, 0101}, 0111, 1001, 1100\}$

$\mathcal{X}_{4.3} = \mathcal{X}_4 \ominus 010* = \{\mathbf{0001}, 0111, 1001, 1100\}$

$\mathcal{X}_{4.4} = \mathcal{X}_4 \ominus *100 = \{\mathbf{0001, 0101}, 0111, 1001\}$

$\mathcal{X}_{4.1}$ kümesi, $\mathcal{X}_{4.2}$ ve $\mathcal{X}_{4.3}$'ün Üst Çarpanıdır ve $\mathcal{X}_{4.2}$ ve $\mathcal{X}_{4.3}$'ün kapsadığı bütün ON Mintermlerini kapsar. Bu sebeple 0*00 ve 010* implikantları AS kümesinden çıkarılır. Fakat $\mathcal{X}_{4.1}$ ve $\mathcal{X}_{4.4}$'ün içerikleri farklı olduğu için karar vermek mümkün değildir. Bu sebeple 0100 mintermi için bir *K* oluşturulur ve *BEK*'e eklenir.

K_3= {0100, (0*0*, *100)}

BEK = {K_2, K_3}

$\mathcal{X}_5$ = {~~0001~~, ~~0100~~, 0101, 0111, 1001, 1100}

Kapsama işlemine sıradaki 0101 mintermi ile devam edilir. Çizelge 4.15'de gösterildiği gibi genişletilmiş ve ayıklanmış üç elemanlı R_5 = {*01*, **10, 1***} kümesi elde edilir. Ardından tamkümeden çıkararak R_5'in tümleyeni alınır.

Çizelge 4.15. "0101" minterminin genişletilmesi ve ayıklanması

$\mathcal{Y}$	$\mathcal{P}_5$		R_5
0011	*01*	Üst Çarpan	*01*
0110	**10	Üst Çarpan	**10
1000	10*0	Üst Çarpan	10*0
1010	1010	10*0 tarafından kapsandı	
1011	101*	*01* tarafından kapsandı	
1101	1***	Üst Çarpan (**10*0 kapsar**)	1***
1110	1*10	1*** tarafından kapsandı	
1111	1*1*	1*** tarafından kapsandı	

$AS_5 = \{**** \ominus R_5\} = \{((**** \ominus *01*) \ominus **10) \ominus 1***\}$

$AS_5 = \{((*1**, **0*) \ominus **10) \ominus 1***\}$

$AS_5 = \{(*10*, *1*1, **0*, **01) \ominus 1***\}$

$AS_5 = \{010*, 01*1, 0*0*, 0*01\}$

$\mathcal{X}_{5.1} = \mathcal{X}_5 \ominus 010* = \{\mathbf{0001}, 0111, 1001, 1100\}$

$\mathcal{X}_{5.2} = \mathcal{X}_5 \ominus 01*1 = \{\mathbf{0001, 0100}, 1001, 1100\}$

$\mathcal{X}_{5.3} = \mathcal{X}_5 \ominus 0*0* = \{0111, 1001, 1100\}$

$\mathcal{X}_{5.4} = \mathcal{X}_5 \ominus 0*01 = \{\mathbf{0100}, 0111, 1001, 1100\}$

$\mathcal{X}_{5.3}$ kümesi, $\mathcal{X}_{5.1}$ ve $\mathcal{X}_{5.4}$'ün Üst Çarpanıdır ve $\mathcal{X}_{5.1}$ ve $\mathcal{X}_{5.4}$'ün kapsadığı bütün ON Mintermlerini kapsar. Bu sebeple 010* ve 0*01 implikantları AS kümesinden çıkarılır. Fakat $\mathcal{X}_{5.2}$ ve $\mathcal{X}_{5.3}$'ün içerikleri farklı olduğu için karar vermek mümkün değildir. Bu sebeple 0101 mintermi için bir *K* oluşturulur ve *BEK*'e eklenir.

K_4= {0101, (01*1, 0*0*)}

BEK = {K_2, K_3, K_4}

$\mathcal{X}_6$ = {~~0001~~, ~~0100~~, ~~0101~~, 0111, 1001, 1100}

Kapsama işlemine sıradaki 0111 mintermi ile devam edilir. Çizelge 4.16'da gösterildiği gibi genişletilmiş ve ayıklanmış üç elemanlı R_6 = {*0**, ***0, 1***} kümesi elde edilir. Ardından tamkümeden çıkarılarak R_6'nin tümleyeni alınır.

Çizelge 4.16. "0111" minterminin genişletilmesi ve ayıklanması

$\mathcal{Y}$	$\mathcal{P}_6$		R_6
0011	*0**	Üst Çarpan	*0**
0110	***0	Üst Çarpan	***0
1000	1000	*0** tarafından kapsandı	
1010	10*0	*0** tarafından kapsandı	
1011	10**	*0** tarafından kapsandı	
1101	1*0*	Üst Çarpan	1*0*
1110	1**0	***0 tarafından kapsandı	
1111	1***	Üst Çarpan (**1*0* kapsar**)	1***

$AS_6 = \{**** \ominus R_6\} = \{((**** \ominus *0**) \ominus ***0) \ominus 1***\}$

$AS_6 = \{((*1** \ominus ***0) \ominus 1***\}$

$AS_6 = \{*1*1 \ominus 1***\} = \{01*1\}$

0111 mintermini kapsayan tek implikant bulunduğu için 01*1 AI'dır ve AI kümesine eklenir. Ardından $\mathcal{X}$ kümesi yeni tanımlanan AI'ya göre sadeleştirilir ve yeni bir AI tanımlandığı için *BEK* kümesindeki K'ların durumu kontrol edilir.

AI = {00*0, 01*1}
$\mathcal{X}_7 = \mathcal{X}_6 - 01*1$ = {~~0001~~, ~~0100~~, ~~0101~~, 0111, 1001, 1100} – 01*1
$\mathcal{X}_7$ = {~~0001~~, ~~0100~~, 1001, 1100}
BEK = {K_2, K_3, K_4}

i) K_2'in kontrol edilmesi
K_2= {0001, (0*0*, *001)}

$K_{2(min)} \ominus$ 01*1 = 0001 $\ominus$ 01*1 = 0001 olduğundan yeni bulunan AI(01*1), K_2'nin mintermini (0001) kapsamaz. Ardından $K_{2(Imp)}$ implikantları $\mathcal{X}$ kümesine göre kontrol edilir.

$\mathcal{X}_A = \mathcal{X}_7 \ominus$ 0*0* = {**1001**, 1100}
$\mathcal{X}_B = \mathcal{X}_7 \ominus$ *001 = {**0100**, 1100}

$\mathcal{X}_A \not\subset \mathcal{X}_B$ veya $\mathcal{X}_A \not\supset \mathcal{X}_B$ veya $\mathcal{X}_A \not\equiv \mathcal{X}_B$ olduğundan dolayı K_2 değişmeden kalır.

ii) K_3'ün kontrol edilmesi
K_3= {0100, (0*0*, *100)}

$K_{3(min)} \ominus$ 01*1 = 0100 $\ominus$ 01*1 = 0100 olduğundan yeni bulunan AI(01*1), K_3'ün mintermini (0100) kapsamaz. Ardından $K_{3(Imp)}$ implikantları $\mathcal{X}$ kümesine göre kontrol edilir.

$\mathcal{X}_A = \mathcal{X}_7 \ominus$ 0*0* = {1001, **1100**}
$\mathcal{X}_B = \mathcal{X}_7 \ominus$ *100 = {**0001**, 1001}

$\mathcal{X}_A \not\subset \mathcal{X}_B$ veya $\mathcal{X}_A \not\supset \mathcal{X}_B$ veya $\mathcal{X}_A \not\equiv \mathcal{X}_B$ olduğundan dolayı K_3 değişmeden kalır.

iii) K_4'ün kontrol edilmesi

K_4= {0101, (01*1, 0*0*)}

K_4'ün mintermi yeni bulunan AI (01*1) tarafından kapsandığı için K_4 silinir. $K_{4(min)} \ominus$ 01*1 = ∅ olduğundan, *BEK* ve $\mathcal{X}_7$ kümelerinin durumu aşağıdaki gibi olur.

BEK = *BEK* – K_4 = {K_2, K_3},

$\mathcal{X}_7$ = {~~0001~~, ~~0100~~, 1001, 1100}

Kapsama işlemine sıradaki 1001 mintermi ile devam edilir. Çizelge 4.17'de gösterildiği gibi genişletilmiş ve ayıklanmış üç elemanlı R_7 = {***0, **1*, *1**} kümesi elde edilir. Ardından tamkümeden çıkarılarak R_7'nin tümleyeni alınır.

Çizelge 4.17. "1001" minterminin genişletilmesi ve ayıklanması

$\mathcal{Y}$	$\mathcal{P}_7$		R_7
0011	0*1*	Üst Çarpan	0*1*
0110	0110	0*1* tarafından kapsandı	
1000	***0	Üst Çarpan	***0
1010	**10	***0 tarafından kapsandı	
1011	**1*	Üst Çarpan (**0*1* kapsar**)	**1*
1101	*1**	Üst Çarpan	*1**
1110	*110	***0 tarafından kapsandı	
1111	*11*	**1* tarafından kapsandı	

$AS_7 = \{**** \ominus R_7\} = \{((**** \ominus ***0) \ominus **1*) \ominus *1**\}$

$AS_7 = \{(***1 \ominus **1*) \ominus *1**\}$

$AS_7 = \{**01 \ominus *1**\} = \{*001\}$

1001 mintermini kapsayan tek implikant bulunduğu için *001 AI'dır ve AI kümesine eklenir. Ardından $\mathcal{X}$ kümesi yeni tanımlanan AI'ya göre sadeleştirilir ve yeni bir AI tanımlandığı için *BEK* kümesindeki K'ların durumu kontrol edilir.

AI = {00*0, 01*1, *001}

$\mathcal{X}_8 = \mathcal{X}_7 - *001$ = {~~0001~~, ~~0100~~, 1001, 1100} – *001

$\mathcal{X}_8$ = {~~0100~~, 1100}

BEK = {K_2, K_3}

i) K_2'in kontrol edilmesi

K_2= {0001, (0*0*, *001)}

$K_{2(min)} \ominus$ *001 = 0001 $\ominus$ *001 = $\emptyset$ olduğundan yeni bulunan AI(*001), K_2'nin mintermini (0001) kapsar. Bu sebeple K_2 silinir.

BEK = *BEK* – K_2 = {K_2, K_3} – K_2 = {K_3}

ii) K_3'ün kontrol edilmesi

K_3= {0100, (0*0*, *100)}

$K_{3(min)} \ominus$ *001 = 0100 $\ominus$ *001 = 0100 olduğundan yeni bulunan AI(*001), K_3'ün mintermini (0100) kapsamaz. Ardından $K_{3(Imp)}$ implikantları $\mathcal{X}$ kümesine göre kontrol edilir.

$\mathcal{X}_A = \mathcal{X}_8 \ominus 0*0* = \{\mathbf{1100}\}$

$\mathcal{X}_B = \mathcal{X}_8 \ominus *100 = \{\emptyset\}$

$\mathcal{X}_A \supset \mathcal{X}_B$ olduğundan *100 AI'dır. Yeni belirlenen AI, AI kümesine eklenir ve ON Mintermler kümesi sadeleştirilir. K_3 *BEK* kümesinden silinir.

AI = AI + *100 = {00*0, 01*1, *001, *100},

$\mathcal{X}_9 = \mathcal{X}_8 \ominus$ *100 = {~~0100~~, 1100} $\ominus$ *100 = $\emptyset$,

BEK = *BEK* – K_3 = {$\emptyset$}.

İşlemler sonucunda 8 mintermden oluşan $\mathcal{X}$ fonksiyonu tamamen kapsanmış ve 4 AI tespit edilmiştir. *BEK* kümesinde bekleyen hiçbir *Kayıt* kalmamıştır.

$\mathcal{X}_{(abcd)}$ = {0, 1, 2, 4, 5, 7, 9, 12},

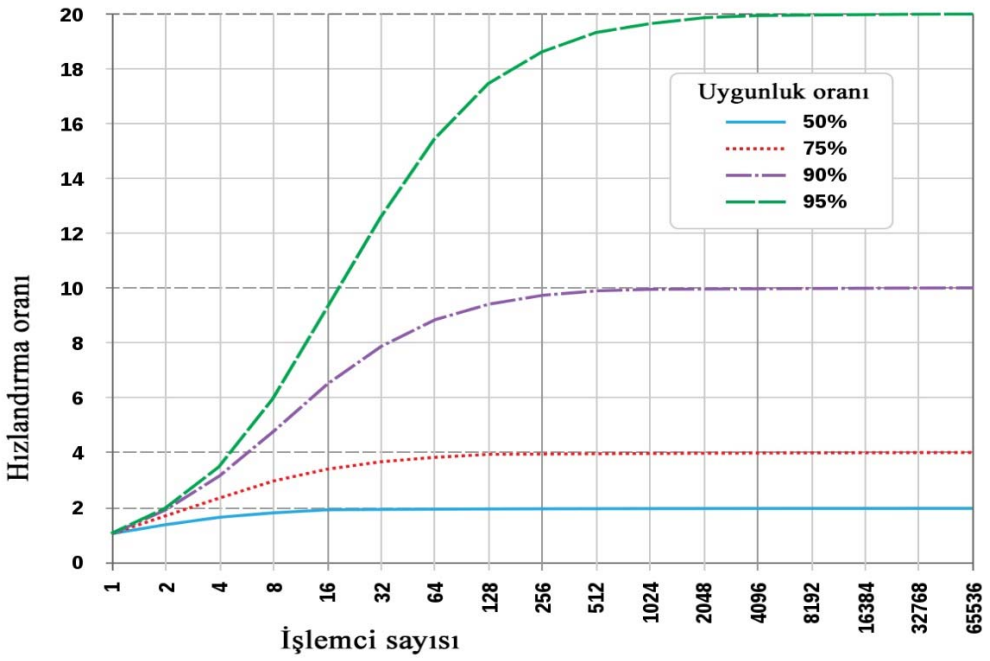
$AI_{(\mathcal{X})}$ = {00*0, 01*1, *001, *100} = { ¬a¬b¬d, ¬abd, ¬b¬cd, b¬c¬d }.

4.5. Algoritmaların Paralel Programlamaya Uyarlanması

Günümüzde bilgisayar işlemcileri çok çekirdekli mimaride üretilmektedir. Programların işlemci gücünü gerçekten kullanabilmesi için algoritmalarının paralel programlamaya uygun olması gerekmektedir. Paralel programlamanın mantığı böl ve yönet sisteminden gelmektedir. Yapılan iş ya da işlenen veri parçalara ayrılarak farklı işlemcilerde işlenir ve ardından birleştirilir. Farklı işlemcilerde aynı anda programın çalışması zaman kazandırsa da bölme ve birleştirme görevi zaman kaybettirmektedir. Amdahl kanununa göre bir algoritmanın işlemci sayısına göre hızlanma oranı Denklem 4.11 de verilmiştir (Hill ve Marty, 2008). Denklemde verilen "S" hızlanma oranını, "f" ise algoritmanın hızlandırılan bölümünü temsil etmektedir.

$$\text{Hızlanma}(f, S) = \frac{1}{(1-f)+\frac{f}{S}} \tag{4.11}$$

Ayrıca kullanılan algoritmaların paralel programlamaya uygunluk oranı gerçekleşebilecek hızlandırma oranını doğrudan etkilemektedir. Şekil 4.8'de farklı algoritmaların işlemci sayısına göre hızlanma oranlarını göstermektedir (Hill ve Marty, 2008). Şekil 4.8'de paralel programlamaya uygunluk oranı %50 olan bir algoritmanın 16 işlemcili bir bilgisayarda en fazla 2 kat hızlandığı görülmüştür.

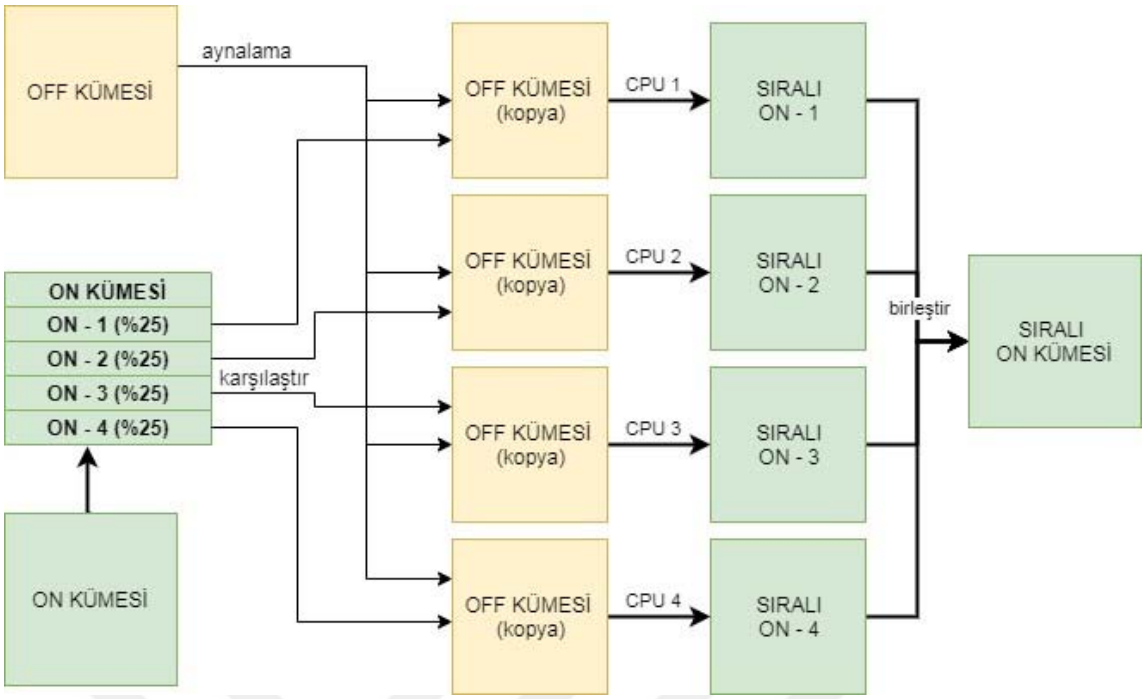


Şekil 4.8. Amdahl kanunu

Bu tez çalışmasında geliştirilen yakın ve kesin sonuç kapsama algoritmaları aynı mintermler üzerinde çalışmaktadır. Kapsama işleminin yapılması için bütün ON mintermlerine ve OFF mintermlerine ihtiyaç duyulmaktadır. Bu mintermler birbirleriyle karşılaştırılarak kontrol edilmektedir. Ayrıca bu mintermler kapsandıkça silinmekte yani küme dinamik olarak değişmektedir. Aksi takdirde en sade kapsama sonucuna ulaşılamaz. Bu sebeple geliştirilen algoritmalarda verilerin paralel programlama ile eş zamanlı işlenmesi mümkün değildir.

Yakın sonuç ve kesin sonuç kapsama algoritmalarında yapılan işler belli ölçüde paralel hale getirilebilir. Algoritmadaki çoğu işlem adımı birbirine bağlı olduğundan, bir öncekinin sonucu diğerini doğrudan etkilediğinden genel olarak kapsama algoritmaları paralel programlamaya uygun değildir. Fakat geliştirilen algoritmalarda özellikle mintermlerin ve implikantların birbirleriyle karşılaştırılmasında "foreach döngüsü" kullanılmaktadır. Bazıları rekürsif olduğu ya da aynı nesneleri kullandığı için paralel programlama mümkün olmamıştır. Diğer döngülerde paralel programlama kullanılmaya çalışılmıştır.

İzole mintermlerin tespitinde bütün ON mintermleri OFF mintermleri ile karşılaştırılarak izole seviyesini belirten puan hesaplanmaktadır. Bu hesaplama işleminde veri setinde değişiklik olmadığından verilerin işlenmesi paralel programlamaya uyarlanmıştır. İzole seviyesi puanını farklı işlemcilerde hesaplamak için ON minterm kümesi dörde bölünmüş ve OFF minterm kümesinin 4 kopyası oluşturulmuştur. Algoritmanın paralel programlamaya uyarlanması için yapılan işlemler Şekil 4.9'de verilmiştir. Bu algoritma paralel programlamaya uygundur.



Şekil 4.9. İzole mintermlerin paralel programlama ile tespiti

Yakın sonuç kapsama algoritmasında bir mintermi kapsayan implikantların kaç tane ON mintermini kapsadığı FOR döngüsü ile yapılmıştır. Burada implikantların kapsama oranı paralel algoritmalarla yapılmıştır. Yakın ve kesin sonuç kapsama algoritmalarında görev parallelleştirmesi yapılarak, AI kümesine yeni bulunan asal implikantı ekleme ve On mintermler kümesinden AI'nın kapsadığı mintermlerin çıkarılması işlemi farklı işlemcilerde yapılmıştır. Ayrıca bulunan implikantların kapsadığı ON mintermlerin tespiti paralel algoritmalar yardımıyla yapılmıştır. Bulunan sonuçlar birleştirilerek birbirleriyle karşılaştırılmıştır. *BEK* kümesine *Kayıt* eklenmesi bir işlemcide yapılırken diğer işlemci onu beklemeden sıradaki mintermi işlemeye devam etmektedir. Buna ek olarak, kesin sonuç kapsama algoritmasında her AI tanımlandıktan sonra yapılan *BEK* kümesindeki K'ların işlenmesi görevi paralel olarak yapılmaktadır.

Her ne kadar algoritmaların paralel programlamaya uyarlanması çabaları programın bazı bölümlerini hızlandırsa da paralel programlama için işlemci düzeyinde yapılan hazırlıklar ve farklı işlemcilerdeki işlem sonuçlarının birleştirilmesi zaman kaybettirmektedir. Bu sebeple küçük dosyalar için paralel algoritmalar verimli olmamaktadır.

4.6. Fonksiyon Sadeleştirme Uygulaması

Bu bölümde geliştirilen sadeleştirme algoritmalarının kodlandığı program tanıtılmıştır. Programın girdi olarak kabul ettiği lojik fonksiyon dosya formatı ve sadeleştirme programının kullanıcı arayüzü aşağıda açıklanmıştır.

4.6.1. Kullanılan dosya formatı

Dünya genelinde iki seviyeli lojik fonksiyon sadeleştirme programlarında girdi olarak PLA dosya formatı kabul edilmektedir. Bu dosyada *ON*, *OFF* ve *DC* mintemleriyle birlikte giriş değişkeni, çıkış değişkeni ve minterm sayıları da verilebilir.

Lojik fonksiyon dosyalarında bulunan anahtar kelimeler ve anlamları aşağıdaki gibidir:

i) *.i sayı* – giriş değişkeni sayısını ifade eder
ii) *.o sayı* – çıkış değişkeni sayısını ifade eder
iii) *.p sayı* – minterm sayısını ifade eder
iv) *.e sayı* – dosya sonunu gösterir.

