T.C.
MALTEPE ÜNİVERSİTESİ


FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

# SIK ALT ÇİZGE MADENCİLİĞİ ALGORİTMALARINA UYGULANABİLİR ALAN VE ZAMAN VERİMLİLİĞİ ARTTIRAN METOTLARIN GELİŞTİRİLMESİ

**MURAT OĞUZ**

Doktora Tezi

**Tez Danışmanı**
**Doç. Dr. Turgay Tugay Bilgin**

**İSTANBUL – 2016**

**T.C.**
**MALTEPE ÜNİVERSİTESİ**
**FEN BİLİMLERİ ENSTİTÜSÜ**
**BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI**

# SIK ALT ÇİZGE MADENCİLİĞİ ALGORİTMALARINA UYGULANABİLİR ALAN VE ZAMAN VERİMLİLİĞİ ARTTIRAN METOTLARIN GELİŞTİRİLMESİ

**DOKTORA TEZİ**

**MURAT OĞUZ**

**Tez Danışmanı**
**Doç. Dr. Turgay Tugay Bilgin**

**İSTANBUL – 2016**

T.C. Maltepe Üniversitesi

Fen Bilimleri Enstitüsü Müdürlüğü'ne.

27.07.2016 tarihinde tezinin savunmasını yapan Murat OĞUZ'a ait "Sık Alt Çizge Madenciliği Algoritmalarına Uygulanabilir Alan Ve Zaman Verimliliği Arttıran Metotların Geliştirilmesi" başlıklı çalışma, jürimiz tarafından Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı Bilgisayar Mühendisliği Doktora Programında Doktora Tezi olarak **Oy Birliği/~~Oy Çokluğu~~** ile kabul edilmiştir.

.....

Doç.Dr.Turgay Tugay BİLGİN

(Başkan)

.....

Yrd.Doç.Dr.Ali AKMAN

Jüri Üyesi

.....

Yrd.Doç.Dr.Önder TOMBUŞ

Jüri Üyesi

.....

Yrd.Doç.Dr.Buket DOĞAN

Jüri Üyesi

.....

Yrd.Doç.Dr.Ali NİZAM

Jüri Üyesi

# ÖZET

Doktora Tezi, Sık Alt Çizge Madenciliği Algoritmalarına Uygulanabilir Alan ve Zaman Verimliliği Arttıran Metotların Geliştirilmesi, Fen Bilimleri Enstitüsü, Bilgisayar Mühendisliği Anabilim Dalı.

Günümüzde sosyal ağlardaki ilişki analizlerinden kimyasal bileşiklerdeki ortak içerik tesbitine, uçak rotalarının verimli planlamasından insan gen dizilimleri üzerindeki benzerliklerin tesbitine dek birçok alanda sık alt çizge madenciliği teknikleri uygulanmaktadır. Bu tekniklerin uygulanması büyük veri setleri üzerinde yapılmakta ve işlemler uzun süreler almaktadır.

Bu doktora tezi sık alt çizge madenciliği alanındaki algoritmalara uygulanabilir, alan ve zaman verimliliklerini iyileştirmeye yönelik methodlar içermektedir. Methodların uygulanması ile algoritmaların yaptığı işlemler, daha kısa sürede ve daha az bellek alanı kullanılarak yapılabilmektedir. Geliştirmeler algoritmaların temel yapılarında bir değişiklik yapılmadan yalnızca bilgisayar bilimleri alanındaki tekniklerin uygulanması ile oluşturulmuştur.

Bu tez 5 bölümden oluşmaktadır. Birinci bölüm tezin amacını, kapsamını ve problemin tanımını içermektedir. İkinci bölümde bu tez çalışması kapsamındaki geliştirmelerin içerdiği düzeyde çizge madenciliği ile ilgili bilgiler ve tanımlar verilmiş, deneysel çalışmalarda kullanılan bilgiler ve araçlar anlatılmıştır. Tezin üçüncü ve dördüncü bölümlerinde sırasıyla alan verimliliği ve zaman verimliliği iyileştirmelerine yönelik yapılan çalışmalar ve deneysel sonuçlar verilmiştir. Son bölümde, tez çalışmasında elde edilen sonuçlar değerlendirilmiştir.

Bu tez 2016 yılında tamamlanmıştır ve 84 sayfadan oluşmaktadır.

**Anahtar Kelimeler:** Sık alt çizge madenciliği, alan verimliliği, zaman verimliliği, OpenMP, özyineleme, çok seviye paralelleştirme

# ABSTRACT

Doktorate Thesis, Development of Space and Time Efficiency Improvement Methods and Applying onto Frequent Subgraph Mining Algorithms, Maltepe University, Institute of Science, Computer Engineering Department.

Todays, like in social networking relationship analyze, chemical compound analyze, flight route optimization or human genome sequence research, frequent subgraph mining techniques are being used. These techniques are working on big data sets, so that processings take much times.

This doctorate thesis focused on developing two methods which can improve space and time efficiency in frequent subgraph mining techniques. By applying these methods, programs will work in less time and less memory usage than original ones. The developments have been done by applying computer engineering techniques without algorithmic structure change.

This thesis consists of 5 sections. First section explains the purpose of the thesis, its scope and the definition of the problem. Second section contains the necessary information and the definitions about graph mining, in addition the tools and the techniques which are used during the developing methods and datasets which are used in experimental sections are also explained. In third and fourth sections, the works on space and time efficiencies are mentioned. The last section, the results are discussed.

This thesis has been completed in 2016 and consists of 84 pages.

**Keywords:** Frequent subgraph mining, space efficiency, time efficiency, OpenMP, recursive, multilevel parallelisation

# TEŞEKKÜR

Doktora tez çalışmamın başlangıcından itibaren beni teşvik ederek her anlamda bir yol gösterici ve rehber olan danışmanım sayın Doç.Dr. Turgay Tugay Bilgin'e, tezim süresince yapıcı eleştiri ve önerileriyle tezime şekil vermemi sağlayan sayın Yrd. Doç.Dr. Ali Akman ve sayın Yrd. Doç.Dr. Önder Tombuş'a, bu uzun ve zorlu yolu tamamlamam için heran yanımda olan ve beni destekleyen değerli eşim Melek Oğuz'a ve aileme teşekkürlerimi sunarım.

# İÇİNDEKİLER

ÖZET ..... i
ABSTRACT ..... ii
TEŞEKKÜR ..... iii
İÇİNDEKİLER ..... iv
KISALTMALAR ..... vi
ŞEKİLLER ..... vii
ÇİZELGELER ..... ix
DENKLEMLER ..... x
1. GİRİŞ ..... 1
1.1 Amaç ..... 1
1.2 Kapsam ..... 2
1.3 Problemin Tanımı ..... 2
1.4 Literatür Araştırması ..... 4
1.5 Literatüre Katkı ..... 5
1.6 Tezin Yapısı ..... 6
2 ÖNBİLGİ VE KULLANILAN ARAÇLAR ..... 7
2.1 Veri Madenciliği ..... 7
2.2 Çizge Madenciliği ..... 8
2.3 Çizgelerin Kullanım Alanları ..... 11
2.4 Sık Alt Çizge Madenciliği ..... 12
2.4.1 Tanım ve SAÇM Problemleri ..... 12
2.4.2 SAÇM Tekniklerinin Özellikleri ..... 15
2.4.3 SAÇM Algoritmaları ..... 19
2.5 Veri Yapısı ..... 28
2.5.1 Hizalama Nedir ..... 28
2.5.2 Boşluk Doldurma ..... 29
2.5.3 Hizalama Hatası ..... 30
2.5.4 Hizalama Nasıl Yapılır ..... 30

2.5.5 Hizalama Optimizasyonu ..... 33
2.5.6 Yapı Paketleme Direktifleri ..... 34
2.6 Paralelleştirme ..... 35
2.6.1 Amdahl Kanunu ..... 36
2.6.2 Paralelleştirme Methodları ..... 38
2.6.3 Verimlilik Ölçütleri ..... 39
2.6.4 OpenMP Teknolojisi ..... 39
2.7 Kullanılan Araçlar ..... 47
2.7.1 Kaynak Kodlar, Geliştirme ve Test Ortamı ..... 47
2.7.2 Yazılım Araçları ..... 48
2.7.3 Veri Setleri ..... 48
3 ALAN VERİMLİLİĞİ ÇALIŞMASI ..... 50
3.1 PDSSP Methodu ..... 50
3.1.1 Öngörülü Sürüm Anahtarı (PVS) ..... 53
3.1.2 İşaretsiz Değişken Uzunluklu Sayısal Veri Tipi (ds_Int) ..... 54
3.1.3 Veri Yapısı Paketleme ..... 56
3.1.4 gSpan'e Uygulanması ..... 57
3.1.5 Teorik Sınırları ..... 61
3.1.6 Deneysel Çalışmalar ..... 61
4 ZAMAN VERİMLİLİĞİ ÇALIŞMASI ..... 64
4.1 ANPTR Methodu ..... 64
4.1.1 Methodun Yapısı ..... 64
4.1.2 gSpan'e Uygulanması ..... 69
4.1.3 Teorik Sınırları ..... 72
4.1.4 Deneysel Çalışmalar ..... 72
5 SONUÇLAR VE DEĞERLENDİRME ..... 77
KAYNAKLAR ..... 79
ÖZGEÇMİŞ ..... 84