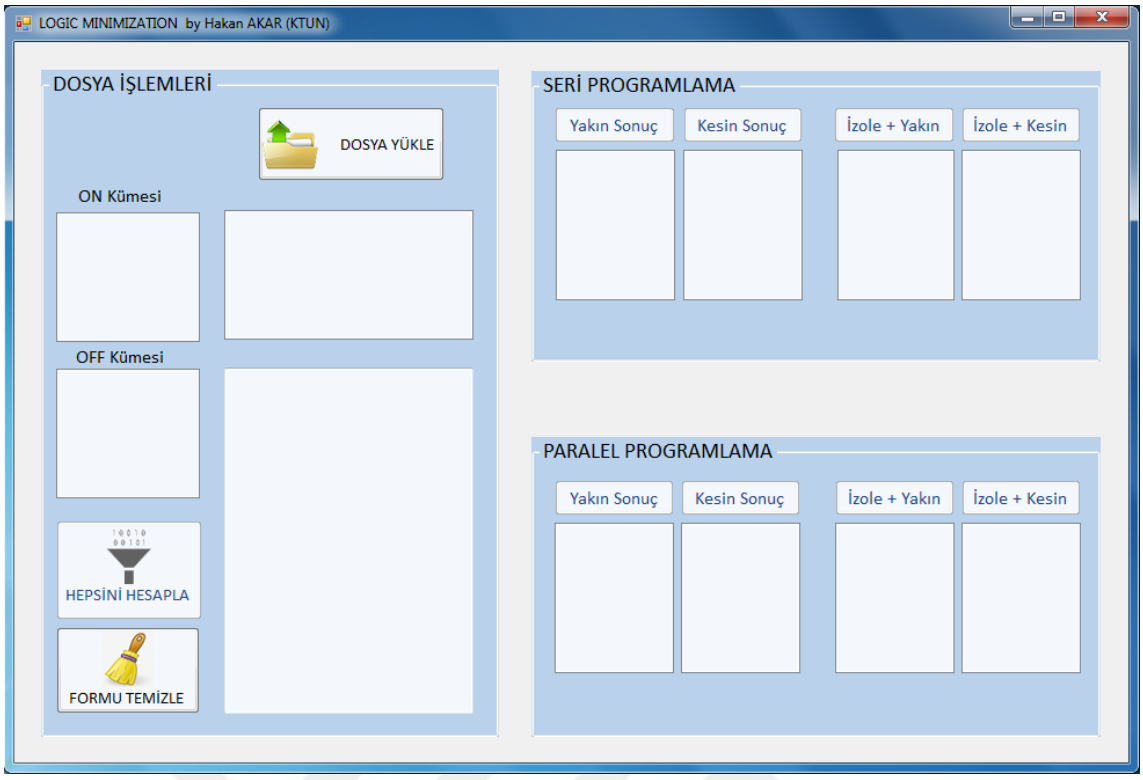
Yukarıdaki özet bilgilerden sonra dosyada mintermler listelenir. Her satırda bir minterm ve bir karakter boşluk'tan sonra mintermin ait olduğu küme verilir. "*1*" mintermin *ON* kümesinde olduğunu, "*0*" mintermin *OFF* kümesinde olduğunu ve "*" mintermin *DC* kümesinde olduğunu belirtir. Anahtar kelimeleri de içeren örnek bir lojik fonksiyon dosyası Şekil 4.10'de verilmiştir.

```
.i 10
.o 1
.p 12
1111001110 1
1001000010 0
0101000010 *
0011110110 1
1011110110 1
0111110110 0
1111110110 *
0000001110 1
1000001110 1
0000100010 0
1000100010 0
0100100010 1
.e
```

Şekil 4.10. Fonksiyon sadeleştirmede kullanılan girdi dosyası

4.6.2. Kullanıcı arayüzü

Bu tez çalışmasında lojik fonksiyon sadeleştirmede kullanılabilecek modüler algoritmalar geliştirilmiştir. Geliştirilen kullanıcı arayüzü bu algoritmaların tek başına ve birarada çalışabilmesine olanak sağlamaktadır. Şekil 4.11'da programın kullanıcı arayüzü verilmiştir. Programın kullanıcı arayüzü temelde 3 bölümden oluşmuştur. Sol tarafta "Dosya İşlemleri", sağ üst tarafta "Seri Programlama" ile çalışan sadeleştirme algoritmaları, sağ alt tarafta "Paralel Programlama" ile çalışan sadeleştirme algoritmaları yer almaktadır. Program ilk açıldığında "Dosya İşlemleri" bölümünde bulunan "Dosya Yükle" düğmesi aktif, diğer bütün düğmeler pasif konumdadır. "Dosya Yükle" düğmesine basıldığında açılan diyalog penceresi ile sadeleştirilmek istenen lojik fonksiyon dosyası seçilir ve programa yüklenir. ON mintermler kümesi, OFF mintermler kümesi tanımlanır ve dosya uzunluğu, küme boyutu, giriş değişkeni gibi sabitlerin değerleri atanır. Bütün bu hazırlıklar tamamlanınca diğer butonlar aktif hale gelir.



Şekil 4.11. Fonksiyon sadeleştirme programı kullanıcı arayüzü

Şekil 4.12'de verilen "Seri Programlama" bölümünde bulunan "Yakın Sonuç" butonu, geliştirilen yakın sonuç kapsama algoritmasını çalıştırarak lojik fonksiyonları çok hızlı sadeleştirmektedir. Sadeleştirme sonucunda bulunan asal implikantlar butonun hemen altındaki listbox'a yazdırılır. İşlem süresini tutan "StopWatch" yardımıyla hesaplanan işlem zamanı ve bulunan asal implikant sayısı listboxun hemen altında yazdırılır. Benzer şekilde aynı bölümünde bulunan "Kesin Sonuç" butonu, geliştirilen kesin sonuç kapsama algoritmasını çalıştırarak lojik fonksiyonları sadeleştirmektedir. Sadeleştirme sonucunda bulunan asal implikantlar butonun hemen altındaki listbox'a yazdırılır. Hesaplanan işlem zamanı ve bulunan asal implikant sayısı listboxun hemen altında yazdırılır.



Şekil 4.12. Seri programlama algoritmaları kullanıcı arayüzü

"Seri Programlama" bölümünde bulunan "İzole + Yakın" butonu iki algoritmayı sırayla çalıştırmaktadır. Öncelikle yüklenen dosyadaki ON mintermlerini OFF mintermleri ile karşılaştırarak izole mintermleri tespit edip yeni bir sıralama yapmaktadır. Ardından sıralı ON mintermlerini geliştirilen yakın sonuç kapsama algoritması yardımıyla çok hızlı sadeleştirmektedir. Sadeleştirme sonucunda bulunan asal implikantlar butonun hemen altındaki listbox'a yazdırılır. İşlem süresini tutan "StopWatch" yardımıyla hesaplanan işlem zamanı ve bulunan asal implikant sayısı listboxun hemen altında yazdırılır. Benzer şekilde aynı bölümde bulunan "İzole + Kesin" butonu öncelikle yüklenen dosyadaki ON mintermlerini OFF mintermleri ile karşılaştırarak izole mintermleri tespit edip yeni bir sıralama yapmaktadır. Ardından sıralı ON mintermlerini geliştirilen kesin sonuç kapsama algoritması yardımıyla sadeleştirmektedir. Sadeleştirme sonucunda bulunan asal implikantlar butonun hemen altındaki listbox'a yazdırılır. Hesaplanan işlem zamanı ve bulunan asal implikant sayısı listboxun hemen altında etiket olarak yazdırılır.

Şekil 4.13'de verilen "Paralel Programlama" bölümünde bulunan "Yakın Sonuç" butonu, geliştirilen yakın sonuç kapsama algoritmasının paralel programlama ile kodlanmış halini çalıştırarak lojik fonksiyonları çok hızlı sadeleştirmektedir. Sadeleştirme sonucunda bulunan asal implikantlar butonun hemen altındaki listbox'a yazdırılır. İşlem süresini tutan "StopWatch" yardımıyla hesaplanan işlem zamanı ve bulunan asal implikant sayısı ilgili listboxun hemen altında yazdırılır. Benzer şekilde aynı bölümde bulunan "Kesin Sonuç" butonu, geliştirilen kesin sonuç kapsama algoritmasının paralel programlama ile kodlanmış halini çalıştırarak lojik fonksiyonları sadeleştirmektedir. Sadeleştirme sonucunda bulunan asal implikantlar butonun hemen

altındaki listbox'a yazdırılır. Hesaplanan işlem zamanı ve bulunan asal implikant sayısı listboxun hemen altında yazdırılır.



Şekil 4.13. Paralel programlama algoritmaları kullanıcı arayüzü

"Paralel Programlama" bölümünde bulunan "İzole + Yakın" butonu paralel programlamaya uygun hazırlanmış iki algoritmayı sırayla çalıştırmaktadır. Öncelikle yüklenen dosyadaki ON mintermlerini 4 eşit gruba ayırıp farklı işlemcilerde OFF mintermleri ile karşılaştırarak izole mintermleri tespit edip yeni bir sıralama yapmaktadır. Ardından sıralı ON mintermlerini geliştirilen yakın sonuç kapsama algoritmasının paralel programlamaya uygun sürümünü kullanarak çok hızlı sadeleştirmektedir. Sadeleştirme sonucunda bulunan asal implikantlar butonun hemen altındaki listbox'a yazdırılır. İşlem süresini tutan "StopWatch" yardımıyla hesaplanan işlem zamanı ve bulunan asal implikant sayısı listboxun hemen altında yazdırılır. Benzer şekilde aynı bölümünde bulunan "İzole + Kesin" butonu öncelikle yüklenen dosyadaki ON mintermlerini OFF mintermleri ile paralel programlama yardımıyla karşılaştırarak izole mintermleri tespit edip yeni bir sıralama yapmaktadır. Ardından sıralı ON mintermlerini geliştirilen kesin sonuç kapsama algoritmasının paralel programlamaya uygun kodlanmış sürümü yardımıyla sadeleştirmektedir. Sadeleştirme sonucunda bulunan asal implikantlar butonun hemen altındaki listbox'a yazdırılır. Hesaplanan işlem zamanı ve bulunan asal implikant sayısı listboxun hemen altında etiket olarak yazdırılır.

Sol taraftaki "Dosya İşlemleri" bölümünde bulunan "Hepsini Hesapla" butonu yüklenen fonksiyon dosyasını sadeleştirmek için sağ tarafta bulunan 8 sadeleştirme butonunu sırasıyla çalıştırır. Gerçekleşen sadeleştirme işlemleri sağ taraftaki ilgili listbox ve etiketlerde yazdırılır. Ayrıca bütün algoritmalara ait bulunan sonuçlar "Dosya

İşlemleri" bölümündeki büyük textbox'a yazdırılır. Bulunan sonuçlar textbox'tan kopyalanıp başka programlarda kullanılabilir. "Dosya İşlemleri" bölümünde bulunan "Formu Temizle" düğmesi ile bellekteki program ve data bilgileri silinerek program tekrar başlatılır. Böylece ardarda farklı dosyaların sadeleştirilmesine imkân sağlanmıştır.

5. GELİŞTİRİLEN ALGORİTMALARIN ANALİZİ

Bir algoritmanın verilen bir problemi çözmek için ne kadar kaynağa ihtiyaç duyduğunu tahmin etmeye algoritma analizi denir. Bir problemi bir yılda çözen bir algoritma kullanışlı değildir. Bu sebeple algoritma analizinde kullanılan en önemli kaynak genellikle çalışma zamanıdır. Fakat bazı algoritmalarda bellek kullanımı ve iletişim ihtiyacı da incelenebilir. Bir problemin çözümünde kullanılacak uygun algoritmayı bulmak için en az birkaç algoritmayı incelemek gerekir. En uygun algoritmayı bulmak için algoritmaları birbiri ile karşılaştırırız ve onları bilgisayarda kodlarız. İnceleme sonuçları ve uygulama sonuçları tutarlı bir şekilde bir algoritmayı ön plana çıkarıyorsa, en iyi algoritmayı buluruz. Bir algoritmayı analiz etmek için donanım, veri ve uygulamalardan bağımsız olarak algoritmayı inceleyen matematiksel teknikler kullanmalıyız. Matematikçi Bachmann (1894) ve Landau (1953) tarafından tanımlanan gösterimlerin kullanıldığı bu tekniğin adı asimtotik algoritma analizidir.

5.1. Boolean Fonksiyonların Sağlanabilirliği ve Geçerliliği

Bir Boolean fonksiyonu $f(x)$'in çıkış değerini 1 yapan en az 1 adet giriş değeri/değerleri kümesi varsa $f(x)$ fonksiyonu için sağlanabilir / gerçeklenebilir (satisfiable) denir ve $f(x)$ fonksiyonu geçerlidir.

$$f(x) = f(x_1, \dots, x_n) \tag{5.1}$$

Eğer $\exists x_1, \dots, x_n\ [f(x_1, \dots, x_n)] = 1 \implies f(x)$ sağlanabilir ve geçerlidir

Örnek 5.1; $f(x) = (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)$ fonksiyonu en az 1 giriş değeri (x_1:1, x_2:1) için *doğru* sonuç verdiğinden sağlanabilir ve geçerlidir. Fakat $d(x) = (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_3 \vee \neg x_2)$ fonksiyonu bütün değerler için *yanlış* sonuç verdiğinden sağlanabilir değildir.

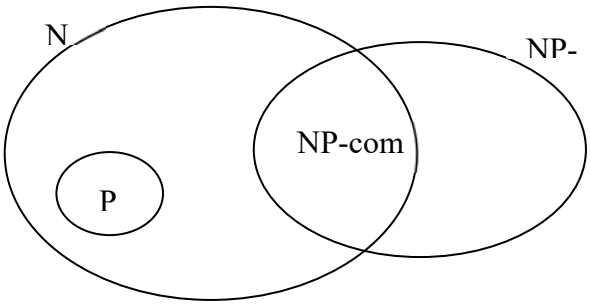
Eğer bir fonksiyon sağlanabilir değilse bu fonksiyonun tersi geçerlidir. Çünkü bütün çıkışları yanlış olan bir fonksiyonun tersini alırsanız en az 1 doğru sonuç veren bir fonksiyon oluşur. Dolayısıyla $\neg d(x) = \neg((\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_3 \vee \neg x_2))$ geçerlidir.

Bir fonksiyonun sağlanabilir olup olmadığını anlamak için bütün giriş değerlerinin kontrol edilmesi gerekmektedir. Sağlanabilirlik probleminin çözümünün karmaşıklığı $O(n^2 2^n)$'dir. Bu problemin zorluk derecesi Bölüm 5.2'de açıklanan NP-tam sınıfındadır. Bu sebeple sağlanabilirlik problemi, *doğru* sonuç oluşturabilecek giriş değerlerini tahmin eden ve deneyen çeşitli sezgisel algoritmalarla çözülmektedir.

5.2. Problemlerin Karmaşıklık Sınıfları

Hesaplama teorisi bir problemin belirli bir algoritma ve hesap modeli ile çözülüp çözülemeyeceğini veya çözülürse ne kadar hızlı ve verimli bir şekilde çözüleceğini inceler. Araştırmacılar problemlerin ne kadar sürede çözülebileceği, girdi değerleri artarsa problemin zorluğunun ne kadar yükseleceği üzerine araştırmalar yapmışlardır. Problemleri zorluk derecelerine göre 4 temel sınıfta inceleyebiliriz: polinomsal (polynomial, P), belirsiz polinomsal (nondeterministic polynomial, NP), belirsiz polinomsal zor (nondeterministic polynomial hard, NP-hard) ve belirsiz polinomsal tam (nondeterministic polynomial complete, NP-com).

P sınıfı, polinomsal zamanda çözülebilen problemlerden oluşur. Bu sınıftaki problemlerin zaman karmaşıklığı sabit bir k değeri ve n girdi değeri için $O(n^k)$'dır. Bir f: {0,1} → {0,1} fonksiyonuna polinomsal zamanda hesaplanabilir denilmesi için polinomsal zamanda çalışan ve verilen bir x girdi değerine $f(x)$ çıktısı üreten bir algoritma olması gerekir. P sınıfı NP sınıfının bir alt kümesi NP ⊇ P olarak kabul edilir.



Şekil 5.1. Problemlerin karmaşıklık sınıfları arasındaki ilişki

Problemlerin büyük bir bölümü P sınıfına girer. Fakat Turing'in meşhur "Halting Problem"i gibi bazı problemler hiçbir algoritma ile polinomsal zamanda çözülememiştir. Bu problemler belirsiz bir zamanda çözülebileceği için NP sınıfına girer. NP sınıfındaki

problemler polinomsal zamanda “doğrulanabilir”. Eğer bu problemler için girdi kümesi verilirse, doğru olup olmadıkları polinomsal zamanda kontrol edilebilir. Şekil 5.1 karmaşıklık sınıfları arasındaki ilişkiyi göstermektedir.

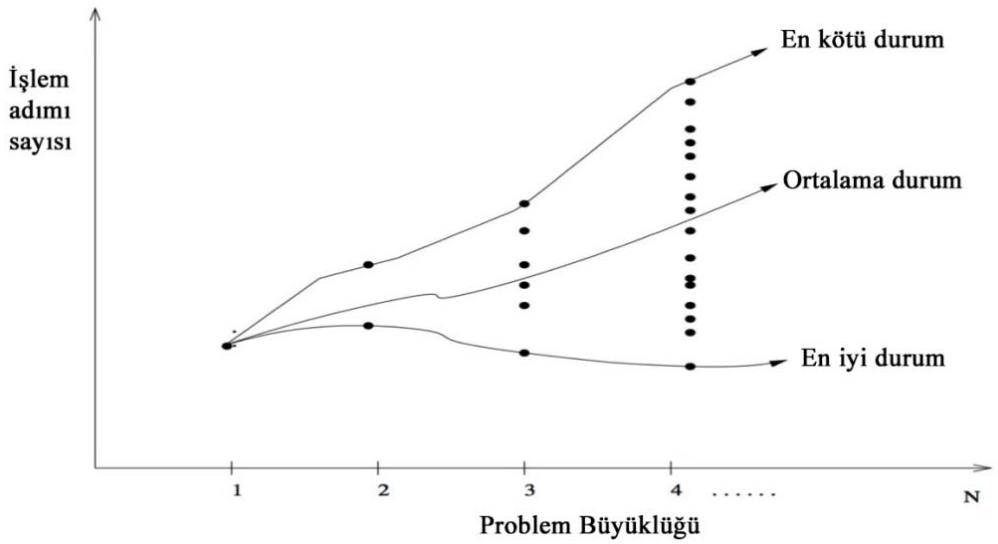
NP sınıfındaki herhangi bir problem kadar zor olan problemler NP-hard sınıfına girer. Bu problemler çözülürse NP sınıfındaki diğer problemler de çözülebilir. Bir problem NP sınıfında olmayabilir fakat NP-hard sınıfına girebilir. Dolayısıyla bazı problemler hem NP sınıfında hemde NP-hard sınıfında ise bu problemlere NP-com denir. Bu sınıftaki problemler hem NP sınıfındadır, hemde NP sınıfındaki diğer problemler kadar zordur.

5.3. Algoritmaların Karmaşıklık Analizi

Bu bölümde algoritmaların çalışma zamanı analizine ait temel bilgiler verilmiş ve asimptotik gösterimler tanıtılmıştır. Ardından doğrudan örtme algoritmalarının karmaşıklığı analiz edilerek yeni geliştirilen bitsel sadeleştirme yaklaşımının analizi ile karşılaştırılmıştır.

5.3.1. Algoritmaların çalışma zamanı analizi

Girdi boyutundaki artışa göre algoritmanın çalışma zamanındaki artışı yaklaşık olarak tahmin ederek ve hesaplayarak algoritmalar arasında göreceli olarak sınıflandırma işlemine çalışma zamanı analizi denir. Örneğin bir programın çalışması saniyeler, günler veya aylar sürebilir. Bu durum genellikle programı oluşturmak için kullanılan algoritmaya bağlıdır. Bir problemin çalışma zamanı, uygulama sırasında gerçekleşen temel işlemlerin sayısını da verir. Bir algoritmanın çalışma zamanından emin olmak için, algoritmanın en iyi, en kötü ve ortalama durumdaki performansını incelemek gerekir. Bu inceleme algoritmanın karmaşıklığını anlamamıza yardımcı olur. Algoritmaların en iyi, ortalama ve en kötü durum grafiği Şekil 5.2’de verilmiştir.



Şekil 5.2. Algoritmaların en iyi, ortalama ve en kötü durum grafiği

En kötü durum analizi, bir algoritmanın *n* boyutunda herhangi bir girişe sahip bir problemi çözmek için ihtiyaç duyduğu en uzun çalışma zamanını inceler. Bir algoritmanın en kötü çalışma zamanı bize hesaplama karmaşıklığının üst sınırını verir. Bu ayrıca algoritma performansının daha kötü olamayacağının da garantisidir.

En iyi durum analizi, bir algoritmanın *n* boyutunda herhangi bir girişe sahip bir problemi çözmek için ihtiyaç duyduğu en kısa çalışma zamanını inceler. Dolayısıyla bu çalışma zamanı hesaplama karmaşıklığının alt sınırını verir. Analizcilerin büyük bir bölümü, pek kullanışlı olmadığı için algoritmanın en iyi durum performansını değerlendirmezler.

Ortalama durum analizi, bir algoritmanın *n* boyutunda herhangi bir girişe sahip bir problemi çözmek için ihtiyaç duyduğu ortalama çalışma zamanını inceler. Ortalama durum çalışma zamanı genellikle en kötü durum çalışma zamanına yakın değerlendirilir. Fakat bir algoritmanın girdi verisinin bütün kümeleri üzerindeki ortalama performansını değerlendirmek için genellikle çok kullanışlıdır. Ortalama durum analizindeki engel analiz işlemini gerçekleştirmenin daha zor olması ve genellikle matematiksel düzeltmeler gerektirmesidir. Bu sebeple en kötü durum analizi yaygın olarak kullanılır.

5.3.2. Asimptotik gösterimler

Bir algoritmanın genişleme oranı, girdi değeri arttıkça algoritmanın ihtiyacının arttığı orandır. Bu bize farklı algoritmaların birbirlerine göre performansını karşılaştırma

imkânı sağlar. Algoritmaların çalışma zamanını temsil etmek için doğal sayılar ve gerçek sayılar kümesinden oluşan asimptotik gösterimler kullanırız (Cormen ve ark., 2009). En çok kullanılan üç temel asimptotik gösterim(notasyon); Büyük O (Büyük Omicron), Büyük Ω (Büyük Omega) ve Büyük Θ (Büyük Teta) gösterimleridir.

Büyük O notasyonu, bir algoritmanın ihtiyaç duyduğu maksimum kaynak miktarını temsil eden asimptotik üst sınırını göstermek için kullanılır. $\mathcal{X}$ ve $\mathcal{Y}$ fonksiyonları $\mathcal{X}, \mathcal{Y}$: $\mathbb{N}\rightarrow\mathbb{R}^+$ şartını sağlayan iki fonksiyon ise

$\mathcal{X}(n) \in O\,(y_{(n)})$ vardır ve, (5.2)
Eğer $\exists$ k $\in \mathbb{R}^+$ ve $(\forall n_0 : n \geq n_0)$ ise $\Longrightarrow\ \mathcal{X}(n) \leq$ k.$\mathcal{Y}(n)$

Verilen bir $x_{(n)}$ fonksiyonu için $O(x_{(n)})$ fonksiyonlar kümesi denklem 5.3'deki gibi hesaplanır.

$O(x_{(n)})$=$\{y_{(n)}$: $\forall\ n \geq n_0$ değerleri için $0 \leq y_{(n)} \leq$ k.$x_{(n)}$ şartını sağlayan pozitif k ve n_0 değerleri vardır$\}$ (5.3)

Büyük Ω notasyonu bir algoritmanın ihtiyaç duyduğu minimum kaynak miktarını temsil eden asimptotik alt sınırını göstermek için kullanılır. $\mathcal{X}$ ve $\mathcal{Y}$ fonksiyonları $\mathcal{X}, \mathcal{Y}$: $\mathbb{N}\rightarrow\mathbb{R}^+$ şartını sağlayan iki fonksiyon ise

$\mathcal{X}(n) \in \Omega\,(y_{(n)})$ vardır ve, (5.4)
Eğer $\exists$ k $\in \mathbb{R}^+$ ve $(\forall n_0 : n \geq n_0)$ ise $\Longrightarrow\ \mathcal{X}(n) \geq$ k.$\mathcal{Y}(n)$

Verilen bir $x_{(n)}$ fonksiyonu için $\Omega(x_{(n)})$ fonksiyon kümesi denklem 5.5'deki gibi hesaplanır.

$\Omega(x_{(n)})$=$\{y_{(n)}$: $\forall\ n \geq n_0$ değerleri için $0 \leq$ k.$x_{(n)} \leq y_{(n)}$ şartını sağlayan pozitif k ve n_0 değerleri vardır$\}$ (5.5)

Büyük Θ notasyonu bir algoritmanın hem asimptotik üst sınırını hemde asimptotik alt sınırını gösteren fonksiyonu tanımlamak için kullanılır.

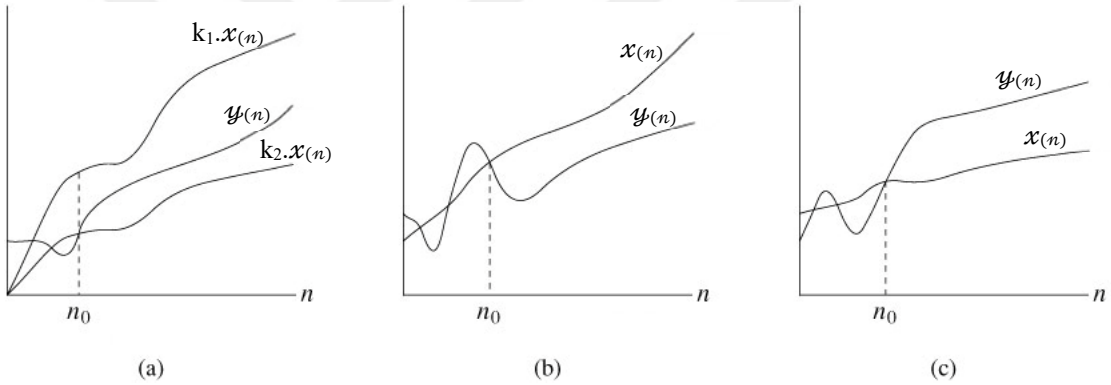
$\mathcal{X}$ ve $\mathcal{Y}$ fonksiyonları $\mathcal{X}, \mathcal{Y}$: $\mathbb{N} \rightarrow \mathbb{R}^+$ şartını sağlayan iki fonksiyon ise

$$\mathcal{X}(n) \in \Theta\,(y_{(n)}) \iff \mathcal{X}(n) \in \Omega\,(y_{(n)}) \text{ ve } \mathcal{X}(n) \in O\,(y_{(n)}) \tag{5.6}$$

Eğer $\exists\ k_1, k_2 \in \mathbb{R}^+$ ve $(\forall n_0 : n \geq n_0)$ ise $\Rightarrow k_1.\mathcal{Y}(n) \geq \mathcal{X}(n) \geq k_2.\mathcal{Y}(n)$

Verilen bir $x_{(n)}$ fonksiyonu için $\Theta(x_{(n)})$ fonksiyonlar kümesi denklem 5.7'deki gibi hesaplanır.

$\Theta(x_{(n)})$=$\{y_{(n)}$: $\forall\ n \geq n_0$ değerleri için $0 \leq k_1.x_{(n)} \leq y_{(n)} \leq k_2.x_{(n)}$ şartını sağlayan pozitif k_1, k_2 ve n_0 değerleri vardır$\}$ (5.7)



Şekil 5.3. Θ, *O* ve Ω notasyonlarının grafik gösterimleri

Şekil 5.3(a)'da Θ notasyonunun k_1 ve k_2 katsayıları ile $y_{(n)}$ fonksiyonunun alt ve üst değerlerini sınırlandırması gösterilmiştir. Şekil 5.3(b)'de Büyük *O* notasyonu $x_{(n)}$ ile $y_{(n)}$ fonksiyonunun üst değerlerini sınırlandırmaktadır. Şekil 5.3(c), Ω notasyonunun $x_{(n)}$ ile $y_{(n)}$ fonksiyonunun alt değerlerini sınırlandırmasını göstermektedir. Her üç grafik için de fonksiyonların doğruluğu $n \geq n_0$ değerinden sonra başlar.

Örneğin bir algoritmanın karmaşıklığı $O(n^2)$ ise algoritmanın girdi değeri büyüklüğü n arttıkça, zaman maliyetinin karesel olarak arttığını belirtir. Karmaşıklığı $O(log\ n)$ olan algoritmanın zaman maliyeti düşüktür (Çölkesen, 2004). Yani $O\ (log\ n)$ karmaşıklığındaki bir algoritmanın, $O(n^2)$ karmaşıklığındaki bir algoritmadan çok daha hızlı çalışması beklenir. Algoritmaların karmaşıklık hesabı, karmaşıklık fonksiyonunun en hızlı büyüyen değeri üzerinden yapılır. Algoritmaların farklı bölümleri farklı

karmaşıklık düzeyine sahip olabilir. Denklem 5.8'de 4 bölümden oluşan bir algoritmanın karmaşıklık fonksiyonu ve eşdeğer karmaşıklık derecesi gösterilmiştir.

$$\mathcal{X}(n) = n^3 + 20\,n^2 + 100n + 1000 \equiv O(n^3) \tag{5.8}$$

$\mathcal{X}(n)$ fonksiyonunun karmaşıklık derecesi $O(n^3)$'dür. Karmaşıklık hesabı algoritmaların yüksek girdi değerlerindeki durumunu inceler. Her ne kadar n sayısının küçük değerleri için ($n < n_0$) diğer bileşenler daha yüksek değerlere sahip olsa da, karmaşıklık hesabını en hızlı artış gösteren terim (n^3) belirler.

5.3.3. Doğrudan örtme algoritmalarının karmaşıklığı

Doğrudan örtme tekniğinde AI'ların bulunması ve mintermlerin kapsanması problemi Quine (1952) tarafından çalışılmıştır. Doğrudan örtme tekniği, lojik fonksiyonları iki aşamada sadeleştirir; mintermlere ait AI'ları üretir ve bütün ON mintermlerini kapsayan minimum AI kümesini bulur. Bu problem üzerinde yıllardır birçok araştırma yapılmasına rağmen, problemin yapısından dolayı çok etkili bir çözüm bulunamamıştır. Bu sebeple araştırmacılar her ne kadar problemi çözebilecek algoritmalar üzerinde çalışsa da teorik açıdan etkili bir çözümün olup olamayacağı da araştırma konusudur.

Doğrudan örtme algoritmasını 3 aşamada inceleyebiliriz.

1. On mintermlerini örten bütün AI'lar hesaplanır.
2. Mintermler ve AI'lar satır ve sütunlara yerleştirilip matriks oluşturulur.
3. Bütün On mintermlerini kapsayan ve maliyeti en az olan AI kümesi bulunur.

Bu algoritmanın ilk aşaması her minterm için AI'ları oluşturur. Bu aşama fonksiyonun On mintermleri kümesi büyüklüğünde polinomsal zamanda çözülebilir. İkinci aşamada bulunan AI'lar ile mintermler matriks'in satır ve sütunlarına yerleştirilerek kapsama tablosu oluşturulur. Böylece bir lojik fonksiyon, kapsama tablosuna dönüşmüş olur. Kapsama matriksi aynı şekilde polinomsal zamanda hazırlanabilir. Bu aşamalar P sınıfında problemlerdir. Fakat 3. aşamadaki minimum AI kümesinin bulunması meşhur NP-tam problem sınıfındadır. 3. aşama olan kapsama tablosundan minimum küme setinin bulunması NP sınıfı başka problemlerde de bulunur. Bu problemlerden birisinin polinomsal zamanda çözülmesi, diğerlerinin de polinomsal

zamanda çözülmesini yani P sınıfına girmesini sağlar. Bazı araştırmacılar minimum AI kümesinin bulunması üzerine çalışmalar yapmışlar fakat bu problem yine de NP-tam sınıfında kalmıştır (Umans ve ark., 2006).

Birçok araştırmacının geliştirdiği tekniklerin (Gavrilov, 1959; Svoboda, 1967; Necula, 1968) AI oluşturma algoritmaları $O(3^n)$ karmaşıklığına, birkaç teknikte (Morreale, 1970) ise algoritmalar $O(2^n)$ karmaşıklığa sahiptir. Sonuç olarak bütün doğrudan örtme algoritmalarının karmaşıklığı üssel (exponential)'dir.

5.3.4. YSKA ve KSKA algoritmalarının karmaşıklığı

Bu araştırmada lojik fonksiyonlar için bitsel operatörler yardımıyla yakın sonuç kapsama ve kesin sonuç kapsama algoritmaları geliştirilmiştir. Bölüm 5.3.3'de belirtildiği gibi kesin minimum AI kümesinin bulunması problemi NP-tam'dır. Araştırmanın bu bölümünde YSKA ve KSKA algoritmalarının karmaşıklığı incelenecektir.

Algoritmaların çalışma süreleri girdi değerine göre değişir. 100 mintermin kapsanması 10 mintermin kapsanmasından daha uzun sürer. Geliştirilen yakın sonuç kapsama algoritmasının kaba kodu Şekil 5.4'de gösterilmiştir.

```
1: PROGRAM YAKIN_SONUC (X, Y) //Sırasıyla ON ve OFF kümeleri
2: BEGIN
3:    WHILE (X) DO
4:       A = X1; // X Kümesinin ilk elemanını A mintermine ata
5:       P = GENISLET(A, Y); // A mintermini Y kümesiyle genislet
6:       R = AYIKLA (P); //P kümesinden kapsananları ayıkla
7:       AS = TUMLEYEN_AL(1-R); // Tamkümeden R kümesini çıkar
8:       FOR Q in AS DO // İmplikantlar(AS)'dan en iyisini bul
9:          S = SAYISI(X -Q); // ON kümesini sadeleştirme oranı
10:         IF (S<S2)    // eğer daha çok kapsarsa AI olur.
11:               S2 = S;
12:               AI = Q;
13:      ENDFOR;
14:      EAI = EAI + AI; // Bulunan AI sonuç kümesine ekleniyor
15:      X = X - AI; // X kümesinden kapsananlar çıkarılıyor
16:   ENDWHILE;
17:   RETURN EAI;
18: END
```

Şekil 5.4. Yakın sonuç kapsama algoritması kaba kodu

On minterm kümesi (X), Off minterm kümesi (Y) ve literal sayısını i ile gösterelim. Algoritmanın 3. satırından başlayan ve bütün ON minterm kümesini kapsayan

"While" döngüsü en fazla $\mathcal{X}$ kadar tekrar eder. Algoritmanın 5. satırı $\mathcal{X}$ mintermlerini $\mathcal{Y}$ mintermleri ile karşılaştırır. Bu işlemde $\mathcal{Y}$ karmaşıklığa sahiptir. Uygulama sonuçları ve benzer problemler 6. satırda bulunan kapsananların ayıklanması aşamasının $i/2$ karmaşıklığa sahip olduğunu göstermiştir. Tümleyen alınması tek seferlik işlem olduğundan tekrarı ve karmaşıklığı yoktur. Son olarak 8-13. satırlar arasındaki döngüde üretilen implikant kadar döngü vardır. Bir mintermi kapsayan implikant sayısı en fazla i kadar olabilir. Sunulan algoritmanın en kötü zaman karmaşıklığı Denklem 5.9'da verilmiştir.

$$O(f) = \mathcal{X}.\mathcal{Y}.(1+i/2)\,.\,i = \mathcal{X}.\mathcal{Y}.i.(i+2)/2 \tag{5.9}$$

Bir fonksiyonda olabilecek en fazla $\mathcal{X}$ ve $\mathcal{Y}$ mintermlerinin toplam sayısı 2^i olduğundan ve $\mathcal{X} + \mathcal{Y}$ sabit verilmiş ise $\mathcal{X}.\mathcal{Y}$'nin en fazla olması için her ikisininde eşit olması gerektiğinden

$\mathcal{X} + \mathcal{Y} = 2^i$ olmak şartıyla ve enfazla $\mathcal{X}$ ve $\mathcal{Y}$ değeri

$$(\mathcal{X}, \mathcal{Y}) \implies \mathcal{X} = \mathcal{Y} = (2^i)/2 = 2^{i-1} \tag{5.10}$$

$\mathcal{X}$ ve $\mathcal{Y}$ yerine 2^{i-1} koyarsak,

$$O(f) = \mathcal{X}.\mathcal{Y}.i.(i+2)/2 = 2^{i-1}.2^{i-1}.i.(i+2)/2 = i\,.\,(i+2).2^{2i-3}$$

$$O(f) = i\,.\,(i+2).2^{2i-3} \equiv O(2^{2i}) \tag{5.11}$$

Sunulan yöntemin karmaşıklığı $O(2^{2i})$ diğer yöntemlere benzer şekilde üsseldir. Fakat geliştirilen algoritma bitsel operatörler ile çalıştığı için bilgisayardaki işlem zamanı oldukça kısadır.

KSKA algoritması, YSKA algoritmasının geliştirilmiş versiyonudur. EAI'ların tespit edilmesi aşamasında kesin doğru bir sonuca ulaşmak mümkün olmadığında o mintermin işlenmesi ertelenir ve bütün mintermler işlendikten sonra tekrar ele alınır. Dolayısıyla KSKA algoritması YSKA algoritmasını kapsar. KSKA algoritmasındaki 13. satıra kadar verilen kodlar YSKA algoritması ile aynıdır. Geliştirilen kesin sonuç kapsama algoritmasının kaba kodu Şekil 5.5'de gösterilmiştir.

```
1: PROGRAM KESİN_SONUC (X, Y) //Sırasıyla ON ve OFF kümeleri
2: BEGIN
3:    WHILE (X) DO
4:       A = X1; // X Kümesinin ilk elemanını A mintermine ata
5:       P = GENISLET(A, Y); // A mintermini Y kümesiyle genislet
6:       R = AYIKLA (P); //P kümesinden kapsananları ayıkla
7:       AS = TUMLEYEN_AL(1-R); // Tamkümeden R kümesini çıkar
8:       FOR Q in AS DO // İmplikantlar(AS)'dan en iyisini bul
9:          S = SAYISI(X -Q); // ON kümesini sadeleştirme oranı
10:         IF (S<S2)   // eğer daha çok kapsarsa AI olur.
11:               S2 = S;
12:               AI = Q;
13:      ENDFOR;
14:      If (EAI)       // EAI bulunursa,
15:         EAI = EAI + AI; // Bulunan AI sonuç kümesine ekleniyor
16:         SW kontrol // SW'ler kontrol ediliyor
17:         X = X - AI; // X kümesinden kapsananlar çıkarılıyor
18:      Else
19:         Yeni_SW=X(eleman)
20:         SW ekle (Yeni_SW)
21:   ENDWHILE;
22://------Kalan SW'leri Patrick Fonksiyonuna göre işle----------
23:   SIRALA(X);  // Kalan SW'lerden sıralı X kümesi oluşturuluyor
24:   WHILE (X) DO
25:      Foreach (SW.AI) //ON mintermine ait En iyi AI bul
26:         X = KALANLAR(SW.AI)     // En iyi AI bul
27:      EAI = EAI + AI; // Bulunan en iyi AI sonuç kümesine ekle
28:      KONTROL (SW) // SW'leri kontrol et
29:      X = X - AI; // X kümesinden kapsananlar çıkarılıyor
30:   ENDWHILE;
31: RETURN EAI;
32: END
```

Şekil 5.5. Kesin sonuç kapsama algoritması kaba kodu

On minterm kümesi (X), Off minterm kümesi (Y), literal sayısını (i) ve bekleme durumundaki ON mintermi ve onu kapsayan implikantları (SW) ile gösterelim. Algoritmanın 3. satırından başlayan ve bütün ON minterm kümesini kapsayan "While" döngüsü en fazla X kadar tekrar eder. Algoritmanın 5. satırı X mintermlerini Y mintermleri ile karşılaştırır. Bu işlemde Y karmaşıklığa sahiptir. Uygulama sonuçları ve benzer problemler 6. satırda bulunan kapsananların ayıklanması aşamasının i/2 karmaşıklığa sahip olduğunu göstermiştir. Tümleyen alınması tek seferlik işlem olduğundan tekrarı ve karmaşıklığı yoktur. 8-13. satırlar arasındaki döngüde üretilen implikant kadar döngü vardır. Bir mintermi kapsayan implikant sayısı en fazla i kadar olabilir. 16. satırdaki *SW_kontrol* fonksiyonunun karmaşıklığı en fazla X kadar olabilir. 22. Satırdan sonra çalışan kodlar "*While*" döngüsü içinde değildir ve ayrı değerlendirilmelidir. 23. satırdaki sıralama fonksiyonunun karmaşıklığı en kötü durumda

$\mathcal{X}^2$ kadardır. 24 ile 30. satır arasındaki kodlar en fazla $\mathcal{X}$ kadar tekrar edebilir. 25. satırdaki SW döngüsü en fazla $\mathcal{X}$ kadar, 26. satırdaki kalanların hesaplanması en fazla $\mathcal{X}$ kadar tekrar edebilir. 28. satırdaki *SW_kontrol* fonksiyonunun karmaşıklığı en fazla $\mathcal{X}$ kadar olabilir. Sunulan algoritmanın en kötü zaman karmaşıklığı Denklem 5.12'da verilmiştir.

$$O(f) = \mathcal{X}.(\mathcal{Y}.(1+i/2).i + \mathcal{X}) + \mathcal{X}^2 + \mathcal{X}(\mathcal{X}.\mathcal{X} + \mathcal{X}) \quad (5.12)$$
$$O(f) = \mathcal{X}.\mathcal{Y}.i.(i+2)/2 + \mathcal{X}^2 + \mathcal{X}^2 + \mathcal{X}^3 + \mathcal{X}^2$$
$$O(f) = \mathcal{X}.\mathcal{Y}.i.(i+2)/2 + 3\,\mathcal{X}^2 + \mathcal{X}^3$$

Bir fonksiyonda olabilecek en fazla $\mathcal{X}$ ve $\mathcal{Y}$ mintermlerinin toplam sayısı 2^i olduğundan ve $\mathcal{X} + \mathcal{Y}$ sabit verilmiş ise $\mathcal{X}.\mathcal{Y}$'nin en fazla olması için her ikisininde eşit olması gerektiğinden

$\mathcal{X} + \mathcal{Y} = 2^i$ olmak şartıyla ve enfazla

$$(\mathcal{X}, \mathcal{Y}) \implies \mathcal{X} = \mathcal{Y} = (2^i)/2 = 2^{i-1} \quad (5.13)$$

olabilir. Denklem 5.12'deki $\mathcal{X}$ ve $\mathcal{Y}$ yerine 2^{i-1} koyarsak,

$$O(f) = \mathcal{X}.\mathcal{Y}.i.(i+2)/2 + 3\,\mathcal{X}^2 + \mathcal{X}^3 \quad (5.14)$$
$$O(f) = 2^{i-1}.2^{i-1}.i.(i+2)/2 + 3(2^{i-1})^2 + (2^{i-1})^3$$
$$O(f) = (2^{2i-2}/2).(i^2+2i+6) + 2^{3i-3}$$
$$O(f) = (2^{2i-3}).(i^2+2i+6) + 2^{3i-3} \equiv O(2^{3i})$$

```
1: PROGRAM İZOLE_TESPİT (𝒳, 𝒴) //Sırasıyla ON ve OFF kümeleri
2: BEGIN
3:    FOREACH (𝒳) // Bütün ON mintermlerini hesapla
4:       FOR 𝒴 DO // OFF mintermleri ile karşılaştır
5:          K = KARSİLASTİR(𝒳-𝒴); // ON & OFF Bitsel karşılaştır
6:          Iz = HESAPLA(K);  // İzole katsayısı hesapla
7:          𝒳.izole = Iz ; //Hesaplanan değeri minterm ile eşleştir
8:       ENDFOR;
9:    END_FOREACH;
10:   SIRALA (𝒳.izole) // 𝒳 kümesi hesaplanan katsayıya göre sırala
11:   RETURN 𝒳;
12: END
```

Şekil 5.6. İzole mintermlerin tespiti algoritması kaba kodu

Sunulan yöntemin karmaşıklığı $O(2^{3i})$ diğer yöntemlere benzer şekilde üsseldir. Fakat geliştirilen algoritma bitsel operatörler ile çalıştığı için bilgisayardaki işlem zamanı oldukça kısadır. İzole mintermlerin tespiti için ON mintermler kümesi, OFF mintermler kümesi ile karşılaştırılmıştır. İzole mintermlerin tespiti algoritmasının kaba kodu Şekil 5.6'da gösterilmiştir.

On minterm kümesi ($\mathcal{X}$) ve Off minterm kümesi ($\mathcal{Y}$) ile gösterelim. Algoritmanın 3. satırından başlayan ve bütün ON minterm kümesini kapsayan "Foreach" döngüsü $\mathcal{X}$ kadar tekrar eder. Algoritmanın 4. satırı $\mathcal{X}$ mintermlerini $\mathcal{Y}$ mintermleri ile karşılaştırır. Bu işlemde $\mathcal{Y}$ karmaşıklığa sahiptir. "*KARŞILAŞTIR*" ve "*HESAPLA*" işlemleri 7 adımda tamamlanmaktadır. 9. satıra kadar olan toplam karmaşıklık $\mathcal{X}.\mathcal{Y}$ kadardır. 10. satırdaki sıralama fonksiyonunun en kötü durum karmaşıklığı $\mathcal{X}^2$ kadardır. Sunulan algoritmanın en kötü zaman karmaşıklığı Denklem 5.15'da verilmiştir.

$$O(f) = \mathcal{X}.\mathcal{Y} + \mathcal{X}^2 \tag{5.15}$$

Denklem 5.10'a göre $\mathcal{X}$ ve $\mathcal{Y}$ yerine 2^{i-1} koyarsak,

$$O(f) = \mathcal{X}.\mathcal{Y} + \mathcal{X}^2 = 2^{i-1} \,.\, 2^{i-1} + 2^{i-1} \,.\, 2^{i-1}$$

$$O(f) = 2 \,.\, 2^{2i-2} = 2^{2i-1} \equiv O(2^{2i}) \tag{5.16}$$

Sunulan yöntemin karmaşıklığı $O(2^{2i})$ üsseldir. Fakat YSKA ve KSKA algoritmaları ile kıyaslandığında işlem adımı çok azdır. Geliştirilen algoritmaların karmaşıklık analizleri Çizelge 5.1'de verilmiştir.

Çizelge 5.1. Geliştirilen algoritmaların karmaşıklık analizleri

No	Algoritma Adı	Zaman Karmaşıklığı	*O(f)*
1	İzole Tespit Algoritması	2^{2i-1}	$O(2^{2i})$
2	YSKA Algoritması	$(2^{2i-3}).(i^2 + 2i)$	$O(2^{2i})$
3	KSKA Algoritması	$(2^{2i-3}) \,.\, (i^2+2i + 6)+ 2^{3i-3}$	$O(2^{3i})$
4	İzole + YSKA Algoritması	$(2^{2i-3}).(i^2 + 2i+4)$	$O(2^{2i})$
5	İzole + KSKA Algoritması	$(2^{2i-3}) \,.\, (i^2+2i + 10)+ 2^{3i-3}$	$O(2^{3i})$

Çizelge 5.1 de sunulan algoritmaların karmaşıklıkları incelendiğinde hepsinin karmaşıklığının üssel olduğu görülmektedir. Sunulan algoritmalardan işlem adımı en az

olan “İzole Tespit algoritması”, en fazla olanı ise “İzole + KSKA algoritması”dır. Ayrıca KSKA algoritmalarında en kötü durum karmaşıklığı $O(2^{3i})$ iken, YSKA algoritmalarında bu karmaşıklık $O(2^{2i})$‘dir.