# KISALTMALAR

| Kısaltma | İngilizcesi | Türkçesi |
|---|---|---|
| ANPTR | Adaptive Nested Parallelism by Thread Requesting | İş Parçacığı İsteyerek Uyarlanır İçiçe Paralelleştirme |
| BFS | Breadth First Search | Genişlemesine İlk Bulma |
| CPU | Central Processing Unit | Merkezi İşlemci Birimi |
| DFS | Depth First Search | Derinlemesine İlk Bulma |
| DOP | Degree of Parallelism | Paralelleşme Derecesi |
| FIFO | First In First Out | İlk Giren İlk Çıkar |
| FPGA | Field Programmable Gate Array | Alanda Programlanabilir Kapı Dizileri |
| FSM | Frequent Subgraph Mining | Sık Alt Çizge Madeciliği |
| Gaston | Graph Sequence Tree Extration | Çizge Sıra Ağaç Özütleme |
| GPU | Graphics Processing Unit | Grafik İşleme Ünitesi |
| gSpan | Graph-based Substructure pattern | Çizge Tabanlı Altyapı Örüntüsü |
| ILP | Integer Linear Programming | Tamsayı Doğrusal Programlama |
| KDD | Knowledge Discovery in Data | Veriden Bilgi Keşfi |
| LIFO | Last In First Out | Son Giren İlk Çıkar |
| LOP | Level of Parallelism | Paralelleşme Seviyesi |
| MapReduce | Mapping and Reducing | Haritalama ve Azaltma |
| MPI | Message Passing Interface | Mesaj Geçirme Arabirimi |
| OpenMP | Open Multi Processing | Açık Çoklu İşleme |
| PDSSP | Predictive Dynamic Sized Structure Packing | Öngörülü Değişken Uzunluklu Yapı Paketleme |
| POSIX | Portable Operating System Interface | Taşınabilir İşletim Sistemi Arayüzü |
| SAÇM | Frequent Subgraph Mining | Sık Alt Çizge Madeciliği |
| SMP | Symetric Multi Processing | Simetrik Çoklu İşleme |
| TID | Transaction Identifier | Hareket Tanımlayıcı |

# ŞEKİLLER

Şekil 2-1: Veriden bilgi keşfi (KDD) işleminin adımları ..... 7
Şekil 2-2: 5 düğüm ve 7 kenardan oluşan bir çizge ..... 8
Şekil 2-3: Etiketli bir çizge ..... 9
Şekil 2-4: Şekil 2-3'teki çizgenin bir alt çizgesi ..... 10
Şekil 2-5: Şekil 2-3'teki çizgeye eşlenik bir çizge ..... 11
Şekil 2-6: Çizge yapıların örnek kullanım alanları ..... 13
Şekil 2-7: (a)(b)(c) bileşikleri içerisindeki ortak bileşikler (d) ve (e)'dir. ..... 14
Şekil 2-8: Apriori (a) tabanlı ve yapı büyümeli (b) aday çizge üretilmesi ..... 17
Şekil 2-9: SAÇM alanındaki algoritmaların sınıflandırılması [38] ..... 21
Şekil 2-10: gSpan'in çalışma stratejisi [27] ..... 22
Şekil 2-11: İleri ve geri kenarları belirtilmiş örnek bir çizge ..... 22
Şekil 2-12: Bir çizgenin üç farklı DFS kod ağacı ..... 24
Şekil 2-13: Şekil 2-12'deki çizgelerin DFS kodları ..... 25
Şekil 2-14: DFS kod ağacı üzerinde destek seviyesi 3 olan alt çizgeler ..... 27
Şekil 2-15: gSpan algoritmasının temsili kodu ..... 28
Şekil 2-16: Doğru hizalanmış ve hizalama hatası olan 4 byte'lık veriye erişim [47]30
Şekil 2-17: Örnek bir veri yapısı ..... 31
Şekil 2-18: Şekil 2-17'deki veri yapısının 4 byte sözcük uzunluklu hizalanışı ..... 32
Şekil 2-19: Şekil 2-17'teki veri yapısının 8 byte sözcük uzunluklu hizalanışı ..... 32
Şekil 2-20: Şekil 2-17'deki veri yapısının tekrar hizalanışı ..... 34
Şekil 2-21: Şekil 2-17'deki veri yapısının 2 byte sözcük uzunluklu hizalanışı ..... 34
Şekil 2-22: Yapı paketleme boyunu 2 byte'a değiştirilmiş bir veri yapısı ..... 35
Şekil 2-23: Tek iş parçacıklı sistemde bir programın çalıştırılması [39] ..... 35
Şekil 2-24: Paralelleştirilmiş bir programın çalıştırılması [39] ..... 36
Şekil 2-25: Paralelleşme miktarına bağlı en yüksek hızlanma eğrisi. ..... 37
Şekil 2-26: OpenMP mimari katmanları ..... 40
Şekil 2-27: OpenMP #parallel direktifi ..... 40
Şekil 2-28: OpenMP #parallel direktifi ile örnek ..... 42

Şekil 2-29: OpenMP # parallel do/for direktifi ..... 42
Şekil 2-30: OpenMP #do/for direktifi ile örnek ..... 44
Şekil 2-31: Seri bir kodun iki farklı OpenMP paralelleştirmesi ..... 44
Şekil 2-32: Critical ve atomic direktiflerinin kullanımı ..... 45
Şekil 2-33: İç içe paralelleştirme örneği ..... 46
Şekil 2-34: False Sharing'in oluşması [58] ..... 47
Şekil 3-1: SAÇM algoritmalarının genel mimarisi ..... 50
Şekil 3-2: PDSSP uygulanmış SAÇM algoritma mimarisi ..... 52
Şekil 3-3: DIMACS biçimindeki giriş veri seti ..... 53
Şekil 3-4: PVS algoritmasının temsili kodu ..... 53
Şekil 3-5: ds_Int algoritmasının temsili kodu ..... 55
Şekil 3-6: Örnek bir yapıda ds_Int ile bellek kazancı ..... 56
Şekil 3-7: Veri hizalama uzunluğu 4 byte olarak Şekil 3-6'nın hesaplanması ..... 56
Şekil 3-8: Yeniden sıralama sonrası Şekil 3-6'nın tekrar hesaplanması ..... 57
Şekil 3-9: gSpan'nin çalışma zamanı sırasındaki bellek profillemesi ..... 58
Şekil 3-10: gSpan içerisindeki yüksek bellek ihtiyacı olan veri yapıları ..... 59
Şekil 3-11: Şekil 3-10'un ds_Int veri tipi ile oluşturulmuş şekli ..... 59
Şekil 3-12: gSpan_PDSSP'nin çalışma zamanı sırasındaki bellek profillemesi ..... 60
Şekil 3-13: gSpan ve gSpan_PDSSP'nin farklı veri setleri ile karşılaştırılması ..... 63
Şekil 4-1: gSpan_ANPTR algoritmasının temsili kodu ..... 67
Şekil 4-2: gSpan_ANPTR'nin çalışma örneklemesi ..... 68
Şekil 4-3: gSpan kodunun çalışma profillemesi ..... 70
Şekil 4-4: gSpan_ANPTR kodunun çalışma profillemesi ..... 71
Şekil 4-5: gSpan_ANPTR için çalışma zamanları ve hızlanma oranları ..... 76

# ÇİZELGELER

Çizelge 1-1: SAÇM algoritmaları ve paralelleştirilmiş sürümleri [8] ..... 4
Çizelge 2-1: Çizge madenciliğinin kullanıldığı popüler alanlar [27]..... 12
Çizelge 2-2: SAÇM alanındaki bilinirliği yüksek algoritmalar ve özellikleri [37] ... 20
Çizelge 2-3: Şekil 2-11'deki çizgenin DFS kodları ..... 23
Çizelge 2-4: Temel veri tipleri ve veri yığın hizalama ihtiyaçları ..... 31
Çizelge 2-5: Kullanılan deneysel veri setleri ve özellikleri ..... 49
Çizelge 3-1: İşaretsiz sayısal verilerin bellek ihtiyacı ve değer aralıkları ..... 51
Çizelge 3-2: ds_Int veri tipi ve temel veri tiplerinin karşılaştırması..... 55
Çizelge 4-1: gSpan algoritmasının çalışma zamanları (saniye) ..... 73
Çizelge 4-2: gSpan_ANPTR için ideal LOP değeri bulunması..... 75

# DENKLEMLER

(2.1) Alt çizge destek seviyesi ..... 15
(2.2) Amdahl kanunu ..... 36
(2.3) Paralel algoritmalarda etkinlik ..... 37
(4.1) ANPTR methodu için ideal LOP değeri..... 73
(4.2) Standart sapma..... 74

# 1. GİRİŞ

Günümüzde kimya ya da biyoloji alanlarında birden fazla bileşik içerisindeki ortak yapıların tesbitinden, internette sosyal ağlarda bulunan kişiler arası ilişkilerin tesbitine dek bir çok alanda sık alt çizge madenciliği teknikleri kullanılmaktadır. Bu teknikler büyük tekil bir yapıda ya da çok sayıda farklı yapıda geçen benzer alt yapıların bulunması için kullanılmaktadır. Veri setlerinin büyük uzunluklara sahip olması ve yüksek karmaşıklık içermesi nedeniyle işlemler için gereken bilgisayar sistemi belleklerinin yüksek olması ihtiyacı ile işlemler için gereken çalışma sürelerinin uzun olması gibi problemler oluşmaktadır.