5.3.5. Espresso algoritmasının karmaşıklığı ve KSKA ile karşılaştırılması

Bu araştırmada geliştirilen YKSA ve KSKA algoritmalarının etkililiğini belirlemek için dünyaca ünlü Espresso algoritmasının karmaşıklılığı incelenmiş ve geliştirilen algoritmaların karmaşıklılığı ile karşılaştırılmıştır. Espresso algoritmasının karmaşıklılığı incelenirken, YSKA ve KSKA algoritmalarında olduğu gibi karmaşıklığı maximum yapacak ON ve OFF kümelerinin eşit olma durumuna göre analiz edilmiştir.

Espresso algoritmasında sadeleştirme sonucunun en sade sonuç olduğu garanti edilemese de çok yakın sonuçlar vermektedir. ESPRESSO, hafıza ve zaman açısından kendinden önce geliştirilen algoritmalardan çok daha verimlidir. Bu algoritma ile onlarca değişken ve onlarca çıkış fonksiyonu kabul edilebilir zaman ve hafıza kullanımıyla hesaplanabilmektedir.

Bu algoritmanın sürekli tekrar eden 4 aşaması vardır.

i) Genişlet (Expand),
ii) Asal implikant (AI) bul (Essential prime),
iii) Kapsa (Irredundant cover),
iv) Sadeleştir (Reduce).

Bu algoritma öncelikle asal terime ulaşana kadar mintermleri genişletir, ardından asal terimleri *DC* yapar, artık gerekmeyen küpleri siler ve ifadeleri mümkün olduğu kadar az terimle ifade edecek kadar daraltır. Espresso algoritması kaba kodu Şekil 5.7’de verilmiştir.

```
1: PROGRAM ESPRESSO (𝒳, 𝒟) //Sırasıyla ON ve DC kümeleri
2: BEGIN
3: 𝒳_OLD = 𝒳;  // Sadeleştirme sonunda doğruluk kontrolü için sakla
4: 𝒴 = COMPLEMENT (𝒳,D);  // OFF minterm kümesi bulunuyor
5: 𝒳 = EXPAND (𝒳, 𝒴);  // İlk genişletme
6: 𝒳 = IRREDUNDANT (𝒳,D); // İlk kapsama
7: E = ESSENTIAL (𝒳,D);  // Gerekli asal implikantları bul
8: 𝒳 = 𝒳 - E ;  // ON minterm kümesinden gerekli AI'ları çıkar
9: D = D + E ;  // Gerekli AI'ları DC kümesine ekle
10:
11: DO {
12:   ϕ2 = cost(𝒳);
13:   DO {        //kararlı çözüm kümesi oluşasıya kadar tekrarla
14:         ϕ1 = |𝒳|;
15:         𝒳 = REDUCE (𝒳, D);
16:         𝒳 = EXPAND (𝒳, 𝒴);
17:         𝒳 = IRREDUNDANT (𝒳, D);
18:   } WHILE ( |𝒳|< ϕ1 );
19:         //Çözümü karıştır ve iyileştirme olurmu kontrol et
20:   G = LAST_GASP (𝒳, D, 𝒴);
21: } WHILE (cost(𝒳)< ϕ2);
22:
23: 𝒳 = 𝒳 + E ;  // Gerekli AI'lar ON minterm kümesine dönüyor
24: D = D - E ;
25: 𝒳 = MAKE SPARSE(𝒳, D, 𝒴);  // Çözüm kümesi sadeleştiriliyor
26:         //Çözüm kümesinin 𝒳 kapsadığı kontrol ediliyor
27: If (!verify(𝒳, D, 𝒳_OLD));
28:   Exit("Doğrulama Hatası");
29: RETURN 𝒳;
30: END
```

Şekil 5.7. Espresso algoritması kaba kodu

Üçüncü satırda sadeleştirme sonucunda doğruluğunun kontrol edilmesi için ON mintermler kümesinin yedeği alınıyor. Bu işlem $\mathcal{X}$ kadar tekrar eder. Dördüncü satırda ON ve DC kümeleri tam kümeden çıkarılarak OFF minterm kümesi bulunuyor. Bu işlem için olabilecek bütün ihtimallerden (2^i) ON ve OFF minterm kümeleri çıkarıldığı için bu işlemin karmaşıklığı 2^i-1 kadar olabilir. Beşinci satırdaki genişletme(EXPAND) prosedürü için herbir $\mathcal{X}$ mintermi, implikanta dönüştürülür ve $\mathcal{Y}$ kümesi ile karşılaştırılır. Ardından $\mathcal{X}$ daha da genişletilerek $\mathcal{Y}$ ile karşılaştırılır ve $\mathcal{X}$ mintermleri en büyük halini bulana kadar bu işlem devam eder. En kötü ihtimalde, herbir $\mathcal{X}$ mintermi giriş sayısı i kadar genişlese ve $\mathcal{Y}$ ile karşılaştırılsa bu işlemin karmaşıklığı $i.\mathcal{X}.\mathcal{Y}$ kadar olur. 6. satırdaki kapsama (IRREDUNDANT) fonksiyonu genişletme sonucu bulunan implikantları, aynı mintermleri kapsayacak şekilde daraltıyor. Genişletilen implikant sayısı $\mathcal{X}$'i geçemeyeceği için bu fonksiyonun en fazla $\mathcal{X} + \mathcal{D}$ kadar sürer ve en kötü

ihtimalde 2^i-1 karmaşıklığa sahiptir. 7 satırdaki zorunlu AI'ların bulunması fonksiyonu (ESSENTIAL) en fazla X ve D sayısı kadar tekrar eder ve en kötü ihtimalde 2^i-1 karmaşıklığa sahiptir. Algoritmanın ilk bölümünün karmaşıklığı Denklem 5.17'de verilmiştir.

$$O(f) = X + 2^i + X \,.\, Y \,.\, i + 2^i - 1 + 2^i - 1 + 1 + 1 \quad (5.17)$$
$$O(f) = X + 2^i + X \,.\, Y \,.\, i + 2^i + 2^i$$
$$O(f) = X + X \,.\, Y \,.\, i + 3 \,.\, 2^i$$

11. satırda başlayıp, 21. satırda biten Do-While döngüsü X kadar tekrar edebilir. 13. Satırda başlayan ve 18. Satırda biten Do-While döngüsü her minterm için implikant sayısı kadar tekrar edebilir. En kötü ihtimalde her minterm için i kadar implikant bulunduğu varsayılırsa bu iki döngünün içindeki kodlar $X.\, i$ kadar tekrar edebilir. 15. satırdaki sadeleştir (REDUCE) fonksiyonu bulunan AI'ları daha küçük kapsama alanı olan implikantlarla değiştirerek genişletme aşamasına hazırlık yapar. En kötü durumda X mintermlerini D kadar karşılaştırır ve $X.\, D$ karmaşıklığa sahiptir. Genişletme ve kapsama fonksiyonları yukarıda açıklandığı gibi hesaplandığında bu iki döngünün toplam karmaşıklığı Denklem 5.18'de verilmiştir.

$$O(f) = X.(1+i\,(\,1+ X.\, D + i.\, X.\, Y + 2^i\text{-}1) +1) \quad (5.18)$$
$$O(f) = X.(1+i + X.\, D.\, i + i \,.\text{i}.\, X.\, Y + i \,.2^i - i\,) +1)$$
$$O(f) = X + i.X + X.X.D.i + i.i.X.X.Y + X.i.2^i - X.\, i + 1$$
$$O(f) = X + X^2.D.i + i^2.X^2.Y + X.i.2^i + 1$$

23-28. satırlarda belirtilen kodların karmaşıklığı 1'dir. Bu satırlarla birlikte Denklem 5.17 ve Denklem 5.18'de verilen formüller toplandığında Denklem 5.19'u elde ederiz.

$$O(f) = (X + X \,.\, Y \,.\, i + 3 \,.\, 2^i) + (\,X + X^2.D.i + i^2.X^2.Y + X.i.2^i + 1) + 5 \quad (5.19)$$
$$O(f) = X + X \,.\, Y \,.\, i + 3 \,.\, 2^i + X + X^2.D.i + i^2.X^2.Y + X.i.2^i + 6$$
$$O(f) = 2X + X.Y.i + 3.2^i + X^2.D.i + i^2.X^2.Y + X.i.2^i + 6$$

Bir fonksiyonda olabilecek en fazla X, Y ve D mintermlerinin toplam sayısı 2^i olduğundan ve $X.Y$'nin veya $X.D$'nin en fazla olması için her ikisininde eşit olması gerektiğinden Denklem 5.13'e göre X, Y ve D yerine $2^{i\text{-}1}$ koyarsak,

$$O(f) = 2X + X.Y.i + 3.2^i + X^2.D.i + i^2.X^2.Y + X.i.2^i + 6 \quad (5.20)$$
$$O(f) = 2(2^{i\text{-}1}) + (2^{i\text{-}1}).\,(2^{i\text{-}1}).i + 3.2^i + (2^{i\text{-}1})^2.\,(2^{i\text{-}1}).i + i^2.\,(2^{i\text{-}1})^2.\,(2^{i\text{-}1}) + (2^{i\text{-}1}).i.2^i + 6$$

$$O(f) = 2(2^{i-1}) + (2^{i-1})^2.i + 3.2^i + (2^{i-1})^3.i + i^2.\ (2^{i-1})^3 + (2^{2i-1}).i + 6$$
$$O(f) = 2(2^{i-1}) + 2^{2i-2}.i + 3.2^i + 2^{3i-3}.i + i^2.\ 2^{3i-3} + (2^{2i-1}).i + 6 \equiv O(i^2.2^{3i})$$

Espresso algoritmasının karmaşıklığı $O(i^2.2^{3i})$ sunulan YSKA algoritmasının karmaşıklığından $O(2^{2i})$ ve KSKA algoritmasının karmaşıklığından $O(2^{3i})$ daha yüksektir. Çizelge 5.1'de verilen karmaşıklık analizleri incelendiğinde geliştirilen bütün algoritmaların karmaşıklığı Espresso algoritmasının karmaşıklığından daha azdır. Ayrıca YSKA ve KSKA algoritmaları bitsel operatörler ile çalıştığı için programlama dilinde kodlandığında daha hızlı çalışması beklenmektedir.

Başçiftçi F. ve Akar H. (2020), geliştirilen KSKA algoritması ile Espresso algoritmasını kullanarak 20 MCNC benchmarkı üzerinde sadeleştirme yapmıştır. Araştırma sonuçları bu bölümde sunulan karmaşıklık analizine uygundur. 20 MCNC benchmarkı üzerinde yapılan sadeleştirme işleminde geliştirilen KSKA algoritması daha kısa sürede, daha az bellek kullanarak daha sade sonuçlara ulaşmıştır.

6. UYGULAMA SONUÇLARI VE TARTIŞMA

Bu bölümde bitsel operatörler ile geliştirilen yakın sonuç kapsama algoritması ve kesin sonuç kapsama algoritmasının analizi yapılmıştır. Algoritmaların hesaplama zamanı ve sonuç kaliteleri karşılaştırılmıştır. İzole mintermlerin tespiti yapılarak sadeleştirmeye izole mintermlerden başlamanın sadeleştirme üzerindeki etkisi incelenmiştir. Ayrıca geliştirilen 3 algoritma paralel olarak programlanarak, seri versiyonlarıyla karşılaştırılmıştır. Böylece her 3 algoritmanın da paralel programlamaya ne kadar uygun olduğu değerlendirilmiştir. Tartışma bölümünde algoritmaların analizi ile bulunan uygulama sonuçları değerlendirilmiştir.

6.1. Lojik Fonksiyonların Sadeleştirilmesi

Lojik fonksiyonların sadeleştirilmesi için geliştirilen YSKA ve KSKA algoritmaları, Microsoft Visual Studio programında C# dili kullanılarak kodlanmıştır. Geliştirilen program dünyaca kabul görmüş 50 temel MCNC Benchmarkları üzerinde denenmiştir. Kullanılan benchmarklar tek çıkışlı olup tam tanımlanmış ve tam tanımlanmamış fonksiyonları içermektedir. Algoritmaların verimliliğinin karşılaştırılması için dosyaları sadeleştirme süreleri ve sadeleştirme sonucunda ulaştıkları SOP sayısı esas alınmıştır.

6.1.1. YSKA ve KSKA'nın karşılaştırılması

Performans karşılaştırmasında kullanılan 50 MCNC benchmarkında toplam 3261 ON mintermi, 8962 OFF mintermi bulunmaktadır. Yapılan performans testinde YSKA algoritması bu dosyaları toplamda 559 AI'ye, 322,07 milisaniyede sadeleştirmiştir. YSKA algoritmasının ON mintermlerini sadeleştirme oranı %82,86 olmuştur. YSKA algoritması 516 ON mintermine sahip max1024 benchmarkını 4 AI'ye sadeleştirerek bu dosyada %99,22 başarı sağlamıştır. Algoritmanın en kısa sürede sadeleştirdiği benchmark 0,33 milisaniye ile "check2" olurken, en uzun sürede sadeleştirdiği benchmark 50,91 milisaniye ile "t12" olmuştur.

KSKA algoritması bu dosyaları toplamda 548 AI'ye 2230,15 milisaniyede sadeleştirmiştir. KSKA algoritmasının ON mintermlerini sadeleştirme oranı %83,20 olmuştur. KSKA algoritması 516 ON mintermine sahip max1024 benchmarkını 4 AI'ye

sadeleştirerek bu dosyada %99,22 başarı sağlamıştır. Algoritmanın en kısa sürede sadeleştirdiği benchmark 0,26 milisaniye ile “check2” olurken, en uzun sürede sadeleştirdiği benchmark 1683,24 milisaniye ile “prom1” olmuştur.

KSKA algoritması YSKA algoritmasına göre 6,92 kat daha yavaş çalışmış ve %0,34 daha fazla sadeleştirme yapmıştır. KSKA algoritmasının 50 dosya üzerinde ortalama çalışma zamanı 44,6 milisaniye olurken, YSKA algoritmasında bu sayı 6,44 milisaniyedir. YSKA ve KSKA algoritmalarının performans testi sonuçları Çizelge 6.1’de verilmiştir.

Çizelge 6.1. Yakın Sonuç ve Kesin Sonuç Kapsama Algoritmaları

Benchmarklar	Değişken Sayısı	$\mathcal{X}_{ON}$ Sayısı	$\mathcal{Y}_{OFF}$ Sayısı	YSKA		KSKA	
				AI Sayısı	Çalışma Zamanı (ms)	AI Sayısı	Çalışma Zamanı (ms)
wim.txt	4	9	1	4	3,27	4	2,26
check3.txt	4	8	5	2	0,56	2	0,45
check2.txt	4	4	6	1	0,33	1	0,26
o4.txt	4	8	8	3	2,31	3	2,45
check.txt	4	4	9	1	3,17	1	3,43
p82.txt	5	11	13	4	3,55	4	5,16
o5.txt	5	13	16	5	2,96	5	3,57
o51.txt	5	13	16	6	3,89	5	5,61
squar.txt	5	9	23	2	3,58	2	2,29
new2.txt	6	3	4	2	0,41	2	0,38
m.txt	6	27	5	4	3,57	4	2,60
m5.txt	6	27	5	4	3,57	4	2,90
poperom.txt	6	56	8	7	3,99	7	4,80
sqr.txt	6	18	46	2	3,38	2	2,33
inc.txt	7	12	22	6	3,24	6	3,19
sqn.txt	7	48	48	8	6,20	8	7,94
linrom.txt	7	65	63	24	11,61	26	13,26
max128.txt	7	29	99	8	6,08	8	7,35
z5xp1.txt	7	25	103	3	3,52	3	2,54
max3.txt	7	12	116	7	4,12	7	4,91
m4.txt	8	220	25	27	8,35	25	143,99
m3.txt	8	98	30	16	8,83	16	16,94
exp1.txt	8	24	42	3	3,45	3	3,83
exp.txt	8	18	52	3	3,35	3	2,63
f51m.txt	8	128	128	23	10,93	23	14,21
e.txt	8	65	128	21	10,86	21	14,71
exps.txt	8	65	131	21	6,47	21	7,84
rd84.txt	8	120	136	85	37,83	84	36,73
dist.txt	8	53	203	12	8,24	12	7,03
ex5.txt	8	33	223	2	3,46	2	2,81
mlp4.txt	8	32	224	9	6,84	9	5,76
root.txt	8	15	241	4	3,83	4	2,82
max4.txt	9	37	8	7	2,34	6	26,35
prom1.txt	9	413	30	23	7,28	22	1683,24
min.txt	9	87	46	6	3,30	6	3,26
prom2.txt	9	142	145	7	4,33	7	4,36
max512.txt	9	267	245	10	8,20	10	6,91
apex4.txt	9	4	434	4	2,38	4	2,19
max1024.txt	10	516	508	4	4,73	4	3,85
t12.txt	10	266	758	108	50,91	104	76,45
br11.txt	12	28	5	3	2,41	3	7,81
br2.txt	12	29	6	2	2,44	2	2,19
br1.txt	12	25	8	5	1,46	5	2,22
t3.txt	12	27	121	6	4,18	6	3,74
pdc.txt	16	29	1886	4	7,60	4	3,53
spla.txt	16	67	2036	31	26,42	29	27,92
den.txt	18	24	3	5	4,39	4	33,48
bca.txt	26	15	13	1	1,23	1	1,54
bcc.txt	26	3	242	2	1,29	2	1,83
bcb.txt	26	10	289	2	1,44	2	2,33

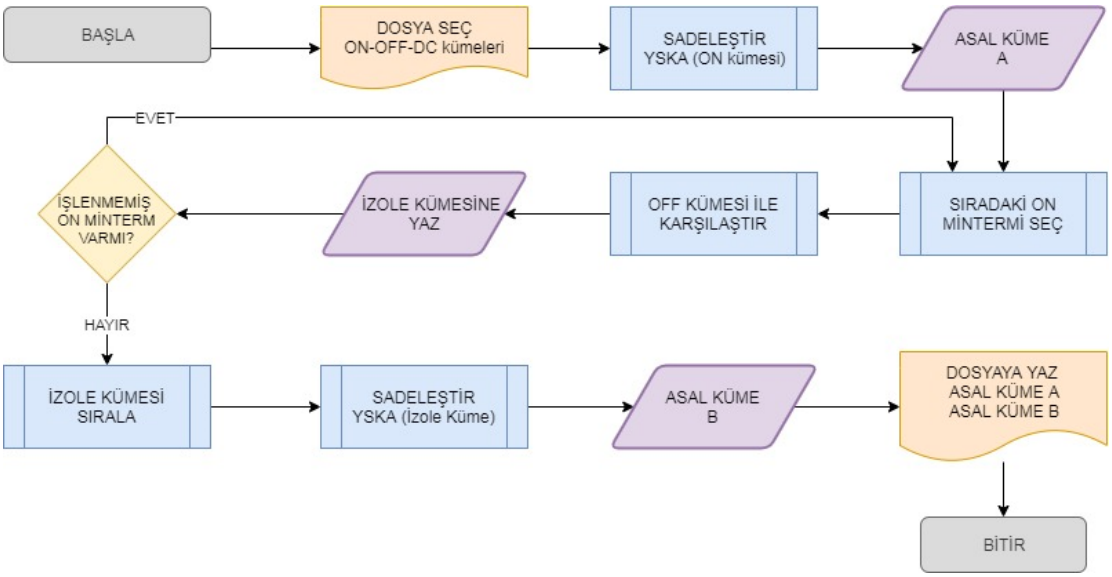
6.2. İzole Mintermlerin Tespiti

Bu araştırmada izole mintermler OFF mintermleri ile karşılaştırılarak tespit edilmiştir. İzole mintermlerin tespit edilmesi için lojik komşuluk ilişkilerine dayanan bir algoritma geliştirilmiştir. İzole mintermler diğer ON mintermlerinden uzakta, OFF mintermleriyle komşu olan mintermlerdir. Geliştirilen algoritma bütün ON mintermlerini OFF mintermleri ile karşılaştırmakta ve komşuluk ilişkilerine göre mintermlere izole değeri atamaktadır. Ardından ON mintermleri izole değerlerine göre sıralanmaktadır. Bu sayede sadeleştirmeye izole mintermlerden başlamak mümkün olmuştur.

Bu bölümde izole mintermlerin tespiti için geliştirilen algoritmanın YSKA ve KSKA algoritmalarını sonuç kalitesi ve çalışma zamanı açısından nasıl etkilediği sunulmuştur. Analiz sonuçlarında izole sıralı YSKA algoritmasının çalışma zamanı hesaplanırken izole sıralama algoritmasının harcadığı zaman dahil edilmiştir.

6.2.1. İzole mintermlerin tespitinin YSKA'ya etkisi

İzole mintermlerin tespitinin YSKA'ya etkisini tespit etmek için, izole mintermlerin tespiti algoritması geliştirilmiştir. Öncelikle rastgele sıralanmış fonksiyon dosyaları YSKA algoritması ile sadeleştirilmiş, ardından fonksiyon dosyaları izole değerlerine göre sıralanarak YSKA algoritmasıyla tekrar sadeleştirilmiştir. İçeriği aynı fakat sıralaması farklı (rastgele – izole değerlerine göre) fonksiyon dosyaları aynı YSKA algoritması ile sadeleştirilmiştir. Böylece mintermlerin sıralamasını değiştirmenin YSKA algoritması üzerindeki etkisi tespit edilmiştir. Şekil 6.1'de üst sıra rastgele sıralanmış fonksiyon dosyasının sadeleştirilmesi, orta sıra izole mintermlerin tespiti, alt sıra izole kümesinin sadeleştirilmesine ait işlem basamaklarını göstermektedir.



Şekil 6.1. İzole mintermlerin YSKA'ya etkisi

İzole mintermleri tespit edip sıralamak için geliştirilen algoritma işlem zamanı kullanırken, YSKA algoritmasının işini kolaylaştırarak sadeleştirme aşamasında zaman kazanmaktadır. Yapılan performans testinde YSKA algoritması MCNC benchmarklarını toplamda 559 AI'ye, 322,07 milisaniyede sadeleştirirken, izole minterm sıralı YSKA algoritması 546 AI'ye 281,23 milisaniyede sadeleştirmiştir. İzole sıralama algoritması 13 AI daha sade sonuca ulaşılmasını sağlarken, 40,84 milisaniye daha hızlı çalışmasını sağlamıştır. YSKA algoritmasının ON mintermlerini sadeleştirme oranı %82,86 iken, izole sıralı YSKA algoritmasında sadeleştirme oranı %83,26 olmuştur. İzole sıralamanın en büyük kazancı "t12" benchmarkında olmuştur. 266 ON mintermine sahip "t12" benchmarkı YSKA ile 50,91 milisaniyede 108 AI'ye sadeleşirken, izole sıralı YSKA ile 50,06 milisaniyede 103 AI'ye sadeleşmiştir.

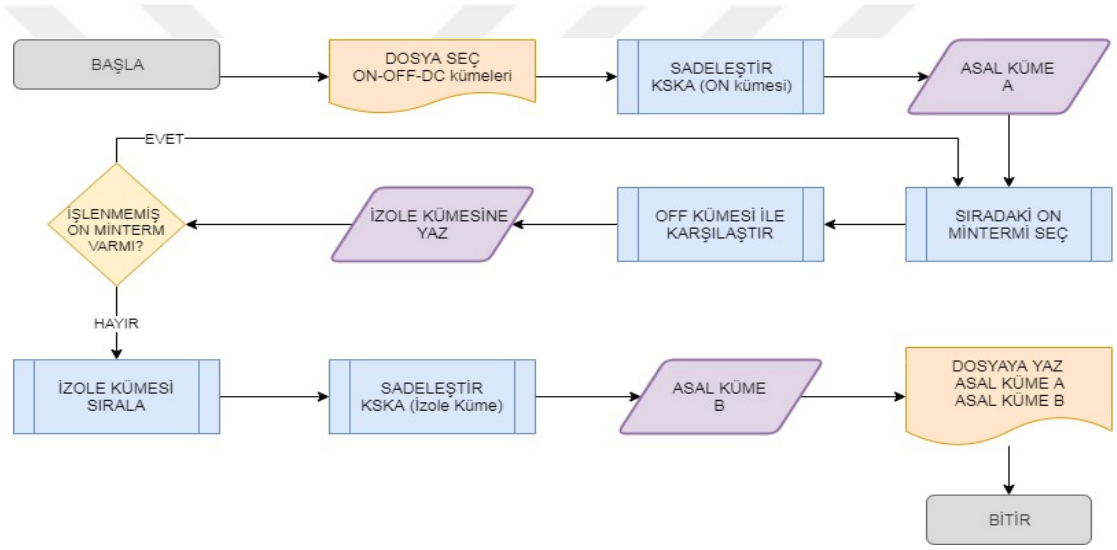
Çalışma zamanı açısından bakıldığında izole mintermlerin tespitinin YSKA'ya en büyük kazancı "rd84" benchmarkında olmuştur. Bu dosya YSKA ile 37,83 milisaniyede sadeleştirilirken, izole sıralı YSKA ile 27,60 milisaniyede sadeleştirilmiştir. İzole mintermlerin tespitinin YSKA'yı en çok yavaşlattığı benchmark "max1024" olmuştur. Bu dosya YSKA ile 4,73 milisaniyede sadeleştirilirken, izole sıralı YSKA ile 9,34 milisaniyede sadeleştirilmiştir. İzole mintermlerin tespitinin YSKA'ya etkisini gösteren performans testi sonuçları Çizelge 6.2'de verilmiştir.

Çizelge 6.2. İzole Mintermlerin Tespitinin YSKA'ya Etkisi

Benchmarklar	YSKA		İzole Sıralı YSKA		Aradaki Fark	
	AI Sayısı	Çalışma Zamanı (ms)	AI Sayısı	Çalışma Zamanı (ms)	AI Sayısı	Çalışma Zamanı (ms)
wim.txt	4	3,27	4	2,34	0	0,94
check3.txt	2	0,56	2	0,47	0	0,09
check2.txt	1	0,33	1	0,32	0	0,01
o4.txt	3	2,31	3	2,44	0	-0,12
check.txt	1	3,17	1	1,93	0	1,24
p82.txt	4	3,55	4	2,33	0	1,22
o5.txt	5	2,96	5	3,12	0	-0,17
o51.txt	6	3,89	5	2,48	1	1,41
squar.txt	2	3,58	2	2,00	0	1,58
new2.txt	2	0,41	2	0,37	0	0,04
m.txt	4	3,57	4	2,37	0	1,20
m5.txt	4	3,57	4	2,34	0	1,23
poperom.txt	7	3,99	7	2,76	0	1,23
sqr.txt	2	3,38	2	2,02	0	1,35
inc.txt	6	3,24	6	3,40	0	-0,16
sqn.txt	8	6,20	8	5,22	0	0,98
linrom.txt	24	11,61	24	9,77	0	1,84
max128.txt	8	6,08	8	4,99	0	1,10
z5xp1.txt	3	3,52	3	2,36	0	1,16
max3.txt	7	4,12	7	3,17	0	0,95
m4.txt	27	8,35	24	9,31	3	-0,96
m3.txt	16	8,83	16	7,94	0	0,89
exp1.txt	3	3,45	3	2,25	0	1,20
exp.txt	3	3,35	3	2,34	0	1,01
f51m.txt	23	10,93	23	10,47	0	0,45
e.txt	21	10,86	21	10,27	0	0,59
exps.txt	21	6,47	21	7,18	0	-0,71
rd84.txt	85	37,83	85	27,60	0	10,24
dist.txt	12	8,24	12	6,89	0	1,35
ex5.txt	2	3,46	2	2,41	0	1,05
mlp4.txt	9	6,84	9	6,15	0	0,70
root.txt	4	3,83	4	2,64	0	1,19
max4.txt	7	2,34	7	2,18	0	0,16
prom1.txt	23	7,28	22	7,18	1	0,09
min.txt	6	3,30	6	3,46	0	-0,16
prom2.txt	7	4,33	7	3,44	0	0,89
max512.txt	10	8,20	10	8,39	0	-0,19
apex4.txt	4	2,38	4	2,44	0	-0,06
max1024.txt	4	4,73	4	9,34	0	-4,61
t12.txt	108	50,91	103	50,06	5	0,84
br11.txt	3	2,41	3	2,55	0	-0,14
br2.txt	2	2,44	2	2,27	0	0,17
br1.txt	5	1,46	5	1,56	0	-0,10
t3.txt	6	4,18	6	3,02	0	1,15
pdc.txt	4	7,60	4	5,91	0	1,69
spla.txt	31	26,42	29	21,63	2	4,79
den.txt	5	4,39	4	2,36	1	2,03
bca.txt	1	1,23	1	0,36	0	0,86
bcc.txt	2	1,29	2	2,20	0	-0,91
bcb.txt	2	1,44	2	1,23	0	0,20

6.2.2. İzole mintermlerin tespitinin KSKA'ya etkisi

İzole mintermlerin tespitinin KSKA'ya etkisini tespit etmek için, izole mintermlerin tespiti algoritması geliştirilmiştir. Öncelikle rastgele sıralanmış fonksiyon dosyaları KSKA algoritması ile sadeleştirilmiş, ardından fonksiyon dosyaları izole değerlerine göre sıralanarak KSKA algoritmasıyla tekrar sadeleştirilmiştir. İçeriği aynı fakat sıralaması farklı (rastgele – izole değerlerine göre) fonksiyon dosyaları aynı algoritma ile sadeleştirilmiştir. Böylece mintermlerin sıralamasını değiştirmenin KSKA algoritması üzerindeki etkisi tespit edilmiştir. Şekil 6.2'de üst sıra rastgele sıralanmış fonksiyon dosyasının sadeleştirilmesi, orta sıra izole mintermlerin tespiti, alt sıra izole kümesinin sadeleştirilmesine ait işlem basamaklarını göstermektedir.



Şekil 6.2. İzole mintermlerin KSKA'ya etkisi

Yapılan performans testinde KSKA algoritması MCNC benchmarklarını toplamda 548 AI'ye, 2230,15 milisaniyede sadeleştirirken, izole minterm sıralı KSKA algoritması 542 AI'ye 2084,93 milisaniyede sadeleştirmiştir. İzole sıralama algoritması 6 AI daha sade sonuca ulaşılmasını sağlarken, 145,22 milisaniye daha hızlı çalışmasını sağlamıştır. KSKA algoritmasının ON mintermlerini sadeleştirme oranı %83,20 iken, izole sıralı KSKA algoritmasında sadeleştirme oranı %83,38 olmuştur. İzole sıralamanın en büyük kazancı "t12" ve "m3" benchmarklarında olmuştur. 266 ON mintermine sahip "t12" benchmarkı KSKA ile 76,45 milisaniyede 104 AI'ye sadeleşirken, izole sıralı KSKA ile 73,64 milisaniyede 102 AI'ye sadeleşmiştir. 98 ON mintermine sahip "m3" benchmarkı KSKA ile 16,94 milisaniyede 16 AI'ye sadeleşirken, izole sıralı KSKA ile

12,69 milisaniyede 14 AI'ye sadeleşmiştir. İzole mintermlerin tespitinin KSKA'ya etkisini gösteren performans testi sonuçları Çizelge 6.3'de verilmiştir.

Çizelge 6.3. İzole Mintermlerin Tespitinin KSKA'ya Etkisi

Benchmarklar	KSKA		İzole Sıralı KSKA		Aradaki Fark	
	AI Sayısı	Çalışma Zamanı (ms)	AI Sayısı	Çalışma Zamanı (ms)	AI Sayısı	Çalışma Zamanı (ms)
wim.txt	4	2,26	4	0,40	0	1,86
check3.txt	2	0,45	2	0,42	0	0,03
check2.txt	1	0,26	1	0,28	0	-0,02
o4.txt	3	2,45	3	2,72	0	-0,27
check.txt	1	3,43	1	0,07	0	3,36
p82.txt	4	5,16	4	0,56	0	4,60
o5.txt	5	3,57	5	2,93	0	0,63
o51.txt	5	5,61	5	0,63	0	4,98
squar.txt	2	2,29	2	0,12	0	2,17
new2.txt	2	0,38	2	0,39	0	-0,01
m.txt	4	2,60	4	0,40	0	2,20
m5.txt	4	2,90	4	0,41	0	2,49
poperom.txt	7	4,80	7	0,89	0	3,90
sqr.txt	2	2,33	2	0,14	0	2,20
inc.txt	6	3,19	6	3,10	0	0,09
sqn.txt	8	7,94	8	3,47	0	4,47
linrom.txt	26	13,26	25	10,78	1	2,49
max128.txt	8	7,35	8	2,88	0	4,47
z5xp1.txt	3	2,54	3	0,65	0	1,89
max3.txt	7	4,91	7	1,01	0	3,90
m4.txt	25	143,99	25	68,23	0	75,77
m3.txt	16	16,94	14	12,69	2	4,25
exp1.txt	3	3,83	3	0,31	0	3,52
exp.txt	3	2,63	3	1,63	0	1,00
f51m.txt	23	14,21	23	8,39	0	5,82
e.txt	21	14,71	21	8,17	0	6,54
exps.txt	21	7,84	21	7,49	0	0,34
rd84.txt	84	36,73	84	33,42	0	3,31
dist.txt	12	7,03	12	4,83	0	2,20
ex5.txt	2	2,81	2	0,53	0	2,28
mlp4.txt	9	5,76	9	4,53	0	1,22
root.txt	4	2,82	4	0,71	0	2,11
max4.txt	6	26,35	6	21,75	0	4,60
prom1.txt	22	1683,24	21	1711,14	1	-27,89
min.txt	6	3,26	6	3,56	0	-0,30
prom2.txt	7	4,36	7	1,74	0	2,62
max512.txt	10	6,91	10	8,50	0	-1,59
apex4.txt	4	2,19	4	2,34	0	-0,15
max1024.txt	4	3,85	4	9,99	0	-6,14
t12.txt	104	76,45	102	73,64	2	2,81
br11.txt	3	7,81	3	7,50	0	0,31
br2.txt	2	2,19	2	2,08	0	0,12
br1.txt	5	2,22	5	5,68	0	-3,46
t3.txt	6	3,74	6	1,05	0	2,68
pdc.txt	4	3,53	4	4,43	0	-0,90
spla.txt	29	27,92	29	19,55	0	8,37
den.txt	4	33,48	4	23,29	0	10,19
bca.txt	1	1,54	1	0,48	0	1,06
bcc.txt	2	1,83	2	3,45	0	-1,62
bcb.txt	2	2,33	2	1,60	0	0,73

Çalışma zamanı açısından bakıldığında izole mintermlerin tespitinin KSKA'ya en büyük kazancı "m4" benchmarkında olmuştur. Bu dosya KSKA ile 143,99 milisaniyede sadeleştirilirken, izole sıralı KSKA ile 68,23 milisaniyede sadeleştirilmiştir. İzole mintermlerin tespitinin KSKA'yı en çok yavaşlattığı benchmark "prom1" olmuştur. Bu dosya KSKA ile 1683,24 milisaniyede sadeleştirilirken, izole sıralı KSKA ile 1711,14 milisaniyede sadeleştirilmiştir.

6.3. Algoritmaların Paralel Programlamaya Uyarlanması

Bu bölümde paralel programlamanın YSKA, KSKA ve izole mintermlerin tespiti algoritmalarını çalışma zamanı açısından nasıl etkilediği sunulmuştur. Seri programlanmış YSKA algoritması, paralel programlanmış YSKA algoritması ile, seri programlanmış KSKA algoritması, paralel programlanmış KSKA algoritması ile karşılaştırılmış, izole sıralı algoritmalar üzerinde paralel programlamanın performans etkisi değerlendirilmiştir.

6.3.1. Yakın sonuç kapsama algoritmasının paralel programlanması

Bir algoritmanın farklı işlemcilerde aynı anda çalışmasını sağlamanın CPU işlem zamanı açısından bir maliyeti vardır. Fakat bir algoritmanın birden fazla işlemci kullanması çalışma zamanını kısaltmaktadır. Paralel programlamada önemli olan algoritmanın paralel programalamaya ne kadar uygun olduğu, verinin yâda program parçacıklarının ne kadar eş zamanlı çalıştırılabileceğidir. Yapılan performans testinde YSKA algoritması MCNC benchmarklarını toplamda 559 AI'ye, 322,07 milisaniyede sadeleştirirken, paralel programlanmış YSKA algoritması 559 AI'ye 248,39 milisaniyede sadeleştirmiştir. Paralel programlanmış YSKA algoritması 73,68 milisaniye daha hızlı çalışarak, sadeleştirme süresini %22,88 kısaltmıştır.

Çalışma zamanı açısından bakıldığında paralel programlamanın YSKA'ya en büyük kazancı %73,43 ile "exps" benchmarkında olmuştur. Bu dosya YSKA ile 6,47 milisaniyede sadeleştirilirken, paralel programlanmış YSKA ile 1,72 milisaniyede sadeleştirilmiştir. Paralel programlamanın YSKA'yı en çok yavaşlattığı benchmark %109,69 ile "bcc" olmuştur. Bu dosya YSKA ile 1,29 milisaniyede sadeleştirilirken, paralel programlanmış YSKA ile 2,70 milisaniyede sadeleştirilmiştir. Paralel

programlamanın YSKA'ya etkisini gösteren performans testi sonuçları Çizelge 6.4'de verilmiştir.

Çizelge 6.4. Paralel Programlamanın YSKA'ya Etkisi

Benchmarklar	YSKA		Paralel YSKA		Aradaki Fark	
	AI Sayısı	Çalışma Zamanı (ms)	AI Sayısı	Çalışma Zamanı (ms)	Milisaniye	Yüzde (%)
wim.txt	4	3,27	4	3,52	-0,25	-8%
check3.txt	2	0,56	2	0,48	0,08	15%
check2.txt	1	0,33	1	0,28	0,05	14%
o4.txt	3	2,31	3	1,36	0,95	41%
check.txt	1	3,17	1	5,86	-2,69	-85%
p82.txt	4	3,55	4	5,94	-2,40	-68%
o5.txt	5	2,96	5	1,50	1,45	49%
o51.txt	6	3,89	6	4,96	-1,07	-28%
squar.txt	2	3,58	2	3,78	-0,20	-6%
new2.txt	2	0,41	2	0,27	0,14	34%
m.txt	4	3,57	4	3,22	0,34	10%
m5.txt	4	3,57	4	3,13	0,43	12%
poperom.txt	7	3,99	7	6,67	-2,68	-67%
sqr.txt	2	3,38	2	3,60	-0,22	-7%
inc.txt	6	3,24	6	1,33	1,91	59%
sqn.txt	8	6,20	8	5,07	1,13	18%
linrom.txt	24	11,61	24	18,08	-6,47	-56%
max128.txt	8	6,08	8	4,95	1,13	19%
z5xp1.txt	3	3,52	3	3,29	0,23	6%
max3.txt	7	4,12	7	5,91	-1,79	-43%
m4.txt	27	8,35	27	2,84	5,52	66%
m3.txt	16	8,83	16	11,19	-2,37	-27%
exp1.txt	3	3,45	3	5,36	-1,91	-55%
exp.txt	3	3,35	3	3,18	0,17	5%
f51m.txt	23	10,93	23	10,12	0,80	7%
e.txt	21	10,86	21	8,33	2,53	23%
exps.txt	21	6,47	21	1,72	4,75	73%
rd84.txt	85	37,83	85	21,60	16,23	43%
dist.txt	12	8,24	12	3,76	4,48	54%
ex5.txt	2	3,46	2	3,46	0,01	0%
mlp4.txt	9	6,84	9	3,66	3,18	46%
root.txt	4	3,83	4	3,42	0,41	11%
max4.txt	7	2,34	7	2,21	0,14	6%
prom1.txt	23	7,28	23	4,72	2,55	35%
min.txt	6	3,30	6	2,80	0,50	15%
prom2.txt	7	4,33	7	6,22	-1,89	-44%
max512.txt	10	8,20	10	3,32	4,89	60%
apex4.txt	4	2,38	4	0,66	1,72	72%
max1024.txt	4	4,73	4	1,36	3,37	71%
t12.txt	108	50,91	108	27,78	23,13	45%
br11.txt	3	2,41	3	0,81	1,60	66%
br2.txt	2	2,44	2	1,41	1,03	42%
br1.txt	5	1,46	5	1,18	0,29	20%
t3.txt	6	4,18	6	3,38	0,80	19%
pdc.txt	4	7,60	4	2,93	4,67	61%
spla.txt	31	26,42	31	20,46	5,96	23%
den.txt	5	4,39	5	1,48	2,91	66%
bca.txt	1	1,23	1	0,51	0,71	58%
bcc.txt	2	1,29	2	2,70	-1,41	-110%
bcb.txt	2	1,44	2	2,61	-1,17	-82%

6.3.2. Kesin sonuç kapsama algoritmasının paralel programlanması

Yapılan performans testinde KSKA algoritması MCNC benchmarklarını toplamda 548 AI'ye, 2230,15 milisaniyede sadeleştirirken, paralel programlanmış KSKA algoritması 548 AI'ye 2256,54 milisaniyede sadeleştirmiştir. Paralel programlanmış KSKA algoritması 26,39 milisaniye daha yavaş çalışarak, sadeleştirme süresini %1,18 arttırmıştır.

Çalışma zamanı açısından bakıldığında paralel programlamanın KSKA'ya en büyük kazancı %68,26 ile "dist" benchmarkında olmuştur. Bu dosya KSKA ile 7,03 milisaniyede sadeleştirilirken, paralel programlanmış KSKA ile 2,23 milisaniyede sadeleştirilmiştir. Paralel programlamanın KSKA'yı en çok yavaşlattığı benchmark %82,73 ile "bcb" olmuştur. Bu dosya KSKA ile 2,33 milisaniyede sadeleştirilirken, paralel programlanmış KSKA ile 4,26 milisaniyede sadeleştirilmiştir. Paralel programlamanın KSKA'ya etkisini gösteren performans testi sonuçları Çizelge 6.5'de verilmiştir.

6.3.3. İzole minterm algoritmasının paralel programlanması

Bu bölümde paralel programlamanın izole sıralı YSKA algoritması ile izole sıralı KSKA algoritmalarını çalışma zamanı açısından nasıl etkilediği sunulmuştur. Yapılan performans testinde izole sıralı YSKA algoritması MCNC benchmarklarını toplamda 546 AI'ye, 281,22 milisaniyede sadeleştirirken, paralel programlanmış izole sıralı YSKA algoritması 546 AI'ye 142,38 milisaniyede sadeleştirmiştir. Paralel programlanmış izole sıralı YSKA algoritması 138,84 milisaniye daha hızlı çalışarak, sadeleştirme süresini %49,37 kısaltmıştır. Çalışma zamanı açısından bakıldığında paralel programlamanın izole sıralı YSKA'ya en büyük kazancı %75,61 ile "f51m" benchmarkında olmuştur. Bu dosya izole sıralı YSKA ile 10,47 milisaniyede sadeleştirilirken, paralel programlanmış izole sıralı YSKA ile 2,55 milisaniyede sadeleştirilmiştir. Paralel programlamanın izole sıralı YSKA'yı en çok yavaşlattığı benchmark %59,21 ile "bcc" olmuştur. Bu dosya izole sıralı YSKA ile 2,20 milisaniyede sadeleştirilirken, paralel programlanmış izole sıralı YSKA ile 3,50 milisaniyede sadeleştirilmiştir. Paralel programlamanın izole sıralı YSKA'ya etkisini gösteren performans testi sonuçları Çizelge 6.6'de verilmiştir.

Çizelge 6.5. Paralel Programlamanın KSKA'ya Etkisi

Benchmarklar	KSKA		Paralel KSKA		Aradaki Fark	
	AI Sayısı	Çalışma Zamanı (ms)	AI Sayısı	Çalışma Zamanı (ms)	Milisaniye	Yüzde (%)
wim.txt	4	2,26	4	1,81	0,45	20%
check3.txt	2	0,45	2	0,19	0,25	56%
check2.txt	1	0,26	1	0,13	0,13	51%
o4.txt	3	2,45	3	1,15	1,31	53%
check.txt	1	3,43	1	3,02	0,41	12%
p82.txt	4	5,16	4	3,40	1,75	34%
o5.txt	5	3,57	5	1,33	2,23	63%
o51.txt	5	5,61	5	2,94	2,67	48%
squar.txt	2	2,29	2	1,82	0,47	21%
new2.txt	2	0,38	2	0,14	0,23	62%
m.txt	4	2,60	4	2,27	0,33	13%
m5.txt	4	2,90	4	2,22	0,68	23%
poperom.txt	7	4,80	7	3,06	1,74	36%
sqr.txt	2	2,33	2	2,19	0,14	6%
inc.txt	6	3,19	6	1,15	2,04	64%
sqn.txt	8	7,94	8	3,34	4,61	58%
linrom.txt	26	13,26	26	6,85	6,41	48%
max128.txt	8	7,35	8	3,00	4,35	59%
z5xp1.txt	3	2,54	3	1,85	0,69	27%
max3.txt	7	4,91	7	2,88	2,03	41%
m4.txt	25	143,99	25	139,65	4,35	3%
m3.txt	16	16,94	16	16,48	0,46	3%
exp1.txt	3	3,83	3	2,86	0,98	25%
exp.txt	3	2,63	3	1,99	0,64	24%
f51m.txt	23	14,21	23	7,63	6,58	46%
e.txt	21	14,71	21	9,26	5,45	37%
exps.txt	21	7,84	21	4,79	3,05	39%
rd84.txt	84	36,73	84	26,66	10,07	27%
dist.txt	12	7,03	12	2,23	4,80	68%
ex5.txt	2	2,81	2	2,63	0,18	6%
mlp4.txt	9	5,76	9	5,21	0,55	10%
root.txt	4	2,82	4	2,45	0,37	13%
max4.txt	6	26,35	6	29,93	-3,58	-14%
prom1.txt	22	1683,24	22	1806,29	-123,05	-7%
min.txt	6	3,26	6	1,45	1,81	56%
prom2.txt	7	4,36	7	3,25	1,11	25%
max512.txt	10	6,91	10	2,31	4,60	67%
apex4.txt	4	2,19	4	0,74	1,45	66%
max1024.txt	4	3,85	4	1,66	2,19	57%
t12.txt	104	76,45	104	72,03	4,42	6%
br11.txt	3	7,81	3	7,84	-0,03	0%
br2.txt	2	2,19	2	1,28	0,91	42%
br1.txt	5	2,22	5	1,89	0,33	15%
t3.txt	6	3,74	6	2,37	1,37	37%
pdc.txt	4	3,53	4	1,73	1,79	51%
spla.txt	29	27,92	29	17,89	10,03	36%
den.txt	4	33,48	4	30,30	3,18	9%
bca.txt	1	1,54	1	1,44	0,10	6%
bcc.txt	2	1,83	2	3,30	-1,47	-80%
bcb.txt	2	2,33	2	4,26	-1,93	-83%