Bu tez çalışmasında sık alt çizge madenciliği alanındaki sorunlar incelenmiş ve belirlenen iki problem üzerinde iyileştirmeye yönelik yaklaşımlar geliştirilmiştir.

## 1.1 Amaç

SAÇM tekniklerinde girdi olarak kullanılan veri setlerinin yüksek karmaşıklığa ve büyük uzunluklara sahip olması, çizge formuna çevrimlerinin ve üzerinde işlem yapılmasının bilgisayar sistemlerinin disk, bellek ve işlemci gücü açısından yetersiz kalmasına ve bazı durumlarda işleyemez olmasına neden olmaktadır. Bu durum, SAÇM tekniklerinin günümüzde sınırlı sayıda alanda kullanılabilmesine olanak vermektedir.

Bu tez çalışmasında amaçlanan, SAÇM alanında alan ve zaman verimliliğini arttıracak methodların geliştirilmesi ve geliştirilen bu methodların bir algoritmaya uygulanarak yapılacak işlemlerin daha az bellek kullanılarak daha kısa sürede yapılabilmesini sağlamaktır. Böylece SAÇM'nin kullanım alanlarının arttırılmasına yönelik katkı sağlanabilecektir.

## 1.2 Kapsam

SAÇM teknikleri kullanıldığı alana özel yöntemler ve algoritmalar içermektedir. Bu teknikler sonucu elde edilen çıktılar, benzerlik ya da benzeşliklerin belirlenen oranlar için tesbit edilmesidir. Bu çıktıların kullanılabilir en doğru sonuçlar olması için veri setlerinin büyüklüğü günden günde arttırılmaktadır.

Bu tez çalışmasında literatürde birçok alanda kullanılabilir, çıktılarının tutarlı olduğu kanıtlanmış alan karmaşıklığı yüksek ve zaman karmaşıklığı ise diğer tekniklere göre orta seviye olan gSpan algoritması çalışılmak üzere seçilmiştir. gSpan üzerinde geliştirilen teknikler, farklı gerçek ve yapay veri setleri ile test edilmiştir [1]. Bu geliştirmeler belirli bir alana özel kılınmamış böylece SAÇM tekniğinin kullanıldığı ilgili tüm alanlardaki veri setlerinin analizi için kullanılabilir olması hedeflenmiştir.

## 1.3 Problemin Tanımı

SAÇM temel olarak tekrarlı dizilerin (frequent itemset) bulunması işlemi olduğundan, tekrarlı dizilerin bulunması işlemlerinde karşılaşılan problemler SAÇM alanında da geçerlidir.

Tekrarlı dizilerin tespitinde kullanılan algoritmaların zaman ve alan karmaşıklıklarının yüksek olması, giriş dizi boyutlarının büyük olması ve dizi operasyonlarının (transaction) üzerinde tekrar edilme oranını temsil eden destek seviyesi miktarlarınında değişkenliğinin birleşimi sonucunda, bu alandaki algoritmalarda zaman verimliliği ve alan verimliliğinin düşüklüğü problemleri ortaya çıkmaktadır. Bu sorunların neticesinde bilgisayar sistemlerinde hafıza ya da işlem gücü yetersizliği problemleri oluşmaktadır.

SAÇM adımlarından olan alt örüntü benzerliklerin test edilmesi işleminin (subgraph isomorphism) polinom zamanda (NP-complete) çözümlendiği bilinmektedir. Dolayısı ile çoğu örüntü bulma algoritmaları aday üretimi aşamasında gereksiz ve fazlalık aday üretimini azaltmayı hedeflemişlerdir. Bunlardan Yan ve Han tarafından geliştirilen gSpan zamansal performansı ile öne çıkmıştır [1].

SAÇM algoritmalarının çalışmasında giriş verilerin tamamı işlem için bilgisayarın hafızasına alınmakta, alt örüntülerin destek seviyesine bağlı olarak oluşturulan aranılacak örneklemeleri de aynı şekilde bilgisayar hafızasında depolanmaktadır. Bu durum giriş büyüklüğünün uzunluğu ve karmaşıklığına bağlı olarak, hafızada zaman zaman yüzlerce kat yer kaplamaktadır [2][3].

Yukarıdaki iki temel problem dışında, çizge madenciliğinde methodların çalışma şekline göre aşağıda belirtilen sorunlar da mevcuttur.

- Arama stratejisi (Search Strategy): Alt çizgelerin araştırılması için kullanılan iki teknik mevcuttur. Bunlar genişletilmiş ilk arama (Breadth First Search) ve derinlemesine ilk arama (Depth First Search) metotlarıdır. Bu yöntemlerin temelinde yatan yüksek bellek kullanım ihtiyacı yada aramanın ana elemandan uzakta olması durumunda çalışma zamanının uzaması gibi problemler SAÇM içinde geçerlidir [4].
- Giriş verisinin yapısı (Nature of Input): Giriş verisi olarak kullanılacak veri setleri tek bir çizgeden ya da birden fazla çizgeden oluşabilir. Bu durumlarda, giriş veri setinin karmaşıklığı ve büyüklüğü problemler oluşturmaktadır.
- Çıktı veri setinin durumu (Completeness of the Output): Algoritmalar çalışma çıktısı olarak tüm alt çizgeleri ya da parçalı alt çizgeleri dönebilir [5].

## 1.4 Literatür Araştırması

SAÇM alanındaki literatür çalışmaları incelendiğinde alan verimliliği arttırmaya yönelik belirgin çalışmaların olmadığı, daha çok giriş verisinin şeklinde yada işlenme mantıklarındaki değişikliklerle bu konuda ilerlenme sağlandığı gözlenmiştir. Dolayısıyla bu tez çalışmasındaki alan verimliliği yaklaşımıyla karşılaştırılabilir literatür örneği bu tez kapsamında verilememektedir.

Zaman verimliliğini arttırmaya yönelik bir çok çalışma literatürde tesbit edilmiştir. Her alanda olduğu gibi işlemlerin hızlı yapılabilmesini sağlamak önemlidir ve SAÇM alanındaki algoritmalarda da gerek algoritma mantıklarının değiştirilmesi ile gerekse değişik paralelleştirme tekniklerinin kullanımı ile bu alanda iyileştirmeler sağlandığı görülmektedir [6][7]. Çizelge 1-1'de bu alanda bilinirliği yüksek algoritmaların tek iş parçacıklı orijinal halleri ve iyi bilinen paralelleştirilmiş sürümleri mevcuttur.

Çizelge 1-1: SAÇM algoritmaları ve paralelleştirilmiş sürümleri [8]

| **Çalışma tipi** | **Kısaltma** | **İsmi** |
|---|---|---|
| Seri | AGM | Apriori-Based Graph Mining |
| Seri | FSG | Frequent Subgraph Mining |
| Seri | gSpan | Graph-Based Substructure Pattern Mining |
| Seri | HSIGRAM | Horizontal Single Graph Miner |
| Seri | MoFA | Molecular Fragment Miner |
| Seri | SUBDUE | Substructure Discovery |
| Seri | VSIGRAM | Vertical Single Graph Miner |
| Paralel | p-MoFa | Paralel Mofa |
| Paralel | p-gSpan | Paralel gSpan |
| Paralel | d-MoFa | Dinamik Yük Dengelemeli Dağıtık MoFa |
| Paralel | MotifMiner | MotifMiner Araç Takımı |
| Paralel | MRFSE | MapReduce Tabanlı Sık Alt Çizge Çıkarımı |
| Paralel | MRPF | MapReduce Tabanlı Örüntü Bulma |
| Paralel | SP-SUBDUE | Sabit Bölümlemeli SUBDUE |
| Paralel | SP-SUBDUE2 | Sabit Bölümlemeli SUBDUE2 |

Çizelge 1-1'deki paralelleştirilmiş algoritma sürümleri genellikle iş parçacığı tabanlı (thread-based) ya da MapReduce teknolojisi kullanan paralelleştirmelerdir

[9][10]. Yukarıdaki çalışmalar dışında literatürde, VSigram algoritması üzerinde OpenMP tekniği ile 12 iş parçacıklı bir sistemde 11X hızlanma [11], gSpan'ın GPU tabanlı 448 çekirdekli bir sistemde 9X'e varan hızlanma [12], gSpan algoritmasının FPGA tabanlı 8 çekirdekli işlemci üzerinde 6.2X'e varan hızlanma [13], gSpan algoritmasının iş kuyruklama (task queuing) yapısı ile 32 iş parçacıklı bir sistemde 27X'e varan hızlanma [14], gSpan algoritmasının başka bir 12 iş parçacıklı paralelleştirmesinde tutarsız sonuçlar olmakla birlikte 5X ve 11X hızlanma [15], gSpan algoritmasının 2496 çekirdekli bir GPU üzerinde CUDA dinamik paralelizasyon ile 80X hızlanma [16] sağladığı sonuçlar mevcuttur [17].

### 1.5 Literatüre Katkı

Bu tezte SAÇM teknikleri ile ilgili belirlenen iki temel probleme yönelik yeni çözüm yaklaşımları geliştirilmiştir. Böylece SAÇM tekniklerinin geliştirilmesi ve kullanım alanlarının arttırılmasına yönelik katkı sağlanmıştır.

SAÇM teknikleriyle ilgili üzerinde çalışılan birinci problem alan verimliliğinin arttırılmasıdır. Probleme yönelik literatürde belirgin bir genel çözüm ya da yaklaşıma dair yayın tesbit edilememiştir.

Üzerinde çalışma yapılan ikinci problem ise zaman verimliliğinin arttırılmasıdır. Dördüncü bölümde detaylandırılan çalışma, SMP sistemlerde OpenMP tekniği kullanılarak yapılan çok seviyeli paralelleştirmenin gSpan algoritmasına uygulandığı literatürdeki ilk çalışma olarak görünmektedir. Bununla birlikte uygulama tekniğinin özyinelemeli (recursive) algoritmalarda da kullanılabilir şekilde açıklanması, benzeri zaman karmaşıklığı olan diğer problemlerde de uygulanabilir olmasını sağlamaktadır [18][19].

## 1.6 Tezin Yapısı

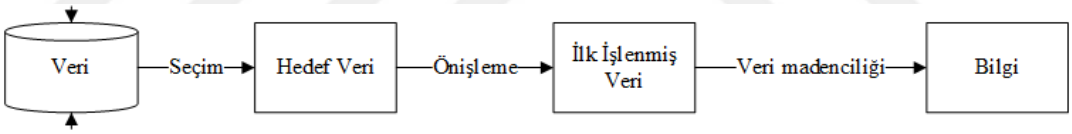
Bu tez 5 bölümden oluşmaktadır. Birinci bölümde tezin amacı, kapsamı ve literatüre katkısı belirtilmiş, ikinci bölümde yapılan çalışmaların içerdiği temel kavramlar açıklanarak yapılan deneysel çalışmalar için gerekli olan araçlar aktarılmıştır. Üçüncü ve dördüncü bölümlerde sırasıyla alan verimliliği ve zaman verimliliği çalışmaları detaylandırılmıştır. Son bölümde, tez çalışmasında elde edilen sonuçlar özetlenmiş ve değerlendirilmiştir.

# 2 ÖNBİLGİ VE KULLANILAN ARAÇLAR

Bu bölümde, tezin içerdiği çalışmalar için gerekli olan kavram ve araçlar anlatılmıştır. Bunlar çizge yapılar, veri madenciliği, SAÇM teknikleri, SAÇM algoritmaları ve kullanımları sırasında karşılaşılan sorunlar, alan ve zaman verimliliği geliştirmeleri için gerekli temel teknik düzey bilgiler ve deneylerde kullanılan veri setleri ile geliştirme ortamlarıdır.

## 2.1 Veri Madenciliği

Veri madenciliği "Veriden Bilgi Keşfi (KDD)" işinin bir parçasıdır. Bu işlemde amaç büyük ve ham halde bulunan bütük veri setlerinden belirli bir amaç ya da alana yönelik anlamlı bilgilerin çıkarılmasıdır. KDD işlemi Şekil 2-1'de gösterilmektedir.




Şekil 2-1: Veriden bilgi keşfi (KDD) işleminin adımları

KDD işleminin ana adımlarının gerçekleşmesi için ara işlem sırasıyla veri seçimi (data selection), önişleme (pre-processing) ve sonraki işlemler yani veri madenciliği dir (post-processing).

**Tanım (Yapı - Pattern):** Genel olarak yapı, verilen bir P obje popülasyonunu karakterize eden boşluk içermeyen ve sonlu sayıdaki özelliklerin birleşimidir. Bu yapı P içerisindeki yüksek frekanslı kısımları belirler.

**Tanım (Veri madenciliği – Data mining):** Ham veriler üzerinde bilgisayar ve hesaplama tekniklerinin kabul edilebilir hesaplama sınırları içerisinde kalarak uygulanmasıdır.

**Tanım (Veriden bilgi bulma – Knowledge discovery in data):** Veriler üzerinde belirli tekniklerin uygulanması sonucu anlamlı, kullanılabilir yapılar bulunmasıdır [20].

## 2.2 Çizge Madenciliği

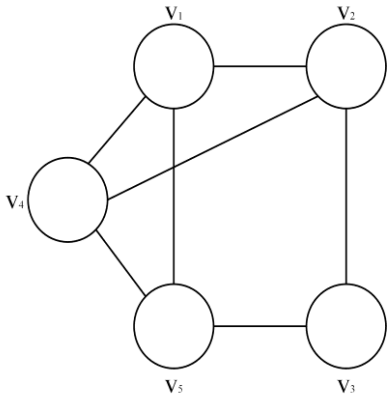
Çizge madenciliği değişik araç ve teknikler kullanarak çizge haline çevrilmiş veri üzerinden işe yarar bilgiler elde etme teknikleridir.

Çizge, nesnelerin birleşip oluşturduğu yapılardır. Bir çizgedeki her nesne bir düğümü (vertices ya da node) temsil eder. Düğümler arasındaki bağlantıları oluşturan çizgiler ise ayrıt ya da kenar (edge) olarak adlandırılır. Çizge yapılar içerisindeki her kenar bağımsız iki düğümü birleştirmektedir.

**Tanım (Çizge - Graph):** Bir çizge matematiksel olarak $G = (V, E)$ olarak ifade edilir. Burada:

- V, çizge içerisindeki düğümleri belirtir.
- $E \subseteq VxV$, düğümler arasındaki bağlantıları yani kenarları ifade eder.

Şekil 2-2, 5 düğüm ($v_1$, $v_2$, $v_3$, $v_4$, ve $v_5$) ve 7 kenardan (($v_1$, $v_2$), ($v_1$, $v_4$), ($v_1$, $v_5$), ($v_2$, $v_4$), ($v_2$, $v_3$), ($v_3$, $v_5$), ($v_4$, $v_5$)) oluşan bir çizgeyi göstermektedir.



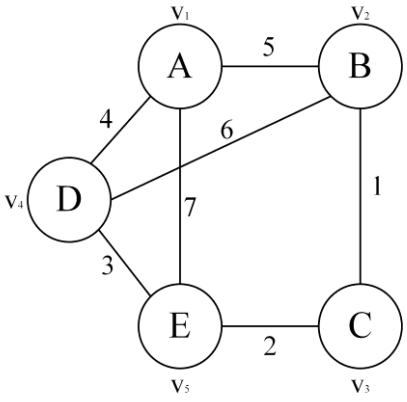

Şekil 2-2: 5 düğüm ve 7 kenardan oluşan bir çizge

**Tanım (Etiketli Çizge – Labeled Graph):** Etiketli çizge bir çizge türü olup değişebilen 4 niceliğe $G = (V, E, L, I)$ sahiptir. Bunlar:

- V, çizge içerisindeki düğümleri belirtir.
- $E \subseteq VxV$, düğümler arasındaki kenarları ifade eder.
- L, etiketlerin listesidir.
- $I : V \cup E$, etiketleme fonksiyonudur.

Şekil 2-3 etiketlenmiş bir çizgeyi göstermektedir. Bu çizgenin değişken 4 nicelik bilgileri şu şekilde ifade edilebilir.