Çizelge 6.6. Paralel Programlamanın İzole Sıralı Algoritmalara Etkisi

Benchmarklar	İzole Sıralı YSKA		İzole Sıralı KSKA		Paralel İS. YSKA		Paralel İS. KSKA	
	AI	Süre (ms)	AI	Süre (ms)	AI	Süre (ms)	AI	Süre (ms)
wim.txt	4	2,34	4	0,40	4	1,22	4	0,15
check3.txt	2	0,47	2	0,42	2	0,17	2	0,12
check2.txt	1	0,32	1	0,28	1	0,15	1	0,15
o4.txt	3	2,44	3	2,72	3	1,10	3	1,11
check.txt	1	1,93	1	0,07	1	1,80	1	0,04
p82.txt	4	2,33	4	0,56	4	2,43	4	0,19
o5.txt	5	3,12	5	2,93	5	1,22	5	1,34
o51.txt	5	2,48	5	0,63	5	2,29	5	0,21
squar.txt	2	2,00	2	0,12	2	1,25	2	0,06
new2.txt	2	0,37	2	0,39	2	0,17	2	0,15
m.txt	4	2,37	4	0,40	4	1,66	4	0,10
m5.txt	4	2,34	4	0,41	4	1,55	4	0,11
poperom.txt	7	2,76	7	0,89	7	2,17	7	0,36
sqr.txt	2	2,02	2	0,14	2	1,51	2	0,09
inc.txt	6	3,40	6	3,10	6	1,16	6	1,13
sqn.txt	8	5,22	8	3,47	8	2,17	8	1,26
linrom.txt	24	9,77	25	10,78	24	3,11	25	3,06
max128.txt	8	4,99	8	2,88	8	1,76	8	1,37
z5xp1.txt	3	2,36	3	0,65	3	1,40	3	0,30
max3.txt	7	3,17	7	1,01	7	1,43	7	0,29
m4.txt	24	9,31	25	68,23	24	2,80	25	63,84
m3.txt	16	7,94	14	12,69	16	2,71	14	8,79
exp1.txt	3	2,25	3	0,31	3	2,29	3	0,11
exp.txt	3	2,34	3	1,63	3	1,64	3	0,96
f51m.txt	23	10,47	23	8,39	23	2,55	23	4,27
e.txt	21	10,27	21	8,17	21	2,72	21	3,51
exps.txt	21	7,18	21	7,49	21	2,46	21	3,22
rd84.txt	85	27,60	84	33,42	85	12,15	84	15,21
dist.txt	12	6,89	12	4,83	12	3,82	12	1,63
ex5.txt	2	2,41	2	0,53	2	1,89	2	0,42
mlp4.txt	9	6,15	9	4,53	9	2,43	9	1,53
root.txt	4	2,64	4	0,71	4	1,84	4	0,38
max4.txt	7	2,18	6	21,75	7	1,98	6	27,35
prom1.txt	22	7,18	21	1711,14	22	3,46	21	1797,40
min.txt	6	3,46	6	3,56	6	1,40	6	1,32
prom2.txt	7	3,44	7	1,74	7	1,66	7	0,55
max512.txt	10	8,39	10	8,50	10	4,43	10	2,35
apex4.txt	4	2,44	4	2,34	4	0,71	4	0,69
max1024.txt	4	9,34	4	9,99	4	3,73	4	4,34
t12.txt	103	50,06	102	73,64	103	26,38	102	57,55
br11.txt	3	2,55	3	7,50	3	1,23	3	6,77
br2.txt	2	2,27	2	2,08	2	1,15	2	1,18
br1.txt	5	1,56	5	5,68	5	0,51	5	5,20
t3.txt	6	3,02	6	1,05	6	2,34	6	0,37
pdc.txt	4	5,91	4	4,43	4	2,44	4	3,55
spla.txt	29	21,63	29	19,55	29	15,55	29	12,48
den.txt	4	2,36	4	23,29	4	0,93	4	29,10
bca.txt	1	0,36	1	0,48	1	0,26	1	0,23
bcc.txt	2	2,20	2	3,45	2	3,50	2	3,07
bcb.txt	2	1,23	2	1,60	2	1,72	2	2,40

Yapılan performans testinde izole sıralı KSKA algoritması MCNC benchmarklarını toplamda 542 AI'ye, 2084,93 milisaniyede sadeleştirirken, paralel programlanmış izole sıralı KSKA algoritması 542 AI'ye 2071,40 milisaniyede sadeleştirmiştir. Paralel programlanmış izole sıralı KSKA algoritması 13,53 milisaniye daha hızlı çalışarak, sadeleştirme süresini %0,65 kısaltmıştır. Çalışma zamanı açısından bakıldığında paralel programlamanın izole sıralı KSKA'ya en büyük kazancı %74,84 ile "m" benchmarkında olmuştur. Bu dosya izole sıralı KSKA ile 0,40 milisaniyede sadeleştirilirken, paralel programlanmış izole sıralı KSKA ile 0,10 milisaniyede sadeleştirilmiştir. Paralel programlamanın izole sıralı KSKA'yı en çok yavaşlattığı benchmark %50,52 ile "bcb" olmuştur. Bu dosya izole sıralı KSKA ile 1,60 milisaniyede sadeleştirilirken, paralel programlanmış izole sıralı KSKA ile 2,40 milisaniyede sadeleştirilmiştir.

6.4. Çok Büyük Dosyalarda Sadeleştirme İşlemleri

Çok büyük dosyaların sadeleştirilmesine genellikle bilimsel araştırmalarda ve endüstriyel alanlarda ihtiyaç duyulabilir. Geliştirilen program 64 değişken sayısına kadar çok büyük dosyaları sadeleştirebilmektedir. Çok büyük dosyalar, genelde kullanılan benchmarklara kıyasla değişken sayısı, minterm sayıları, minterm sıralaması açısından farklı olabilirler. Ayrıca büyük dosyalarda program çöp toplama işlemine ihtiyaç duymakta, artık gerekmeyen hafızayı boşaltmak için işlem zamanı harcamaktadır. Bu sebeple sadeleştirme sonuçlarında büyük farklılıklar ortaya çıkmaktadır. Bu farklılıkların araştırma sonuçlarını etkilememesi için büyük boyutlu dosyalar ayrı bir başlıkta incelenmiştir.

Bu bölümde giriş değişkeni sayısı 27 ile 63 arasında olan 15 büyük benchmark sadeleştirilmiş ve sonuçları analiz edilmiştir. Çizelge 6.7'de verildiği gibi giriş değişkeni sayısı en fazla olan (63 değişkenli) ve en çok minterme sahip benchmark 622 mintermli "x7dn63" dür. Araştırma sonuçlarına göre geliştirilen algoritma çok büyük dosyaları kısa sürede sadeleştirmektedir. Yapılan sadeleştirme işlemlerinde çöp toplama prosedürü sayesinde programın en fazla 45,9 MB hafıza kullandığı görülmüştür. Toplamda 582 ON mintermini YSKA algoritması %79,73 sadeleştirerek 118 AI'ya, KSKA algoritması %80,93 sadeleştirerek 111 AI'ya indirgemiştir.

15 büyük dosya için izole mintermlerin tespiti, YSKA algoritmasının bulduğu implikantların toplamını 118'den, 116'e düşürmüş, işlem süresini ise 420,02'den

383,70'e indirmiştir. İzole mintermlerin tespiti, sonuç kalitesini %1,69 arttırmış, işlem süresini ise %8,65 azaltmıştır. YSKA algoritmasında izole mintermlerin tespiti, "chkn" benchmarkının sadeleştirme süresini %55,34 azaltırken, "mish" benchmarkının sadeleştirme süresini %12,85 arttırmıştır. Bu farkın çöp toplama işleminden kaynaklandığı değerlendirilmektedir. Çok büyük benchmarklarda izole sıralamanın etkisini gösteren performans testi sonuçları Çizelge 6.7'de verilmiştir.

Çizelge 6.7. Çok Büyük Benchmarklarda İzole Sıralamanın Etkisi

Benchmark	Değişken Sayısı	x_{ON}	y_{OFF}	YSKA		İzole YSKA		KSKA		İzole KSKA	
				AI	Süre (ms)	AI	Süre (ms)	AI	Süre (ms)	AI	Süre (ms)
main.txt	27	104	67	11	19,86	11	13,18	11	20,60	11	12,81
chkn.txt	29	57	53	7	10,63	6	4,75	5	241,74	5	54,63
b3.txt	32	51	111	8	7,14	8	5,97	8	39,32	8	8,55
int4.txt	32	12	150	2	1,26	2	1,24	2	2,00	2	1,32
in3.txt	35	17	50	4	1,33	4	1,26	4	1,59	4	1,07
jbp.txt	36	5	113	5	2,13	5	2,18	5	3,81	5	2,20
x6dn.txt	39	34	60	11	15,25	11	14,00	11	46,96	11	52,23
signet.txt	39	35	67	8	52,32	8	43,66	7	3.741,62	7	3.760,18
xparc.txt	41	171	367	23	142,42	23	141,94	22	3.815,25	22	3.341,15
mish.txt	43	3	31	1	4,02	1	4,54	1	4,69	1	2,24
ti.txt	47	34	131	15	10,82	14	10,46	13	14,84	13	9,39
accpla.txt	47	34	131	14	10,95	14	10,22	13	11,91	13	10,05
misg.txt	56	6	8	6	4,89	6	5,32	6	8,05	6	0,72
soar.txt	60	5	284	1	64,84	1	60,33	1	61,24	1	61,86
x7dn63.txt	63	14	608	2	72,14	2	64,66	2	66,19	2	66,08

KSKA algoritmasında izole mintermlerin tespiti sonuç kalitesini etkilemezken, toplam çalışma süresini 8079,82 ms.'den 7384,50 ms.'ye düşürerek %8,61 azaltmıştır. İzole minterm tespitinin KSKA çalışma süresini en fazla indirgediği benchmark %91,02 ile "misg" olmuştur. İzole minterm tespitinin KSKA çalışma süresini en fazla yükselttiği benchmark %11,22 ile "x6dn" olmuştur.

Çizelge 6.8'de görüldüğü gibi algoritmaların paralel programlanması, sonuç kalitesi üzerinde bir değişiklik yapmamış, çalışma sürelerini ise önemli ölçüde arttırmıştır. Algoritmaların paralel çalıştırılmasında daha fazla çöp toplama ihtiyacı olması, çalışma süresini önemli ölçüde arttırmaktadır. YSKA algoritmasının paralel programlanması 15 büyük benchmark için toplam çalışma süresini %111,24 arttırmıştır. Ayrıca izole sıralı YSKA algoritmasında bu oran %128,93 olmuştur. Paralel

programlamanın KSKA algoritmasında kullanılması toplam çalışma süresini %36,10 arttırırken, bu oran izole sıralı KSKA algoritmasında %35,94 olmuştur.

Çizelge 6.8. Çok Büyük Benchmarklarda Paralel Programlamanın Etkisi

Benchmark	Değişken Sayısı	x_{ON}	y_{OFF}	Paralel YSKA		Paralel İzole YSKA		Paralel KSKA		Paralel İzole KSKA	
				AI	Süre (ms)	AI	Süre (ms)	AI	Süre (ms)	AI	Süre (ms)
main.txt	27	104	67	11	20,84	11	19,69	11	28,78	11	37,92
chkn.txt	29	57	53	7	32,32	6	22,08	5	484,35	5	182,98
b3.txt	32	51	111	8	10,33	8	6,64	8	52,92	8	13,54
int4.txt	32	12	150	2	2,62	2	2,22	2	2,54	2	1,72
in3.txt	35	17	50	4	2,07	4	1,60	4	1,54	4	2,13
jbp.txt	36	5	113	5	5,70	5	5,66	5	3,84	5	3,19
x6dn.txt	39	34	60	8	106,57	8	118,30	7	4.562,07	7	4.448,34
signet.txt	39	35	67	11	34,28	11	24,31	11	83,29	11	85,93
xparc.txt	41	171	367	23	328,78	23	365,95	22	5.440,46	22	4.997,54
mish.txt	43	3	31	1	11,75	1	1,33	1	3,24	1	0,16
ti.txt	47	34	131	14	23,70	14	19,88	13	30,85	13	20,07
accpla.txt	47	34	131	15	19,65	14	18,01	13	38,77	13	24,47
misg.txt	56	6	8	6	6,50	6	1,28	6	3,47	6	0,12
soar.txt	60	5	284	1	116,62	1	109,74	1	108,15	1	99,56
x7dn63.txt	63	14	608	2	165,51	2	161,71	2	152,56	2	121,08

Paralel YSKA algoritmasında izole mintermlerin tespiti, toplam çalışma süresini 887,22'den 878,42'ye indirgeyerek %0,99 azaltmıştır. Paralel KSKA algoritmasında izole mintermlerin tespiti, toplam çalışma süresini 10996,83'den 10038,76'ya indirgeyerek %8,71 azaltmıştır.

6.5. Geliştirilen Programın Bellek Kullanımı

Geliştirilen sadeleştirme algoritmalarının kullanıldığı "Lojik Fonksiyonları Sadeleştir" programı Windows üzerinde çalışan bir masaüstü uygulamasıdır. Program .NET platformunda C# dili kullanılarak geliştirilmiştir. Bu platformun en önemli özelliklerinden birisi Çöp Toplama (Garbage Collection) işlemini başarılı bir şekilde yapmasıdır. Sadeleştirilecek program ne kadar büyük ya da karmaşık olursa olsun, gereksiz veri parçacıkları temizlenerek hafızanın şişmesine fırsat vermemektedir (Strauss, 2019).

Çok büyük benchmarkların sadeleştirilmesinde kullanılan bellek miktarları Çizelge 6.9'da verilmiştir. Program yüklendiğinde 26,9 MB. bellek kullanmaktadır.

Sadeleştirilecek dosyayı yükleme işlemi için "Windows dosya gezgini" isimli dosya yöneticisi açılır. Çizelge 6.9 incelendiğinde, dosya yükleme işlemi esnasında en fazla 41,60 MB., en az 40,90 MB. ve ortalama 41,21 MB. bellek kullanıldığı görülmüştür.

Çizelge 6.9. Çok Büyük Benchmarklarda Bellek Kullanımı

Benchmark	Değişken Sayısı	x_{ON}	y_{OFF}	Program Açılışı (MB)	Dosya Yükleme (MB)	Dosya Sadeleştirme (MB)
main.txt	27	104	67	26,9	41,10	40,20
chkn.txt	29	57	53	26,9	41,20	43,20
b3.txt	32	51	111	26,9	41,30	39,90
int4.txt	32	12	150	26,9	41,20	41,40
in3.txt	35	17	50	26,9	41,20	41,70
jbp.txt	36	5	113	26,9	40,90	44,00
x6dn.txt	39	34	60	26,9	41,30	44,80
signet.txt	39	35	67	26,9	41,30	40,80
xparc.txt	41	171	367	26,9	41,30	45,90
mish.txt	43	3	31	26,9	41,00	39,10
ti.txt	47	34	131	26,9	41,30	43,30
accpla.txt	47	34	131	26,9	41,60	44,40
misg.txt	56	6	8	26,9	40,90	39,10
soar.txt	60	5	284	26,9	41,50	43,40
x7dn63.txt	63	14	608	26,9	41,10	44,60

Dosyaların sadeleştirilmesi işlemlerinde en fazla 45,90 MB., en az 39,10 MB ve ortalama 42,39 MB. bellek kullanıldığı görülmüştür. Büyük benchmarkların sadeleştirilmesinde bile program 46 MB. bellek kullanımını aşmamıştır. Bellek kullanımı incelendiğinde programda çöp toplama işleminin sürekli aktif olduğu ve artık ihtiyaç olmayan her verinin bellekten temizlendiği gözlemlenmiştir. Sık yapılan çöp toplama işlemi her nekadar programın çalışma süresini uzatsada, hafızanın şişmesine ve bilgisayarın kilitlenmesine müsaade etmemektedir.

6.6. Tartışma

Bu bölümde geliştirilen 8 algoritmanın 50 standart MCNC benchmarkı üzerindeki performansı değerlendirilmiş ve birbirleri ile karşılaştırılmıştır. Benchmarkların giriş değişkeni, ON ve OFF minterm sayıları Çizelge 6.1'de verilmiştir. Performans testinde 8 GB RAM belleği bulunan ve 1,6 GHz hızında çalışan i5 işlemcili bir bilgisayar kullanılmıştır. Her algoritma test edilmeden önce bütün değişkenler sıfırlanmış ve

algoritma üzerinde çöp toplama işlemi yapılmıştır. Algoritmaların çalışma zamanını hesaplamasında, performans testleri için hazırlanan "System.Diagnostics" kütüphanesinin "stopwatch" özelliği kullanılmıştır. Test sonuçları YSKA ve KSKA algoritmalarının karşılaştırılması, random ve izole sıralı algoritmaların karşılaştırılması, seri programlanmış ve paralel programlanmış algoritmaların karşılaştırılması olmak üzere 3 başlıkta incelenmiştir.

6.6.1. Yakın ve kesin sonuç kapsama algoritmalarının karşılaştırılması

MCNC benchmarkları fonksiyon sadeleştirme testleri için özel olarak hazırlanmış, farklı özelliklerdeki dosyalardır. Benchmarklar içindeki ON, OFF ve DC minterm sayılarının ve içeriklerinin değişmesi, bir algoritmayı bazı dosyalarda yüksek performans göstermesini sağlarken, bazı dosyalarda düşük performans göstermesini sağlamaktadır. YSKA algoritması işlediği ON mintermleri için en verimli hesaplanan AI'yi oluştururken, KSKA algoritması bütün ihtimalleri değerlendirerek, emin olmadığı sürece ON mintermlerinin işlenmesini ertelemektedir. Bu sebeple bir mintermi birkaç kez işlemesi/kontrol etmesi gerekmektedir. Tekrar ve tekrar ele alınan ON mintermlerinin işlenmesi daha sade sonuca ulaşılmasını sağlarken, daha uzun çalışma zamanına sebep olmaktadır.

YSKA algoritmasının 50 MCNC benchmarkında bulunan ON mintermleri sadeleştirme oranı %82,86 iken, KSKA algoritmasında bu oran %83,20 olmuştur. YSKA ve KSKA algoritmaları sonuç kalitesi açısından karşılaştırıldığında 41 benchmarkta eşit sayıda AI sayısı, 9 benchmarkta ise KSKA algoritması daha iyi sonuç bulmuştur. Fakat YSKA algoritmasının toplam hesaplama süresi 322,07 milisaniye iken, KSKA algoritmasında bu oran 2230,15 milisaniyedir. KSKA algoritması YSKA algoritmasına göre 6,92 kat daha yavaş çalışmış ve %0,34 daha iyi sadeleştirme yapmıştır. KSKA algoritmasının 50 dosya üzerinde ortalama çalışma zamanı 44,6 milisaniye olurken, YSKA algoritmasında bu sayı 6,44 milisaniyedir.

YSKA ve KSKA algoritmalarının en kısa sürede sadeleştirdiği benchmark sırasıyla 0,33 milisaniye ve 0,26 milisaniye ile "check2" olmuştur. YSKA algoritmasının en uzun sürede sadeleştirdiği benchmark 50,91 milisaniye ile "t12" olurken, KSKA algoritması 1683,24 milisaniyede "prom1""i sadeleştirmiştir. Performans sonuçları incelendiğinde giriş değişkeni ve ON mintermi sayısının fazla olmadığı benchmarklarda YSKA ve KSKA algoritmalarının benzer sonuçlar verdiği görülmüştür. Yüksek sayıda

ON ve OFF mintermi içeren “m4”, “t12” ve “spla” gibi benchmarklarda KSKA yüksek çalışma süresine rağmen daha sade sonuçlara ulaşmıştır. Performans sonuçlarına göre zamanın kısıtlı olduğu ya da çok fazla ON – OFF mintermi içermeyen dosyaların sadeleştirilmesinde YSKA tercih edilebilir. Büyük dosyaların sadeleştirilmesinde ve yeterli zaman varsa KSKA tercih edilebilir. YSKA ve KSKA algoritmalarının performans testi sonuçları Çizelge 6.1’de verilmiştir.

6.6.2. İzole mintermlerin tespitinin sadeleştirmeye etkisi

İzole mintermleri tespit edip sıralamak için geliştirilen algoritma sıralama aşamasında işlem zamanı kullanırken, YSKA ve KSKA algoritmalarının işini kolaylaştırarak sadeleştirme aşamasında zaman kazandırmaktadır. Bazı dosyalar zaten sıralı olduğu için bu dosyalar üzerinde tekrar sıralama yapılması sadece zaman kaybettirmektedir.

İzole mintermlerin tespiti algoritması YSKA algoritmasının 13 AI daha sade sonuca ulaşmasını ve 40,84 milisaniye daha hızlı çalışmasını sağlamıştır. Geliştirilen algoritma YSKA üzerinde %2,33 sonuç kalitesi artışı, %12,68 çalışma zamanı kazanımı sağlamıştır. “t12” benchmarkının YSKA ile 50,91 milisaniyede 108 AI’ye sadeleşirken, izole sıralı YSKA ile 50,06 milisaniyede 103 AI’ye sadeleşmesi dikkat çekmiştir. Diğer taraftan “max1024” benchmarkı YSKA ile 4,73 milisaniyede sadeleşirken, izole sıralı YSKA ile 9,34 milisaniyede sadeleşmiştir. İzole mintermlerin tespitinin YSKA’ya etkisini gösteren performans testi sonuçları Çizelge 6.2’de verilmiştir.

İzole mintermlerin tespiti algoritması KSKA algoritmasının 6 AI daha sade sonuca ulaşmasını ve 145,22 milisaniye daha hızlı çalışmasını sağlamıştır. Geliştirilen algoritma KSKA üzerinde %1,09 sonuç kalitesi artışı, %6,51 çalışma zamanı kazanımı sağlamıştır. “m3” benchmarkının KSKA ile 16,94 milisaniyede 16 AI’ye sadeleşirken, izole sıralı KSKA ile 12,69 milisaniyede 14 AI’ye sadeleşmesi dikkat çekmiştir. Diğer taraftan “max1024” benchmarkı KSKA ile 3,85 milisaniyede sadeleşirken, izole sıralı KSKA ile 9,99 milisaniyede sadeleşmiştir. İzole mintermlerin tespitinin KSKA’ya etkisini gösteren performans testi sonuçları Çizelge 6.3’de verilmiştir.

Performans testi sonuçlarına göre izole mintermlerin tespiti hem YSKA hemde KSKA algoritmalarının sonuç kalitelerini arttırmış ve genel ortalamada çalışma zamanlarını düşürmüştür. Genel olarak bakıldığında izole minterm sıralama algoritması sıralamada kullandığı zamandan daha fazlasını sadeleştirme aşamasında kazandırmıştır.

Fakat “max1024”, “br1”, “bcc” gibi birkaç dosyada sonuç kalitesini etkilemeden çalışma süresini uzatmıştır. İzole mintermlerin tespiti algoritması YSKA algoritmasında 12 benchmarkın sadeleştirme süresini uzatırken, 38 benchmarkın sadeleştirme süresini kısaltmıştır. Benzer şekilde izole mintermlerin tespiti algoritması KSKA algoritmasında 11 benchmarkın sadeleştirme süresini uzatırken, 39 benchmarkın sadeleştirme süresini kısaltmıştır. Sadeleştirmeye izole mintermlerden başlanılması, daha doğru sonuca ulaşılmasını sağlamakla birlikte bazı istisna dosyalarda hesaplama zamanını arttırmaktadır.

6.6.3. Paralel programlamanın sadeleştirmeye etkisi

Bir algoritmanın farklı işlemcilerde aynı anda çalışmasını sağlamanın CPU işlem zamanı açısından bir maliyeti vardır. Fakat bir algoritmanın birden fazla işlemci kullanması çalışma zamanını kısaltmaktadır. Paralel programlamada, algoritmanın paralel programalamaya ne kadar uygun olduğu, verinin yâda program parçacıklarının eş zamanlı çalışıp çalışmayacağı önemlidir. Algoritmada kullanılan kod bloklarının birbirine bağlı olması, çalışma zamanında kodlar arasında bilgi göndermek paralel programlama kullanmayı zorlaştırmaktadır.