- Düğüm: $V = \{\mathrm{v}_1, \mathrm{v}_2, \mathrm{v}_3, \mathrm{v}_4, \mathrm{v}_5\}$.
- Kenar: $E = \{(\mathrm{v}_1, \mathrm{v}_2), (\mathrm{v}_1, \mathrm{v}_4), (\mathrm{v}_1, \mathrm{v}_5), (\mathrm{v}_2, \mathrm{v}_4), (\mathrm{v}_2, \mathrm{v}_3), (\mathrm{v}_3, \mathrm{v}_5), (\mathrm{v}_4, \mathrm{v}_5)\}$
- Etikerler: $L = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, 1, 2, 3, 4, 5, 6, 7\}$.
- Etiketleme fonksiyonu: $I(\mathrm{v}_1) = A,\ I(\mathrm{v}_2) = B,\ I(\mathrm{v}_3) = C,\ I(\mathrm{v}_4) = \mathrm{D},\ I(\mathrm{v}_5) = E,\quad I(\mathrm{v}_1, \mathrm{v}_2) = 5,\ I(\mathrm{v}_1, \mathrm{v}_4) = 4,\ I(\mathrm{v}_1, \mathrm{v}_5) = 7,\ I(\mathrm{v}_2, \mathrm{v}_4) = 6, I(\mathrm{v}_2, \mathrm{v}_3) = 1, I(\mathrm{v}_3, \mathrm{v}_5) = 2\ ve\ I(\mathrm{v}_4, \mathrm{v}_5) = 3.$



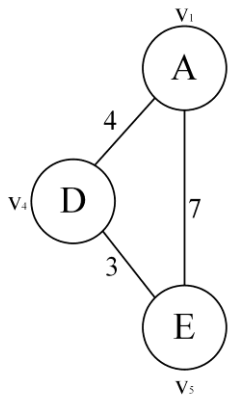

Şekil 2-3: Etiketli bir çizge

Alt çizge (subgraph), çizge eşleniklik kontrolü (isomorphism) ve alt çizge eşleniklik kontrolü kavramlarının tanımları sırasıyla aşağıdaki şekildedir.

**Tanım (Alt çizge – Subgraph):** Eğer aşağıdaki şartlar sağlanıyorsa $G = (V, E)$ olarak ifade edilen bir çizgenin alt çizgesi $G' = (V', E')$ olarak ifade edilebilir:

- $V' \subseteq V$ ve
- $E' \subseteq E \cap (V' \times V')$.

Şekil 2-4'te bulunan çizge, Şekil 2-3'te bulunan çizgenin bir alt çizgesidir. Bu çizge üç düğüm $(v_1, v_4, v_5)$ ve üç kenar $((v_1, v_4), (v_4, v_5)$ ve $(v_5, v_1))$ içermektedir.

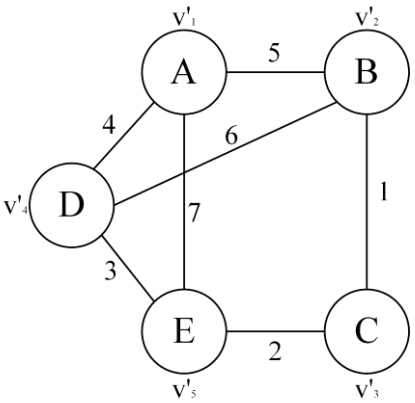



Şekil 2-4: Şekil 2-3'teki çizgenin bir alt çizgesi

**Tanım (Çizge eşlenikliği – Graph Isomorphism):** Örneğin G ve H gibi 2 çizge arasında şöyle bir fonksiyon tanımlanırsa $f : V(G) \rightarrow V(H)$ ve bu fonksiyon hem birebir hemde örten ise, G çizgesinin herhangi iki düğümü *u* ve *v* birbirlerine bitişiktir G içerisinde eğer ve yalnız $f(u)$ ve $f(v)$ H çizgesi içerisinde bitişikse, bu noktada G ve H eşleniktir diyebiliriz.

Şekil 2-5 içerisindeki çizge, Şekil 2-3 içerisinde bulunan çizgeye eşleniktir çünkü $f(v_1) = v'_1, f(v_2) = v'_2, f(v_3) = v'_3, f(v_4) = v'_4$ ve $f(v_5) = v'_5$.

Şekil 2-5'deki tüm bitişik düğümlerin aynı zamanda Şekil 2-4'tede bitişik düğümler olduğu görünmektedir.




Şekil 2-5: Şekil 2-3'teki çizgeye eşlenik bir çizge

**Tanım (Alt çizge eşlenikliği – Subgraph Isomorphism):** Bir çizge $G' = (V', E')$, başka bir çizgenin $G = (V, E)$ alt çizgesi ve ilgili kısmın eşleniğidir denilebilir, eğer şu şartlar sağlanıyorsa:

- $f : V'(G') \rightarrow V(G)$, şeklinde bir fonksiyon tanımlanacak,
- Ve her $(u, v) \in E'$ için $\big(f(u), f(v)\big) \in E$ yazılacaktır.

Her alt çizge aynı zamanda ilgili çizge kısmı için eşleniktir. Şekil 2-4'te gösterilen çizge, Şekil 2-3'te gösterilen çizgenin alt çizgesi ve ilgili alt çizgenin eşleniğidir (subgraph-isomorphic). Çünkü, Şekil 2-4'teki çizgenin tüm bitişik düğümleri aynı zamanda Şekil 2-3'teki çizge içerisindede bitişim düğümlerdir [21] [22].

## 2.3 Çizgelerin Kullanım Alanları

Çizgeler günümüzde değişik alanlarda kullanılmaktadır [23]. Şekil 2-6'da bazı örnekleri verildiği üzere kimsayal bileşikler, biyolojik veriler, sosyal ağlarda ilişkiler, program akış döngüleri, uçuş rotaları, bilgisayar ağları gibi verilerin büyük olduğu, anlamlandırılma desteğinin gerektiği yerlerde kullanılmaktadır. Çizge madenciliğinin popülar kullanım alanları ise Çizelge 2-1'de verilmiştir [24]. Bu kadar çeşitli alanlarda kullanılan çizge madenciliği konusunda, ilgili alana özel tekniklerin geliştirilmeside gerekmektedir [25][26].

Çizelge 2-1: Çizge madenciliğinin kullanıldığı popüler alanlar [27]

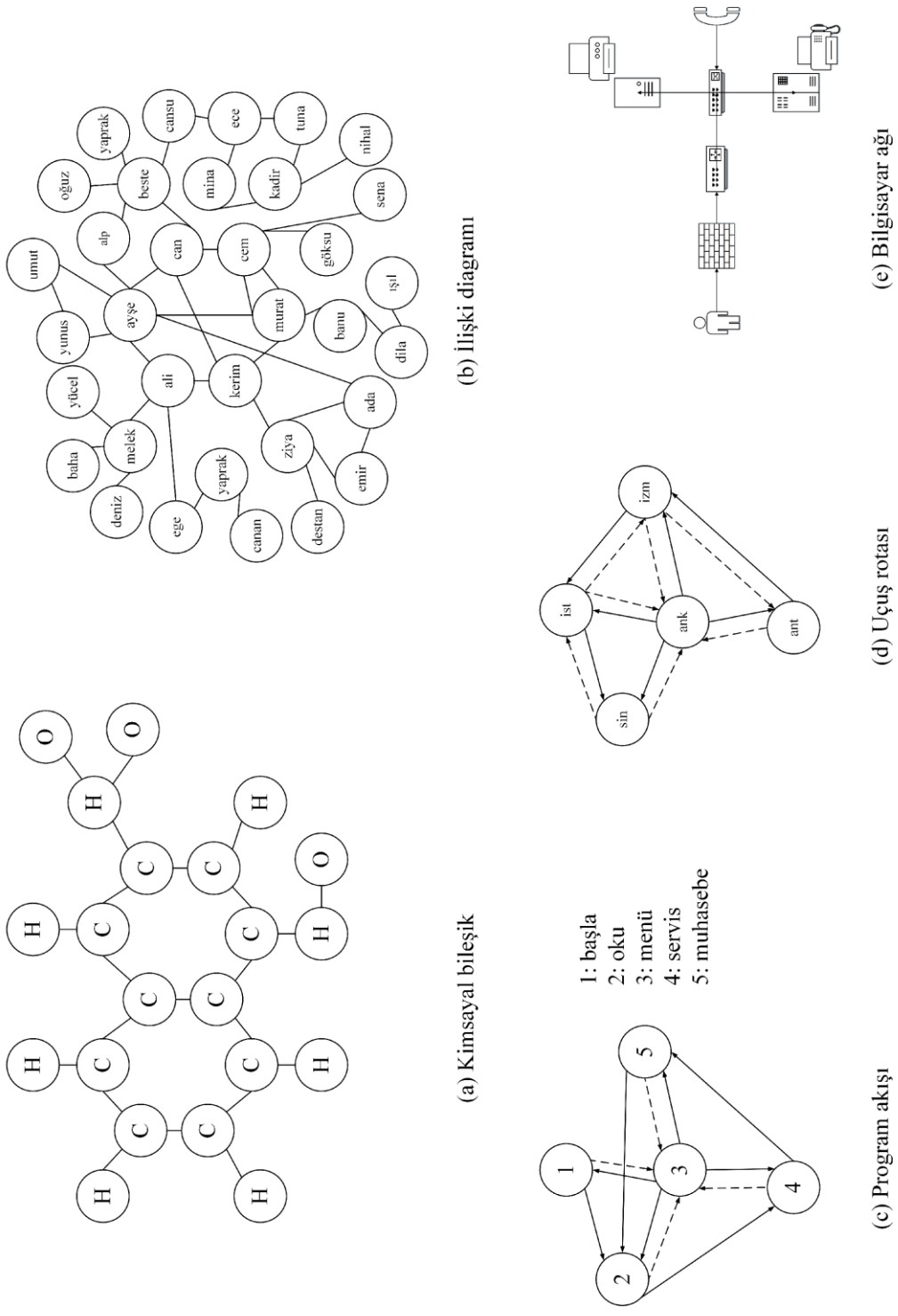
- Sık alt çizge madenciliği (Frequent subgraph mining)
- İlintilendirilmiş çizge örüntü madenciliği (Correlated graph pattern mining)
- En iyi çizge örüntü madenciliği (Optimal graph pattern mining)
- Yaklaşık çizge örüntü madenciliği (Approximate graph pattern mining)
- Çizge örüntü özetleme (Graph pattern summarization)
- Çizge sınıflandırma (Graph classification)
- Çizge kümeleme (Graph clustering)
- Çizge indisleme (Graph indexing)
- Çizge arama (Graph searching)
- Çizge çekirdekleri (Graph kernels)
- Bağlantı madenciliği (Link mining)
- Örümcek ağı yapı madenciliği (Web structure mining)
- İş akış madenciliği (Work-flow mining)
- Biolojik ağ madenciliği (Biological network mining)

## 2.4 Sık Alt Çizge Madenciliği

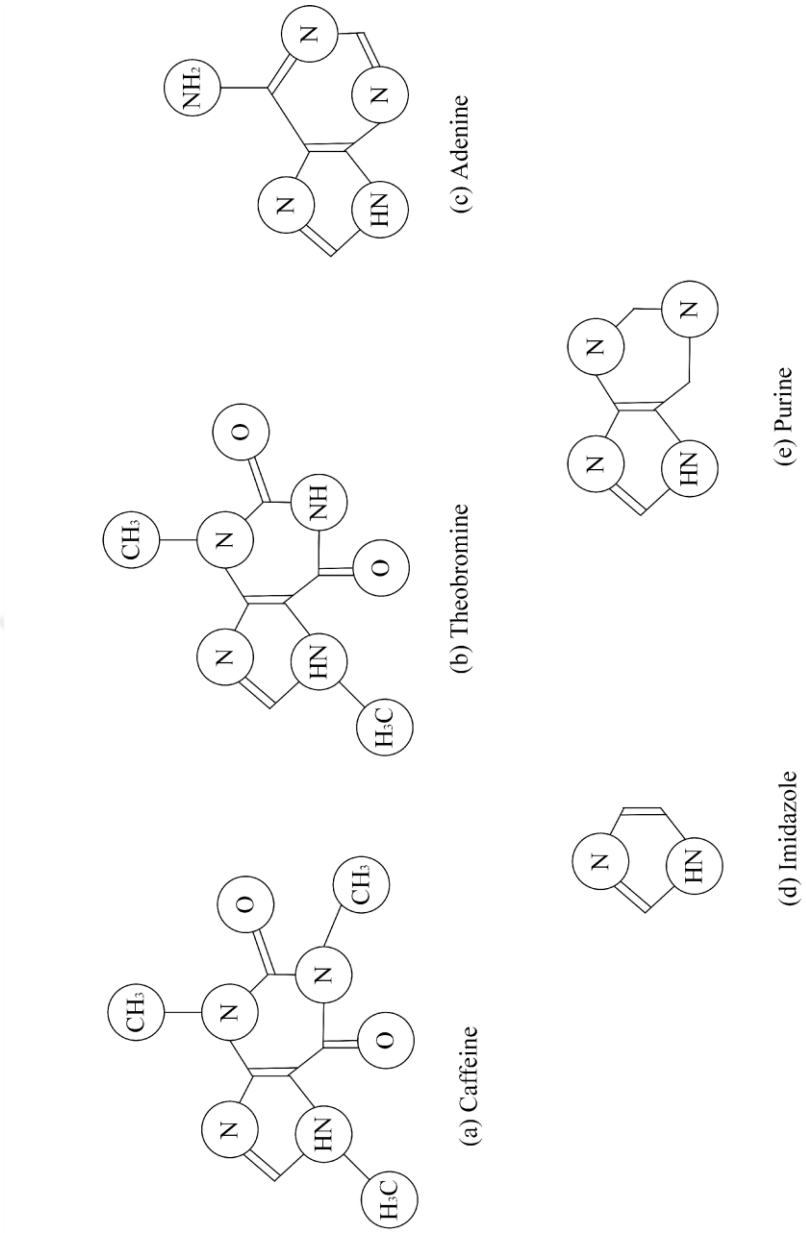
### 2.4.1 Tanım ve SAÇM Problemleri

SAÇM teknikleri iki farklı modeldeki veriler için uygulanmaktadır. Birincisi, birden fazla hareket içeren çizge yapıdaki veriler, ikincisi ise tek bir çizge yapıdır. Birden fazla hareket içeren yapılarda giriş verisi birden fazla orta büyüklükte çizge içermektedir. Tekli çizge yapılarında ise giriş tek ve büyük bir veri yapısından oluşmaktadır.

Bu çalışmada, birden fazla hareket içeren çizgesel yapılar üzerinde durulmuştur. Alt çizge madenciliği için gerekli olan alt çizge destek seviyesi ve çizge hareket tanımlaması aşağıda yapılmıştır. Örnek olarak, Şekil 2-7'de üç farklı kimyasal bileşik içerisinde geçen sık alt çizgelerin gösterimi mevcuttur.




Şekil 2-6: Çizge yapıların örnek kullanım alanları




Şekil 2-7: (a)(b)(c) bileşikleri içerisindeki ortak bileşikler (d) ve (e)'dir.

**Tanım (Alt çizge destek seviyesi – Subgraph support level):** *D* giriş veri seti olmak üzere, *G* çizgesi için destek seviyesi $G_S$, Sup $(G_D)$ olarak gösterilir ve Denklem 2.1'deki formül ile hesaplanır.

$$Sup\ (G_S) = \frac{D\ içerisindeki\ G_S\ destek\ seviyesini\ sağlayan\ toplam\ çizge\ sayısı}{D\ içerisindeki\ toplam\ çizge\ sayısı} \qquad (2.1)$$

**Tanım (Çizge hareketleri – Graph transaction):** Verilen en küçük destek seviyesi $G_S$ olmak üzere, *V* veri seti içerisinde bu en küçük destek seviyesini sağlayan tüm alt çizgelerin oluşturduğu çizge listesine verilen isimdir.

### 2.4.2 SAÇM Tekniklerinin Özellikleri

SAÇM teknikleri çizge yapılardan alt çizgelerin çıkarılması için değişik yaklaşımlar içermektedir. Bu farklılıklar arama teknikleri, aday çizge yapıların seçimi ve destek seviyesi hesaplamada ortaya çıkmaktadır.

**Arama teknikleri (Search strategy):** SAÇM teknikleri içerisinde kullanılan iki arama methodu vardır. Bunlar derinlemesine ilk bulma (Depth First Search) ve genişlemesine ilk bulma (Breadth First Search) methodlarıdır.

DFS methodu, çizge ya da ağaç (tree) yapılarda gezinme ve arama için kullanılır. Önce ana (root) noktadan başlar ve gidebileceği daha derin bir nokta kalmayıp geri dönmesi gerekene dek ilerler. Şekil 2-3'teki çizge için, DFS B düğümünden başlayarak komşuları olan A, C ve D'yi belirledikten sonra önce D'yi ardından C ve A'yı bir yığın (stack) içerisine koyar. Yığından son gireni ilk olarak (LIFO) alır, yani A'yı alarak ona gider ve bitişik komşularına bakarak D'yi ve E'yi bularak yığın içerisine E'yi ve ardından D'yi koyar. D şuanda yığın içerisinde 2 defa mevcuttur. Bu noktada önemli bir durum vardır. Oda C'den sonra E ve D'nin yığın içerisine eklenmiş olmasıdır. Bu

durum onların C'den önce işlenmesine neden olur. DFS'in buradaki kuralı, en yakında ziyaret edilmiş (A) düğümün kalan diğer düğümlerden (B) önce işlenmesi gerekliliğidir. Bu özyinelemeli işlem tüm düğümler ziyaret edilene kadar devam eder.

BFS methodu ise, iki işlem ile sınırlıdır.