YSKA ve KSKA algoritmaları benchmarkların sadeleştirilmesinde ON ve OFF mintermleri kullanmaktadır. Her bulunan AI sonucunda ON mintermler kümesi tekrar kontrol edilip kapsanan mintermler çıkarılarak liste güncellenmektedir. Bu sebeple ON mintermlerini birbirinden ayrı değerlendirmek, ON mintermlerini topluca paralel programlama ile işlemek mümkün değildir. Fakat algoritma içerisinde kullanılan küçük kod parçacıkları paralel hale getirilebilmektedir. Performans testi sonuçları paralel programlamanın bazı benchmarklarda performans artışı sağladığını, bazı benchmarklarda ise performansı düşürdüğünü göstermiştir.

50 MCNC benchmarkı üzerinde yapılan test sonuçlarına göre, paralel programlama YSKA algoritmasının çalışma süresini 322,07 milisaniyeden 248,39 milisaniyeye düşürerek, sadeleştirme süresini %22,88 kısaltmıştır. KSKA algoritması üzerinde yapılan testlerde ise paralel programlama çalışma süresini 2230,15 milisaniyeden 2256,54 milisaniyeye çıkararak, sadeleştirme süresini %1,18 arttırmıştır. Paralel programlama YSKA çalışma zamanını “exps”, “apex4” ve “max1024” benchmarklarında %70’in üzerinde düşürürken, “bcc”, “check” ve “bcb” dosyalarında %70’den fazla arttırmıştır. Paralel programlamanın YSKA’ya etkisini gösteren

performans testi sonuçları Çizelge 6.4'de verilmiştir. Paralel programlama KSKA çalışma zamanını "dist", "max512" ve "apex4" benchmarklarında %65'in üzerinde düşürürken, "bcc" ve "bcb" dosyalarında %80'den fazla arttırmıştır. Sonuçlar incelendiğinde paralel programlanmış YSKA ve KSKA algoritmalarının, giriş değişkeni sayısının fazla, ON mintermi sayısının az olduğu dosyalarda verimli çalışmadığı sonucuna varılabilir. Paralel programlamanın KSKA'ya etkisini gösteren performans testi sonuçları Çizelge 6.5'de verilmiştir.

Performans testi sonuçlarına göre, paralel programlama izole sıralı YSKA algoritmasının çalışma süresini 281,22 milisaniyeden 142,39 milisaniyeye düşürerek, sadeleştirme süresini %49,37 kısaltmıştır. İzole sıralı KSKA algoritması üzerinde yapılan testlerde ise paralel programlama çalışma süresini 2084,93 milisaniyeden 2071,40 milisaniyeye düşürerek, sadeleştirme süresini %0,65 kısaltmıştır. Paralel programlama izole sıralı YSKA çalışma zamanını "f51m", "apex4" ve "e" benchmarklarında %70'in üzerinde düşürürken, "bcc" ve "bcb" dosyalarında %30'dan fazla arttırmıştır. Paralel programlama izole sıralı KSKA çalışma zamanını "m", "m5" ve "max512" benchmarklarında %72'nin üzerinde düşürürken, "bcb", "max4" ve "den" dosyalarında %24'den fazla arttırmıştır. Paralel programlamanın izole sıralı KSKA'ya etkisini gösteren performans testi sonuçları Çizelge 6.6'da verilmiştir.

İzole mintermlerin tespiti algoritmasında bütün ON mintermleri OFF mintermleri ile karşılaştırıldığı için birbirini etkileyen ya da başka bir işlemin sonucuna bağlı hareket eden bir durum söz konusu değildir. Bu sebeple izole mintermlerin tespiti algoritması paralel programlamaya çok uygundur. Performans testi sonuçları incelendiğinde, YSKA algoritmasının paralel programlamaya uyarlanması çalışma zamanını %22,88 azaltırken bu oran izole sıralı YSKA için %49,37 olarak gerçekleşmektedir. Benzer şekilde KSKA algoritmasının paralel programlamaya uyarlanması çalışma zamanını %1,18 arttırırken, izole sıralı KSKA algoritmasında bu oran %0,65 olmaktadır. Algoritmaların paralel programlama ile kodlanması izole mintermlerin tespiti ve YSKA algoritmalarında olumlu sonuçlar (%49,37, %22,88) verirken, KSKA algoritmasında belirgin bir farklılık (%1,18, %0,65) oluşturmamaktadır. Paralel programlamanın algoritmaların bulduğu AI sayısı üzerinde herhangi bir etkisi olmamıştır.

7. SONUÇLAR VE ÖNERİLER

7.1. Sonuçlar

Bilişim sektörünün hemen hemen bütün alanlarında kullanılan elektronik devrelerin daha küçük, daha sade ve daha hızlı olabilmesi için lojik fonksiyonların sadeleştirilmesi büyük önem taşımaktadır. Daha sade lojik fonksiyonlar sayesinde daha az lojik kapılar kullanmak mümkündür. Bu sayede daha az malzeme kullanarak, daha küçük boyutlarda ve daha hızlı çalışan entegre devreler yapmak mümkündür.

Bu tez çalışmasında bitsel operatörler kullanarak lojik fonksiyonların kapsanması için 3 algoritma geliştirilmiştir. Bunlar Yakın Sonuç Kapsama Algoritması (YSKA) ve Kesin Sonuç Kapsama Algoritması (KSKA) ve İzole Mintermlerin Tespiti Algoritmasıdır. Geliştirilen 3 algoritma, Visual Studio platformunda C# dili kullanılarak hem seri hemde paralel programlama ile kodlanmıştır. Geliştirilen algoritmaların performans ve sonuç kalitelerinin karşılaştırılması için dünya çapında kabul görmüş MCNC benchmarkları kullanılmıştır. Bu benchmarklar fonksiyon sadeleştirme algoritmalarının performansını ölçmek için özel olarak hazırlanmış dosyalardır. Dosyaların giriş değişkeni sayıları 4 ile 26 arasında değişmekte olup, bazı dosyalar 100'lerce ON mintermi içerirken, bazıları 100'lerce OFF mintermi içermektedir.

Tez çalışmasında sunulan yöntemlerden YSKA ve KSKA birbirleri ile karşılaştırılarak bulunan SOP sayısı ve çalışma süresi açısından analiz edilmiştir. Bununla birlikte geliştirilen İzole Mintermlerin Tespiti Algoritması hem YSKA hemde KSKA algoritmalarına uygulanarak sadeleştirme sonucunu ne kadar etkilediği ve ne kadar sürede hesapladığı incelenmiştir. İzole minterm sıralı versiyonları ile birlikte geliştirilen 4 sadeleştirme programının birebir paralel versiyonları geliştirilmiştir. Toplamda geliştirilen 8 program benchmarklar üzerinde denenerek, hangi programın paralel programlamaya daha uygun olduğu araştırılmıştır.

Araştırma sonuçlarına göre YSKA algoritmasının sadeleştirme oranı %82,86 iken, KSKA algoritmasında bu oran %0,34 artarak %83,20 olmuştur. YSKA ve KSKA algoritmaları 41 benchmarkta eşit sayıda AI sonucuna ulaşmış, 9 benchmarkta ise KSKA algoritması daha iyi sonuç bulmuştur. Fakat YSKA algoritmasının toplam hesaplama süresi 322,07 milisaniye iken, KSKA algoritmasında bu oran 6,92 kat artarak 2230,15 milisaniye olmuştur. Araştırma sonucunda giriş değişkeni ve ON mintermi sayısının fazla olmadığı benchmarklarda YSKA ve KSKA algoritmalarının aynı ya da çok yakın sadeleştirme yaptıkları tespit edilmiştir. Yüksek sayıda ON ve/veya OFF mintermi içeren

benchmarklarda KSKA yüksek çalışma süresine rağmen daha sade sonuçlara ulaşmıştır. Zamanın kısıtlı olduğu ya da çok fazla ON – OFF mintermi içermeyen dosyaların sadeleştirilmesinde YSKA tercih edilebilir. Büyük dosyaların sadeleştirilmesinde ve yeterli zaman varsa KSKA tercih edilebilir.

Bu tez çalışmasında izole mintermlerin tespitinin hem YSKA hemde KSKA algoritmalarının sonuç kalitelerini arttırdığı ve genel ortalamada çalışma zamanlarını düşürdüğü tespit edilmiştir. Genel olarak bakıldığında izole minterm sıralama algoritması sıralamada kullandığı zamandan daha fazlasını sadeleştirme aşamasında kazandırmıştır. İzole mintermlerin tespiti algoritması YSKA üzerinde %2,33 sonuç kalitesi artışı, %12,68 çalışma zamanı kazanımı sağlamıştır. İzole mintermlerin tespiti algoritması KSKA üzerinde %1,09 sonuç kalitesi artışı, %6,51 çalışma zamanı kazanımı sağlamıştır.

İzole mintermlerin tespiti algoritması YSKA algoritmasında 12 benchmarkın sadeleştirme süresini uzatırken, 38 benchmarkın sadeleştirme süresini kısaltmıştır. Benzer şekilde izole mintermlerin tespiti algoritması KSKA algoritmasında 11 benchmarkın sadeleştirme süresini uzatırken, 39 benchmarkın sadeleştirme süresini kısaltmıştır.

İzole mintermleri tespit edip sıralamak için geliştirilen algoritma sıralama aşamasında işlem zamanı kullanırken, YSKA ve KSKA algoritmalarının işini kolaylaştırarak sadeleştirme aşamasında zaman kazandırmaktadır. Fakat sıralama ihtiyacı olmayan, kendisinden sıralı olan dosyalar üzerinde tekrar sıralama yapılırsa bir kazanç sağlamayıp, çalışma zamanını arttırmaktadır.

Performans testi sonuçları paralel programlamanın bazı benchmarklarda performans artışı sağladığını, bazı benchmarklarda ise performansı düşürdüğünü göstermiştir.

Paralel programlama YSKA algoritmasının çalışma süresini %22,88 kısaltırken, KSKA algoritmasının çalışma süresini %1,18 arttırmıştır. Benzer şekilde paralel programlama izole sıralı YSKA algoritmasının çalışma süresini %49,37 kısaltırken, izole sıralı KSKA algoritmasının çalışma süresini sadece %0,65 kısaltmıştır. Sonuçlar incelendiğinde paralel programlanmış YSKA ve KSKA algoritmalarının, giriş değişkeni sayısının fazla, ON mintermi sayısının az olduğu dosyalarda verimli çalışmadığı sonucuna varılabilir.

İzole mintermlerin tespiti algoritmasında bütün ON mintermleri OFF mintermleri ile karşılaştırıldığı için birbirini etkileyen ya da başka bir işlemin sonucuna bağlı hareket eden bir durum söz konusu değildir. Bu sebeple izole mintermlerin tespiti algoritması

paralel programlamaya çok uygundur. İzole sıralı algoritmaların paralel programlama ile kodlanması algoritma performansını arttırmıştır. Paralel programlama izole sıralı YSKA için çalışma zamanını %49,37 kısaltırken, izole sıralı KSKA algoritmasında bu oran %0,65 olmaktadır. Algoritmaların paralel programlama ile kodlanması izole mintermlerin tespiti ve YSKA algoritmalarında önemli iyileştirmeler (%49,37, %22,88) sunarken, KSKA algoritmasında belirgin bir farklılık (%1,18, %0,65) oluşturmamaktadır. Paralel programlamanın algoritmaların sonuç kalitesi, yani bulduğu AI sayısı üzerinde herhangi bir etkisi olmamıştır.

Sonuç olarak izole mintermlerin tespiti lojik fonksiyonların sadeleştirilmesinde hem sonuç kalitesi, hemde çalışma zamanı açısından performansı önemli ölçüde arttırmıştır. Algoritmaların paralel programlamaya uyarlanması, izole mintermlerin tespiti ve YSKA algoritmalarında performans artışı sağlarken, KSKA algoritmasında belirgin bir fark oluşturmamıştır.

7.2. Öneriler

Bu tez çalışmasında geliştirilen algoritmalar lojik fonksiyonların sadeleştirilmesi üzerine hazırlanmış ve klasik mantık devrelerinde kullanılmaktadır. Bu algoritmalar geliştirilerek quantum lojik devrelerinin tasarımı ve sadeleştirilmesi için kullanılabilir. Ayrıca çıkış sayısı arttırılarak çok değerli çıkış devrelerinde kullanılabilir.

Araştırma kapsamında geliştirilen fonksiyon kapsama algoritmaları için Visual Studio platformu kullanılmıştır. Uygulamada kullanılan arayüz Windows Forms uygulaması olup sadece Windows tabanlı bilgisayarlarda çalışmaktadır. Farklı kullanıcı arayüzleri tasarlanarak programın Linux ve MacOs işletim sistemlerinde de kullanılması sağlanabilir. Böylece platform bağımsız bir uygulama olup her bilgisayarda fonksiyon sadeleştirilmesi yapılabilir.

İnternetin çok hızlı gelişimi sayesinde depolama ve hesaplama dahil hemen hemen bütün bilgisayar işlemleri bulut üzerinde yapılmaktadır. Geliştirilen uygulama ASP server'da çalışacak şekilde değiştirilerek buluta yüklenebilir. Bu sayede cep telefonu, tablet gibi cihazlarda dâhil olmak üzere internet erişimi olan herhangi bir cihaz üzerinden fonksiyon sadeleştirmesi yapmak mümkün olacaktır.

Lojik fonksiyonların sadeleştirilmesi bir indirgeme problemi olduğu için minimizasyon gerektiren çok farklı alanlarda kullanılabilir. Geliştirilen sadeleştirme programının kullanıcı arayüzü programın farklı alanlarda kullanılmasına imkan

tanımaktadır. Günümüzde özellik indirgeme, resim sıkıştırma, akıllı evler ve binalarda kural seti indirgeme, kriptoloji, routing tabloları gibi çok farklı alanlarda fonksiyon sadeleştirme kullanılmaktadır. Geliştirilen algoritmalar bu alanlarda kullanılabilir.

Bu tez çalışmasında sunulan algoritmalarda mintermlerin sıralanması ve AI'ların seçimi için sezgisel, metasezgisel algoritmalar geliştirilebilir, yapay zeka veya bulanık mantık uygulamaları tercih edilebilir.

KAYNAKLAR

Abd-El-Barr, M. I. ve Khan, E. A., 2014, Improved direct cover heuristic algorithms for synthesis of multiple-valued logic functions, *International Journal of Electronics*, 101 (2), 271-286.

Abd-El-Barr, M., Qureshi, K. ve Sarif, B., 2020, Swarm intelligence versus direct cover algorithms in synthesis of Multi-Valued Logic functions, *Applied Computing and Informatics*, DOI: 10.1016/j.aci.2020.03.002.

Agarwal, A. ve Lang, J., 2009, Foundations of Analog and Digital Electronic Circuits, *New York; Boulder*, Elsevier NetLibrary, Incorporated [distributor], p.

Akers, 1978, Binary Decision Diagrams, *IEEE Trans. Comput. IEEE Transactions on Computers*, 27 (6).

Aksoy, L. ve Altun, M., 2019, A Satisfiability-Based Approximate Algorithm for Logic Synthesis Using Switching Lattices, Design, Automation & Test in Europe Conference & Exhibition (DATE), 1637-1642.

Allahverdi, N. M., Kahramanli, S. S. ve Erciyes, K., 2000, A fault tolerant routing algorithm based on cube algebra for hypercube systems, *Journal of Systems Architecture Journal of Systems Architecture*, 46 (2), 201-205.

Altun, M. ve Riedel, M. D., 2012, Logic Synthesis for Switching Lattices, *IEEE Trans. Comput. IEEE Transactions on Computers*, 61 (11), 1588-1600.

Amarú, L., Soeken, M., Haaswijk, W., Testa, E., Vuillod, P., Luo, J., Gaillardon, P. E. ve Micheli, G. D., 2017a, Multi-level logic benchmarks: An exactness study, *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 157-162.

Amarú, L., Vuillod, P., Luo, J. ve Olson, J., 2017b, Logic optimization and synthesis: Trends and directions in industry, *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 1303-1305.

Arslan, N. ve Sertbaş, A., 2002, An educational computer tool for simplification of boolean function's via petrick's method, *Istanbul University Journal of Electrical & Electronics Engineering*, 2 (2), 555-561.

Bachmann, P., 1894, Die Analytische Zahlentheorie.

Basciftci, F. ve Akar, H., 2014, Finding isolated minterms in simplification of logic functions, *International Conference on challenges in IT,Engineering and Technology*, Phuket.

Basciftci, F. ve Akar, H., 2020, Smart minterm ordering and accumulation approach for insignificant function minimization, *Ain Shams Engineering Journal*, DOI: 10.1016/j.asej.2020.04.003.

Basciftci, F. ve Avuçlu, E., 2018, An expert system design to diagnose cancer by using a new method reduced rule base, *Computer Methods and Programs in Biomedicine*, 157, 113-120.

Basciftci, F. ve Hatay, O. F., 2011, Reduced-rule based expert system by the simplification of logic functions for the diagnosis of diabetes, *Comput Biol Med*, 41 (6), 350-356.

Basciftci, F., Kahramanli, S. ve Selek, M., 2012, A reduced offset based method for fast computation of the prime implicants covering a given cube, *Int. J. Innov. Comput. Inf. Control International Journal of Innovative Computing, Information and Control*, 8 (6), 4333-4345.

Başçiftçi, F., 2006, Anahtarlama fonksiyonları için yerel basitleştirme algoritmaları, Doktora Tezi, *Selçuk Üniversitesi*, Konya.

Başçiftçi, F. ve Kahramanli, Ş., 2011, Fast computation of the prime implicants by exact direct-cover algorithm based on the new partial ordering operation rule, *Advances in Engineering Software*, 42 (6), 316-321.

Başçiftçi, F. a. K., Ş., 2010a, Anahtarlama fonksiyonları için yeni yakın minimum sadeleştirme algoritması, *Gazi Üniversitesi Mühendislik Mimarlık Fakültesi Dergisi*, 25 (1), 83-91.

Başçiftçi, F. v. K., Ş., 2010b, Fast computation of determination of the prime implicants by a novel near minimum minimization method, *Turkish Journal of Electrical Engineering & Computer Science*, 18 (6), 1041-1051.

Bernasconi, A., Cavalle-Garrido, T. ve Redington, A., 2009a, Spontaneous intraoperative ventricular haematoma in a neonate, *BMJ case reports*, 2009, bcr2006098475.

Bernasconi, A., Ciriani, V., Trucco, G., Villa, T., Design, A., Test in Europe, C. ve Exhibition, 2009b, On decomposing Boolean functions via extended cofactoring, 1464-1469.

Bernasconi, A., Ciriani, V., Fišer, P. ve Trucco, G., 2012, Weighted dont cares. 10th Int Workshop on Boolean Problems IWSBP. Freiberg Germany**:** 123-130.

Bernasconi, A., ve ark., 2018, Enhancing logic synthesis of switching lattices by generalized Shannon decomposition methods, *Microprocessors and Microsystems,* 56, 193-203.

Besslich, P. W., 1986, Heuristic Minimization of MVL Functions: A Direct Cover Approach, *IEEE Transactions on Computers*, C-35 (2), 134-144.

Bindal, A., 2017, Electronics for Embedded Systems.

Boole, G., 1998, The Mathematical Analysis of Logic, St Augustine Pr Inc, p.

Borowik, G., Łuba, T. ve Zydek, D., 2012, Features Reduction Using Logic Minimization Techniques, 58 (1), 71.

Boyar, J., Matthews, P. ve Peralta, R., 2013, Logic Minimization Techniques with Applications to Cryptology, *Journal of cryptology : the journal of the International Association for Cryptologic Research.*, 26 (2), 280-312.

Boyd, M. J., 2005, Complexity analysis of a massive parallel boolean satisfiability implication circuit.

Braun, W. ve Menth, M., 2014, Wildcard Compression of Inter-Domain Routing Tables for OpenFlow-Based Software-Defined Networking, *2014 Third European Workshop on Software Defined Networks*, 25-30.

Brown, D. W., 1981, A State-Machine Synthesizer—SMS, *Conference on Design Automation*, 6 (1), 301-305.

Bryant, R. E., 1986, Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Trans. Comput. IEEE Transactions on Computers*, 35 (8), 677-691.

Bryant, R. E., 1992, Symbolic Boolean manipulation with ordered binary-decision diagrams, *ACM Comput. Surv.*, 24 (3), 293-318.

Cepek, O., Kucera, P. ve Savický, P., 2012, Boolean functions with a simple certificate for CNF complexity, *DAM Discrete Applied Mathematics*, 160 (4-5), 365-382.

Cepek, O., Kucera, P. ve Kurík, S., 2013, Boolean functions with long prime implicants, *IPL Information Processing Letters*, 113 (19-21), 698-703.

Chai, L., 2000, ESOP circuit minimization based on the function on-set.

Chikalov, I., Hussain, S. ve Moshkov, M., 2013, Totally Optimal Decision Trees for Monotone Boolean Functions with at Most Five Variables.

Cobb, J. L., 2007, A robust window-based multi-node minimization technique using Boolean relations, *Texas A&M University*, Texas.

Conceição, C. ve Reis, R., 2019, Transistor Count Reduction by Gate Merging, *IEEE Transactions On Circuits And Systems,* 66(6), 2175-2187.

Congguang, Y. ve Ciesielski, M., 2002, BDS: a BDD-based logic optimization system, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21 (7), 866-876.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. ve Stein, C., 2009, Introduction to Algorithms, Third Edition, The MIT Press, p.

Coudert, O., 1994, Two-level logic minimization: an overview, *Integration, the VLSI Journal*, 17 (2), 97-140.

Crama, Y., Hammer, P. L. ve Cambridge University, P., 2011, Boolean functions : theory, algorithms, and applications, *Cambridge*, Cambridge University Press, p.

Creignou, N. ve Daudé, H., 2013, Sensitivity of Boolean formulas, *European Journal of Combinatorics*, 34 (5), 793-805.

Cruz-Cano, R., Lee, M.-L. T. ve Leung, M.-Y., 2012, Logic minimization and rule extraction for identification of functional sites in molecular sequences, *BioData mining*, 5 (1), 10.

Çebi, Ç., 2006, C Programlama Dersi - X, http://www.cagataycebi.com/programming/c_programming/c_programming_10.html.

Çini, U., 2010, Alternative Arithmetic Structures Using Redundant Numbers and Multi-Valued Circuit Techniques, *Boğaziçi University*, İstanbul.

Çölkesen, R., 2004, Veri yapilari ve algoritmalar, *Istanbul*, Papatya Yayincilik, p.

Denning, P. J. ve Lewis, T. G., 2017, Exponential laws of computing growth, *Commun ACM Communications of the ACM*, 60 (1), 54-65.

De Souza, L. ve ark., 2020, A benchmark suite for designing combinational logic circuits via metaheuristics, *Applied Soft Computing*, 91, 106246.

Dick, R. P., 2014, Optimal Two-Level Boolean Minimization, 1-6.

Drucker, A. D., 2012, The complexity of joint computation, *Massachusetts Institute of Technology*.

Du, Z., Palem, K., Lingamneni, A., Temam, O., Chen, Y. ve Wu, C., 2014, Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators, *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 201-206.

Du, Z., Lingamneni, A., Chen, Y., Palem, K. V., Temam, O. ve Wu, C., 2015, Leveraging the Error Resilience of Neural Networks for Designing Highly Energy Efficient Accelerators, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34 (8), 1223-1235.

Dueck, G. W. ve Miller, D. M., 1987, A direct cover MVL minimization using the truncated sum, *IEEE International Symposium on Multiple-Valued Logic*, 221-226.

Dueck, G. W., 1988, Algorithms for the minimization of binary and multiple-valued logic functions, *University of Manitoba*.