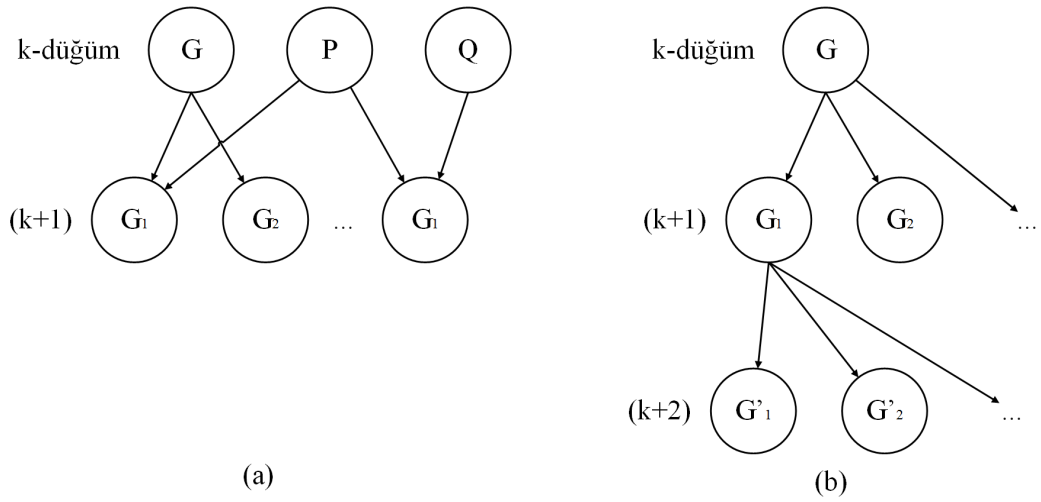
1. Çizgedeki bir düğümü ziyaret et ve işaretle,
2. Ziyaret edilen düğümün komşularını belirle ve onları ziyaret et.

BFS ana düğümden başlayarak bitişik komşuların hepsini sırasıyla ziyaret eder. Ardından ziyaret edilen komşuların ziyaret edilmemiş tüm komşuları belirlenerek sırasıyla ziyaret edilir. Örneğin, Şekil 2-3'deki çizge üzerinde BFS methodu uygulanırsa şu şekilde ilerlenir. B düğümünden başlanırsa D, C ve A sırasıyla ziyaret edilecek listeye eklenir. Burada en son eklenen değil ilk eklenen (FIFO) düğüm öncelikle yığından alınır. Dolayısıyla D ziyaret edilir. D'nin başka bitişik komşusu olmadığından C ziyaret edilir. C, E ile bitişik komşu olduğundan, E yığına eklenir. Bu nedenle E'den önce A ziyaret edilmiştir. B'nin ziyaret edilmesinden sonra B'ye bitişik komşu olan düğümlerin hepsi ziyaret edilir. Ziyaret edilecek bir düğüm kalmadığından işlem tamamlanmış olur.

**Aday yapıların üretilmesi (Generation of candidate patterns):** Olası aday alt çizge yapılarının üretilmesi işlemi SAÇM tekniklerinin önemli bir parçasıdır. Bu alanda bazı algoritmalar aday yapıların üretilmesi işlemini yapmadanda çalışmakla birlikte birçok algoritma temel olarak iki aday yapı üretme tekniğini kullanmaktadır. Şekil 2-8,'te bu methodlar gösterilmektedir. Bunlar apriori (apriori-based) tabanlı ve yapı büyüme (pattern growth-based) tabanlı tekniklerdir.

Apriori tabanlı teknik temelde apriori tabanlı sık çizge bulma algoritmasına benzemektedir. Şekil 2-8a' da gösterildiği üzere, sık çizgeler öncelikle küçük uzunluktaki çizgelerin bulunması ve ardından her iterasyonda bu çizgelere bir düğümün eklenmesi ile devam eder ve çizgeler bu şekilde büyütülerek ilerlenir. Yapı

büyümeli teknik ise Şekil 2-8b' de gösterildiği üzere, bulunan çizgelerin birbirlerine katılarak (join) yeni yapıların oluşturulması şeklinde ilerlemektedir.




Şekil 2-8: Apriori (a) tabanlı ve yapı büyümeli (b) aday çizge üretilmesi

Ardından yeni üretilen çizgelerin sıklığı (frequency) kontrol edilir. Apriori tabanlı algoritmaların üretim iş yükü yüksektir. Bunun nedeni üretilen aday çizge miktarının fazlalığıdır. Şöyleki, k miktardaki aday için, k miktarın iki katı kadar aday (k+1) miktardaki çizge adayın üretilmesi için birleştirilmektedir [28].

Yapı büyümeli algoritmalar, Şekil 2-8b'de olduğu gibi mümkün olan her yönde bir kenar ekleyerek çizgeleri büyütüp adayları ortaya çıkarmaktadır. Her yöne ekleme işlemi nedeniyle, üretilen bir aday çizgenin birden fazla kez üretilmesi problemi oluşabilir. Bu sorunu çözmek için gSpan algoritmasında en sağda olandan büyüme (right-most) methodu geliştirilmiştir. Method basitçe aday çizge genişletilirken her zaman kenarı en sağa ekleyerek ilerlemek şeklinde çalışır. En sağdan devam etme tekniği, başlangıç düğümünden son düğüme kadar sürdürülmekte ve çizgelerin aranması DFS tekniği ile yapılmaktadır [29].

**Destek seviyesi hesaplanması (Support computing):** Çizgelerin sayılmasında birden fazla method kullanılmaktakdır. Bazı SAÇM algoritmaları hareket tanımlayıcıları

(TID) üzerinden sıklık sayımı yapar. Her bir sık alt çizgenin kendi TID'i vardır. *k* sayıdaki alt çizgenin sıklığını hesaplamak, (k-1) sayıdaki alt çizgenin TID listesinin kesişimin hesaplanması şeklindedir. Ardından DFS sözlük sıralaması (lexicographic ordering) sıklık hesaplaması için kullanılabilir. Her bir çizge DFS sıralaması yapılır ve arama ağacı oluşturulur. Ardından en küçük (minimum) DFS kodları bu ağaçtan oluşturularak sıklık hesaplamasına yardımcı olacak hale getirilir. İçe gömük listeler (embedding list) destek hesaplaması için kullanılır [30]. Tüm çizgeler için içe gömük listeler oluşturulurken, her bir çizgenin içerisindeki ön çizge yapılarını gösteren birer dizi (array) ile ilgili çizgedeki ve veri yapısındaki çizge tanımlayıcısı kullanılır. Çizgenin sıklığı içe gömük listelerdeki sıklığından hesaplanır. Diğer bir method ise, aktif çizgelerin kaç kere oluştuğunun tek tek sayılarak bulunmasıdır.

**Yapı tipleri (types of patterns):** Alt çizge yapılarının tesbiti için farklı amaca hizmet eden SAÇM algoritmaları geliştirilmiştir. Örneğin, gSpan ve FFSM algoritmaları belirtilen destek seviyesine eşit ya da büyük sıklıktaki alt çizgeleri bulurlar [31].

**Tanım (Sık alt çizge – Frequent subgraph):** $D = \{G_1, \ldots, G_K\}$ giriş veri seti olmak üzere, belirlenen destek seviyesi $S$ ise, sık alt çizgeler $Support(G', V) \geq \varnothing$ olarak ifade edilir.

**Tanım (Kapalı sık alt çizgeler – Closed frequent subgraph):** Bir *G* çizgesinin sık alt çizgesi olan *G'* , eğer *G* içerisinde aynı destek seviyesinde olan ve *G'* nü içeren bir başka alt çizge mevcut değilse kapalı sık alt çizge olarak kabul edilir.

Kapalı sık alt çizge methodu (CloseGraph), gSpan algoritmasının bir devamı olarak geliştirilmiştir. Kapalı sık alt çizge bulma işlemi aynı destek seviyesinde yapıldığında sık alt çizge bulma işlemi ile aynı miktar işlem gerektirmektedir [32].

Algoritmalardan SPIN ve Gaston maksimum sık alt çizgeleri bulmak üzere geliştirilmiştir [33][34].

### 2.4.3 SAÇM Algoritmaları

Bu alanda bilinirliği en yüksek on algoritma Çizelge 2-2'de gösterilmiştir. Şekil 2-9'da, SAÇM alanında bilinen algoritmaların çalışma yapılarına göre sınırlandırılması verilmiştir [35].

gSpan algoritması, SAÇM tekniği itibarıyle çıktıları tam ve tutarlı bulması, en sağ uzantı özelliğini kullanması, hafıza kullanım oranının ortalama olması, literatürde üzerinde bir çok çalışma yapılan algoritmalardan olması nedeniyle, bu tez çalışması kapsamında çalışılmak üzere seçilmiştir. Algoritmanın çalışma detayları ve temsili kodu (pseudo code) sıradaki bölümde detaylandırılmıştır [36].