Duşa, A. ve Thiem, A., 2015, Enhancing the Minimization of Boolean and Multivalue Output Functions With eQMC, *The Journal of Mathematical Sociology*, 39 (2), 92-108.

El-Bakry, H. M. ve Mastorakis, N., 2009, A fast computerized method for automatic simplification of boolean functions. Proceedings of the 9th WSEAS International Conference on Systems Theory and Scientific Computation. Moscow, Russia, World Scientific and Engineering Academy and Society (WSEAS)**:** 99-107.

El-Bakry, H. M. ve Atwan, A., 2010, Simplification and Implementation of Boolean Functions, *international Journal of Universal Computer Sciences*, 1 (1), 41-50.

El-Bakry, H. M., Atwan, A. ve Mastorakis, N., 2010, A new technique for realization of Boolean functions. Proceedings of the 9th WSEAS international conference on Artificial intelligence, knowledge engineering and data bases. UK, World Scientific and Engineering Academy and Society (WSEAS)**:** 260-270.

Fallah, F., 1996, A new algorithm for factorization of logic expressions.

Färm, P., 2004, Advanced algorithms for logic synthesis.

Feldman, V., 2009, Hardness of approximate two-level logic minimization and PAC learning with membership queries, *YJCSS Journal of Computer and System Sciences*, 75 (1), 13-26.

Fiser, P., Hlavicka, J. ve Kubatova, H., 2003, FC-Min: a fast multi-output Boolean minimizer, *Euromicro Symposium on Digital System, Design*, 451-454.

Fiser, P. ve Kubatova, H., 2006, Flexible Two-Level Boolean Minimizer BOOM-II and Its Applications, *9th EUROMICRO Conference on Digital System Design (DSD'06)*, 369-376.

Fiser, P., Rucky, P. ve Vanova, I., 2008, Fast Boolean Minimizer for Completely Specified Functions, *Ieee International Workshop on Design Diagnostics of Electronic, Circuits Systems*, 1-6.

Fiser, P. ve Toman, D., 2008, BoolTool: A Tool for Manipulation of Boolean Functions, *8th Int. Workshop on Boolean Problems*, Freiberg 109-114.

Fiser, P. ve Toman, D., 2009, A Fast SOP Minimizer for Logic Funcions Described by Many Product Terms, *12th Euromicro Conference on Digital Systems Design*, Patras 757-764.

Fišer, P., 2002a, Minimization of Boolean functions, Yüksek Lisans University, , *Czech Technical University*, Prague.

Fišer, P. ve Kubatova, H., 2004a, Two-level Boolean minimizer BOOM-II, *6th Int. Workshop on Boolean Problems*, Freiberg, 221-228.

Fišer, P. ve Kubatova, H., 2004b, Boolean minimizer fc-min: coverage finding process, *30th Euromicro Symposium On Digital Systems Design*, Rennes, 152-159.

Fišer, P. v. H., J., 2001a, BOOM-minimizer, *Prague Czech*.

Fišer, P. v. H., J., 2001b, On the use of mutations in Boolean minimization. Euromicro Symposium On Digital Systems Design. Warsaw**:** 300-307.

Fišer, P. v. H., J., 2002b, A flexible minimization and partitioning method. Workshop on Boolean Problems Freiberg, Germany. 5th**:** 83-90.

Fišer, P. v. H., J., 2003, Boom — A heuristic Boolean minimizer, *Computing and Informatics*, 22, 19-51.

Flynn, M. J., 1972, Some Computer Organizations and Their Effectiveness, *IEEE Trans. Comput. IEEE Transactions on Computers*, 21 (9), 948-960.

Fränzle, M., Herde, C., Teige, T., Ratschan, S. ve Schubert, T., 2007, Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure, *Journal on Satisfiability, Boolean Modeling and Computation*, 1, 209-236.

Gavrilov, M. A., 1959, Minimization of boolean functions that characterize switching circuits, *Avtomat. i Telemekh.*, 20 (9), 1217–1238.

Ghasemzadeh, M., 2005, A new algorithm for the quantified satisfiability problem, based on zero-suppressed binary decision diagrams and memoization.

Goyal, P., 2014, Comparative Study of C, Java, C# and Jython, MS Thesis, Master Tezi, Unıversıty Of North Florıda.

Gröpl, C., 1999, Binary decision diagrams for random Boolean functions, *Humboldt-Universitat*, Berlin.

Hacibeyoglu, M., Basçiftçi, F. ve Kahramanlı, Ş., 2011, A logic method for efficient reduction of the space complexity of the attribute reduction problem, *Turkish Journal of Electrical Engineering and Computer Science*, 19 (4), 643-656.

Hill, M.D. ve Marty, M.R., 2008, Amdahl's Law in the Multicore Era, *Computer* 41(7), 33–38.

Hlavicka, J. ve Fiser, P., 2001, BOOM-a heuristic Boolean minimizer, *Ieee Acm International Conference on Computer Aided Design. ICCAD* . 439-442.

Hlavicka, J. ve Fiser, P., 2002, Minimization and partitioning method reducing input sets, *First Ieee International Workshop on Electronic Design, Test Applications*, 434-436.

Hlavička, J. ve Fišer, P., 2000, Algorithm for minimization of partial Boolean functions. Design and Diagnostic of Electronic Circuits and Systems Workshop. Smolenice-Slovakia**:** 130-133.

Hlavička, J. ve Fišer, P., 2001, A Heuristic method of two-level logic synthesis, *The 5th World Multiconference on Systemics, Cybernetics and Informatics* Orlando, 283-288.

Hong, S. J., Cain, R. G. ve Ostapko, D. L., 1974, MINI: a heuristic approach for logic minimization, *IBM Corp.*, 18 (5), 443-458.

Ivutin, A. N., Troshina, A. G., Yesikov, D. O. ve Vasiliev, S. N., 2017, Estimation of parallel algorithms efficiency based on modified Petri nets, *2017 6th Mediterranean Conference on Embedded Computing (MECO)*, 1-4.

Jenkins, T., 2005, A brief history of...semiconductors, *Physics education.*, 40 (5), 430-439.

Jiang, J.-H. ve Devadas, S., 2009, Logic synthesis in a nutshell, In: Electronic Design Automation, Eds: Elsevier, p. 299-404.

Kahramanli, S., Hacibeyoglu, M. ve Arslan, A., 2011, A Boolean function approach to feature selection in consistent decision information systems, *Expert Systems with Applications*, 38 (7), 8229-8239.

Kahramanlı, Ş., Güneş, S., Şahan, S. ve Başçiftçi, F., 2006, A new method based on cube algebra for the simplification of Lojik functions, *The Arabian Journal For Science And Engineering*, 32 (1B), 101-114.

Kahramanlı, Ş. v. B. F., 2003, Boolean functions simplification algorithm of On complexity *Mathematical Computational Applications*, 8 (3), 271-278.

Kang, S. ve vanCleemput, W. M., 1981, Automatic PLA Synthesis from a DDL-P Description, *18th Design Automation Conference*, 391-397.

Karnaugh, M., 1953, The map method for synthesis of combinational logic circuits, *American Institute of Electrical Engineers Par...* 72 (5), 593-599.

Knuth, D. E., 2009, The art of computer programming.

LaMeres, B. J., 2017, Introduction to Logic Circuits & Logic Design with Verilog.

Landau, E., 1953, Handbuch der Lehre von der Verteilung der Primzahlen . [2 vols. in 1, *New York*, Chelsea, p.

Lee, C. Y., 1959, Representation of switching circuits by binary-decision programs, , *The Bell System Technical Journal*, 38 (4), 985-999.

Lee, S., Kim, S. ve Kang, S., 2019, Ternary Logic Synthesis with Modified Quine-McCluskey Algorithm, *49th International Symposium on Multiple-Valued Logic (ISMVL)*, 158-163.

Lemberski, I. ve Fiser, P., 2009a, Multi-Level Implementation of Asynchronous Logic Using Two-Level Nodes, *IFAC Proceedings Volumes IFAC Proceedings Volumes*, 42 (21), 178-183.

Lemberski, I. ve Fiser, P., 2009b, Asynchronous two-level logic of reduced cost, *2009 12th International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, 68-73.

Lemberski, I., Fiser, P. ve Suleimanov, R., 2014, Asynchronous sum-of-products logic minimization and orthogonalization, *International Journal of Circuit Theory and Applications*, 42 (6), 562-571.

Li, P.-y. P., 1989, A parallel execution model for logic programming, *California Institute of Technology*.

Li, X., Zhou, Q., Qian, H., Yu, Y. ve Tang, S., 2013, Balanced 2p-variable rotation symmetric Boolean functions with optimal algebraic immunity, good nonlinearity,

and good algebraic degree, *J. Math. Anal. Appl. Journal of Mathematical Analysis and Applications*, 403 (1), 63-71.

Limketkai, B., 2009, NAND NOR Cubes. Retrieved 2018, from https://courses.cs.washington.edu/courses/cse370/09sp/lectures/05-NAND-NOR-cubes.pdf.

Lin, C. C., Wang, C. Y., Chen, Y. C. ve Huang, C. Y., 2014, Rewiring for threshold logic circuit minimization, *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 1-6.

Lu, P., 1990, Boolean techniques in discrete optimization and expert systems, *National Library of Canada*, Ottawa.

Lukac, M., Kameyama, M., Perkowski, M. ve Kerntopf, P., 2012, Minimization of Quantum Circuits using Quantum Operator Forms, p.

Macii, E., Calimera, A., Macii, A. ve Poncino, M., 2017, Logic Synthesis of CMOS Circuits and Beyond, In: Nanoelectronics, Eds, p.

Mailhot, F., 1994, Technology Mapping for VLSI Circuits Exploiting Boolean Properties and Operations, *Stanford University*.

Markham Brown, F., 2010, McColl and minimization, *History and philosophy of logic / ed. by I. Grattan-Guinness.*, 31.

Martins, M. G. A., Ribas, R. P. ve Reis, A. I., 2012, Functional composition: A new paradigm for performing logic synthesis, *Thirteenth International Symposium on Quality Electronic Design (ISQED)*, 236-242.

McCluskey, E. J., 1956, Minimization of Boolean functions, *The Bell System Technical Journal*, 35 (5), 1417-1444.

Microsoft, 2015, C# Diline ve.NET Framework'e Giriş, https://docs.microsoft.com/tr-tr/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework, Erişim Tarihi: 30.12.2019.

Miller, J. F., Bradbeer, V. G. ve Thomson, P., 1996, Experiences of using evolutionary techniques in Logic minimisation, *1st Online Workshop On Soft Computing*, Nagoya-Japan.

Mizuki, T., Mikami, D. ve Sone, H., 2014, Minimizing ESCT forms for two-variable multiple-valued input binary output functions, *Discrete Applied Mathematics*, 169 (6), 186-194.

Moore, G. E., 1965, Cramming more components onto integrated circuits, *Electronics*, 38 (8).

Morgan, A. D., 1847, Formal logic - or the calculus of inference, necessary and probable, *London*, Taylor and Walton, p.

Morreale, E., 1970, Computational Complexity of Partitioned List Algorithms, *IEEE Transactions on Computers*, C-19 (5), 421-428.

Movsisyan, Y. M. ve Aslanyan, V. A., 2014, A functional completeness theorem for De Morgan functions, *Elsevier Science Publishers B. V.*, 162 (162), 1-16.

Nandan, V. ve Rao, R., 2020, Minimization of digital logic gates and ultra-low power AES encryption core in 180CMOS technology, *Microprocessors and Microsystems,* 74, 103000.

Nazemi, M., Pasandi, G. ve Pedram, M., 2019, Energy-Efficient, Low-Latency Realization of Neural Networks through Boolean Logic Minimization, *24th Asia and South Pacific Design Automation Conference,* 274–279.

Necula, N. N., 1968, An Algorithm for the Automatic Approximate Minimization of Boolean Functions, *IEEE Transactions on Computers*, C-17 (8), 770-782.

Nosrati, M. ve Hariri, M., 2011, An Algorithm for Minimizing of Boolean Functions Based on Graph DS, *World Applied Programming*, 1 (3), 209-214.

Nowick, S. M. ve Dill, D. L., 1995, Exact Two-Level Minimization of Hazard-Free Logic with Multiple-Input Changes, *IEEE transactions on computer-aided design of integrated circuits and systems : a publication of the IEEE Circuits and Systems Society.*, 14 (8), 986.

Oliveira, A. L., Carloni, L. P., Villa, T. ve Sangiovanni-Vincentelli, A. L., 1998, Exact minimization of binary decision diagrams using implicit techniques, *IEEE Trans. Comput. IEEE Transactions on Computers*, 47 (11), 1282-1296.

Oral, S. O., 2005, Çoklu değerli mantık fonksiyonlarının küçültülmesinde ayrıştırma kullanımı ve uyumsuz çoklu değerli mantık fonksiyonları, *Ankara University*, Ankara.

Özcan, U. ve Dervişoğlu, A., 2003, Ikili Karar Diyagramlari Yardimiyla Boole Fonksiyonlarinin Asal Bilesenlerinin Belirlenmesi. 10th National Conference on Electrical, Electronic and Computer Engineering. İstanbul. SRC - BaiduScholar**:** 295-298.

Pang, Y., 2012, Fast Algorithm to Generate Arithmetic Transform for Incompletely Specified Boolean Functions Using Block Matrix, *Procedia Engineering Procedia Engineering*, 29, 3722-3726.

Papakonstantinou, G., 2017, Exclusive or Sum of Complex Terms expressions minimization, *VLSI Integration, the VLSI Journal*, 56, 44-52.

Perinkulam, A. S., 2007, Logic Simulation Using Graphics Processors, *University of Massachusetts Amherst*.

Piscitello, A., Nacci, A. A., Rana, V., Santambrogio, M. D. ve Sciuto, D., 2016, Ruleset Minimization in Multi-tenant Smart Buildings. Ieee Intl Conference on Computational Science Engineering**:** 72-79.

Pomper, G. M. ve Armstrong, J. R., 1981, Representation of multivalued functions using the direct cover method, *IEEE Transactions on Computer*, 30 (9), 1981.

Quine, W. V., 1952, The Problem of Simplifying Truth Functions, *The American Mathematical Monthly*, 59 (8), 521-531.

Quine, W. V., 1955, A Way to Simplify Truth Functions, *The American Mathematical Monthly*, 62 (9), 627-631.

Rawat, V., Singh, R., Pawar, M. ve Mishra, R., 2012, Lossless Gray Image Compression Using Logic Minimization, *Recent Research in Science and Technology*, 4 (1), 14-18.

Rhyne, V. T., Noe, P. S., Mckinney, M. H. ve Pooch, U. W., 1977, A New Technique for the Fast Minimization of Switching Functions, *IEEE Transactions on Computers*, C-26 (8), 757-764.

Roth, J. P. ve Karp, R. M., 1962, Minimization over Boolean graphs, *IBM Corp.*, 6 (2), 227-238.

Roy, S. ve Bhunia, C. T., 2014, Minimization algorithm for multiple input to two input variables, *Proceedings of The 2014 International Conference on Control, Instrumentation, Energy and Communication (CIEC)*, 555-557.

Roy, S. ve Bhunia, C. T., 2015, Simplification of Switching Functions Using Hex-Minterms, *International Journal of Applied Engineering Research*, 10 (24), 45619-45624.

Roy, S. ve Tilak, C., 2015, On Synthesis of Combinational Logic Circuits, *International Journal of Computer Applications*, 127 (1), 21-26.

Roy, S., 2017, An Efficient Technique For Switching Functions Simplification, *International Journal Of Advanced Engineering And Management*, 2 (1), 21.

Rudell, R., 1989, Logic synthesis for VLSI design, *University of California*, Berkeley.

Rudell, R. L., 1986, Multiple-valued logic minimization for PLA synthesis, *Berkeley*, Electronics Research Laboratory, College of Engineering, University of California, p.

Sampson, M., Kalathas, M., Voudouris, D. ve Papakonstantinou, G., 2012, Exact ESOP expressions for incompletely specified functions, *Integration -Amsterdam*, 45 (2), 197-204.

Sasao, T., 1999, Switching Theory for Logic Synthesis, Springer US, p.

Savran, İ., 2006, Mantıksal fonksiyonların sadeleştirilmesi, *Selçuk Üniversitesi* Konya.

Seiffertt, J., 2017, Digital Logic for Computing, Springer International Publishing, p.

Selek, M., Başçiftçi, F. ve Örücü, S., 2017, Designing Medical Expert System Based On Logical Reduced Rule for Basic MalariaDiagnosis from Malaria Signs and Symptoms, *World Journal of Engineering*, 14.

Shannon, C. E., 1938, A symbolic analysis of relay and switching circuits, *Electr. Eng. Electrical Engineering*, 57 (12), 713-723.

Strauss D., 2019, C# 7 in Focus. In: Exploring Advanced Features in C#, Apress, Berkeley, CA.

Strzemecki, T., 1992, Polynomial-time algorithms for generation of prime implicants, *Journal of Complexity*, 8 (1), 37-63.

Svoboda, A., 1967, Ordering of Implicants, *IEEE Transactions on Electronic Computers*, EC-16 (1), 100-105.

Tirumalai, P. P. ve Butler, J. T., 1988, Analysis of minimization algorithms for multiple-valued programmable logic arrays, *The Eighteenth International Symposium on Multiple-Valued Logic*, 226-236.

Tirumalai, P. P. ve Butler, J. T., 1991, Minimization algorithms for multiple-valued programmable logic arrays, *IEEE Transactions on Computers*, 40 (2), 167-177.

Toman, D. ve Fišer, P., 2010, A sop minimizer for Logic functions described by many product terms based on ternary trees, *9th Int Workshop on Boolean Problems IWSBP* Freiberg, 165-172.

Toman, D. ve Fišer, P., 2011, Using Ternary Trees in Lojik Synthesis, *Workshop ČVUT*, Prague, 1-15.

Tomaszewski, S. P., Celik, I. U. ve Antoniou, G. E., 2003, WWW-based Boolean function minimization, *Internatıonal Journal Of Applıed Mathematıcs And Computer Science*, 13, 577-584.

Uçar, O. ve Dervişoğlu, A., 2003, Genel örtü problemini çözen bir bilgisayar programının geliştirilmesi ve programın iki özel hale uygulanması, *Elektrik-Elektronik-Bilgisayar Mühendisliği 10. Ulusal Kongresi Ve Fuarı Bildirileri*, 275-278.

Umans, C., Villa, T. ve Sangiovanni-Vincentelli, A. L., 2006, Complexity of two-level logic minimization, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25 (7), 1230-1246.

Verma, S. ve Permar, K. D., 2004, A Novel Method for Minimization of Boolean Functions using Gray Code and development of a Parallel Algorithm, *6th International Workshop on Boolean Problems*, Freiberg.

Wang, H. ve Blanton, R. D. S., 2016, Ensemble reduction via logic minimization, *ACM Trans. Design Autom. Electron. Syst. ACM Transactions on Design Automation of Electronic Systems*, 21 (4).

Wang, L., 2000, Automated synthesis and optimization of multilevel Lojik circuits, Doktora, Edinburgh, *Napier University*.

Wang, Y., 1995, Data structures, minimization and complexity of Boolean functions, *Saskatchewan*, Ottawa.

Wu, L. ve Qiu, D., 2010, Automata theory based on complete residuated lattice-valued logic: Reduction and minimization, *Fuzzy Sets and Systems*, 161, 1635-1656.

Yang, C. ve Wang, Y. M., 1990, A neighborhood decoupling algorithm for truncated sum minimization, *Proceedings of the Twentieth International Symposium on Multiple-Valued Logic*, 153-160.

Yılmaz, B., 2007, Mantık fonksiyonlarının OFF kümesinin hızlı sadeleştirme algoritması, *Selçuk Üniversitesi*, Konya.

Zolfaghari, B. ve sheidaeian, H., 2011, A New Case for Image Compression Using Logic Function Minimization, *IJMA The International journal of Multimedia & Its Applications*, 3 (2), 45-62.

Zulehner, A. ve Wille, R., 2017, Improving Synthesis of Reversible Circuits: Exploiting Redundancies in Paths and Nodes of QMDDs, 232-247.

ÖZGEÇMİŞ

KİŞİSEL BİLGİLER

Adı Soyadı	**:**	Hakan AKAR
Uyruğu	**:**	T.C.
Doğum Yeri ve Tarihi	**:**	Antalya, 21/10/1981
Telefon	**:**	(542) 480-9030
Faks	**:**	
e-mail	**:**	me@hakanakar.com

EĞİTİM

Derece		**Adı, İlçe, İl**	**Bitirme Yılı**
Lise	:	**Anadolu Teknik Lisesi, Merkez, Antalya**	**1999**
Üniversite	:	**ODTU, Ankara**	**2004**
Yüksek Lisans	:	**GOP, Tokat**	**2009**
Doktora	:		

İŞ DENEYİMLERİ

Yıl	**Kurum**	**Görevi**

UZMANLIK ALANI

Lojik Fonksiyonlar, C#, Paralel Programlama, Veri Yapıları, Web Tasarım.

YABANCI DİLLER

İyi derecede İngilizce

BURSLAR

TÜBİTAK, Doktora Sırası Yurtdışı Araştırma Bursu, 2015, 1059B141500323.
Bilimsel Araştırma Projeleri, Gaziosmanpaşa Üniversitesi, 2008.

YAYINLAR

Basçiftçi, F., **Akar, H.**, 2020, Smart Minterm Ordering and Accumulatıon Approach For Insignificant Function Minimization, *Ain Shams Engineering Journal*, https://doi.org/10.1016/j.asej.2020.04.003.

Basçiftçi, F., **Akar, H.**, 2016, Parallelised Algorithm of Isolated Minterm Detection for Logic Function Simplification, *4th International Symposium on Innovative Technologies in Engineering and Science*, 3-5 November, Antalya, 1006-1014.

Kasalak, M., **Akar, H.**, Kasalak, F., 2015, Personal Internet Usage of Employees and Cyberslacking Trends In Antalya Hotel Businesses, *International Journal Of Business and Management Studies,* 4, 257-261.

Sezgin, E., **Akar, H.**, Dikilitaş, S., 2015, Semantik Web Bulutunun (Linked Data Cloud) Oluşumu ve Gelişim Durumu, *Akademik Bilişim'15*, 4-6 Şubat 2015, Eskişehir.

Basciftci, F., **Akar, H.**, 2014, Finding Isolated Minterms In Simplification Of Logic Functions, *International Conference on Challenges in IT, Engineering and Technology*, 17-18, July 2014, Thailand.

Akar, H., Başçiftçi, F., Uğuz, H., 2013, Paralel ve Sıralı Brute Force Algoritmasının Karşılaştırılması, *Akademik Bilişim'13*, 23-25 Ocak 2013, Antalya.

Ercan, U., **Akar, H.**, Koçer, A., 2013, Paralel Programlamada Kullanılan Temel Algoritmalar, *Akademik Bilişim'13*, 23-25 Ocak 2013, Antalya.

Akar, H., Ersozlu, Z. Arslan, M., 2012, Primary School Teachers' and Administrators' Opinions About Success of "E-School" Application in Turkey, *Mediterranean Journal of Social Sciences,* 3, 263-269.

Çetin, H., **Akar, H.**, 2012, Üniversite Öğrencilerinin Geleneksel Ve İnternet Gazeteciliğine İlişkin Görüşleri: Akdeniz Üniversitesi Örneği, *Dumlupınar Üniversitesi Sosyal Bilimler Dergisi*, 33, 263-275.

Akar, H., 2009, E-Okul Uygulamasının Başarısına Yönelik İlköğretim Okulu Öğretmen Ve İdarecilerinin Görüşleri, *Yayınlanmamış Yüksek Lisans Tezi*, Tokat, Gaziosmanpaşa Üniversitesi.