#### 2.4.3.1 gSpan Algoritması

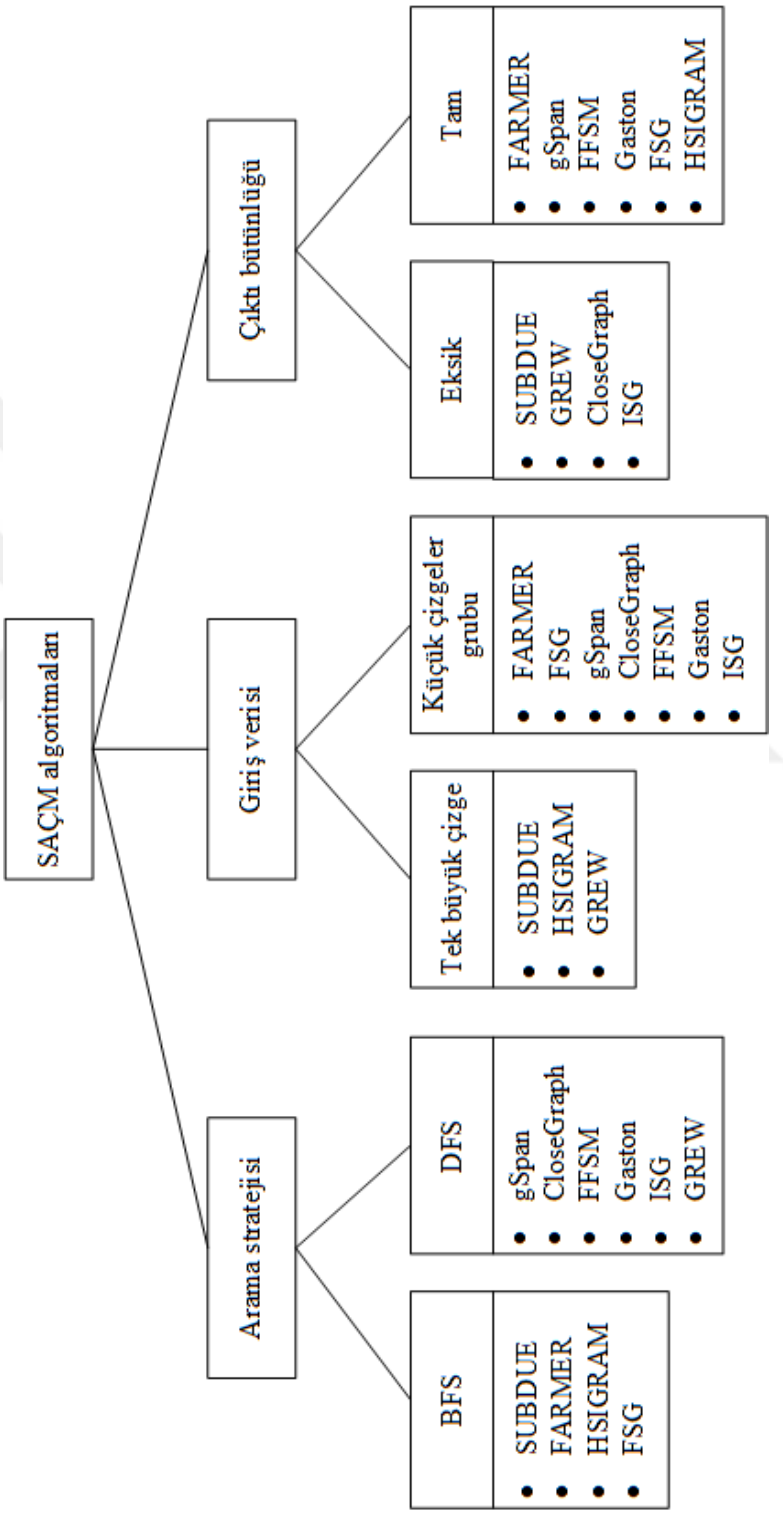
gSpan algoritması 2002 yılında Xifeng Yan ve Jiawei Han tarafından geliştirilmiştir. Teknik özellikleri açısından ilk DFS kullanımını barındırması, özyinelemeli çalışması, aday alt çizge yapılarını oluşturmaya gerek duymadan ve yanlış olumlu aday temizleme (false positive pruining) yapmadan sonuç üretebildiği görülmektedir.

gSpan'in genel çalışma mantığı aşağıdaki adımlardan oluşur ve yapısı Şekil 2-10'daki gibidir.

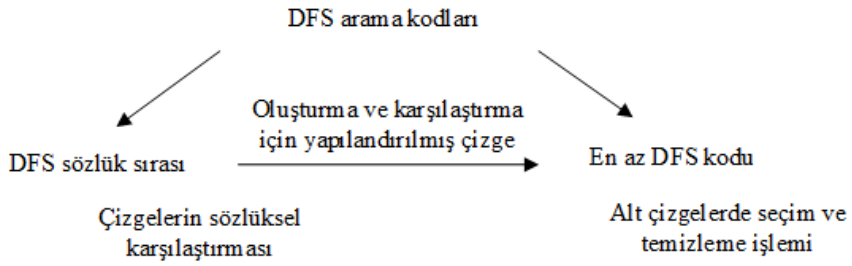
- Çizgeleri oluşturup tek tek aramak ve her biri için çizge eşleniklik kontrolü yapmak yerine, çizgelerin DFS kodları oluşturulur.
- Oluşturulan DFS kodlarının en küçük olanları belirlenir.
- En küçük DFS kodları üzerinden DFS kod ağacı (DFS code tree) oluşturulur.
- DFS kod ağacı üzerinde gerekli seçim ve temizlik (selection, pruning) işlemleri yapılarak sık alt çizge yapılar tesbit edilir.

Çizelge 2-2: SAÇM alanındaki bilinirliği yüksek algoritmalar ve özellikleri [37]

| Algoritma | Çizge gösterimi | Alt çizge oluşturma | Algoritmik yaklaşım | Sıklık hesaplama |
|---|---|---|---|---|
| SUBDUE | Komşuluk matrisi | Düzey-yöntemli arama | Örüntü büyütme | En küçük tanımlama uzunluk kodu ve geçmiş bilgi |
| FARMER | Trie yapısı | Düzey-yöntemli arama | ILP ve apriori tabanlı | Trie veri yapısı |
| FSG | Komşuluk listesi | Bir düğüm uzatma | Apriori tabanlı | TID listesi |
| gSpan | Komşuluk listesi | En sağ uzatma | Örüntü büyütme | DFS sözlük sırası |
| CloseGraph | Komşuluk listesi | En sağ uzatma | Örüntü büyütme | DFS sözlük sırası |
| HSIGRAM | Komşuluk matrisi | Özyineli birleştirme | Apriori tabanlı | En büyük bağımsız küme |
| GREW | Seyrek çizge gösterimi | Özyineli birleştirme | Örüntü büyütme | En büyük bağımsız küme |
| FFSM | Komşuluk listesi | Birleştirme ve uzatma | Apriori tabanlı | Optimize edilmiş komşuluk matris ağacı |
| Gaston | Kargaşa tablosu | Uzatma | Örüntü büyütme | Gömülü liste |
| HSIGRAM | Düğüm üçlü | Düğüm üçlü uzatma | Apriori tabanlı | TID listesi |

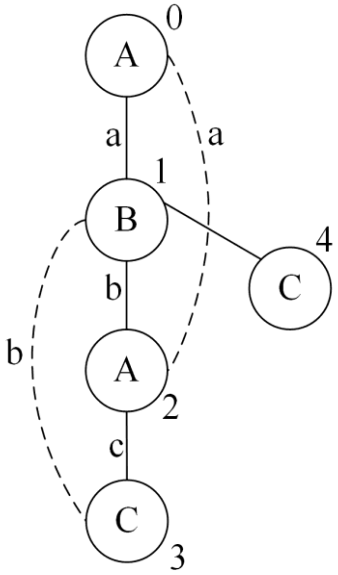



Şekil 2-9: SAÇM alanındaki algoritmaların sınıflandırılması [38]




Şekil 2-10: gSpan'in çalışma stratejisi [27]

Şekil 2-11'de örnek bir çizge, ileri kenarları düz çizgilerle geri kenarları ise kesikli çizgilerle olacak şekilde gösterilmiştir.




Şekil 2-11: İleri ve geri kenarları belirtilmiş örnek bir çizge

**Tanım (İleri kenar ve geri kenar - Forward edge ve backward edge):** Verilen bir çizge yapı $G$ için, ileri kenar (forward edge) DFS ağacı içerisindeki tüm kenarları içerir ve geri kenar (backward edge) DFS ağacı içerisinde bulunmayan kenarları içerir. Basitçe $(i, j)$ bir düğümü göstermek için sıralanmış kenarlar olsun. Eğer $i < j$ ise, bu bir ileri kenardır, diğer durumda geri kenardır.

Doğrusal sıralama (linear order) $Lo$, $G$'de bulunan tüm düğümlerin şu 3 kurala göre taranmasıyla elde edilir.

1. Eğer $i_1 = i_2 \; ve \; j_1 < j_2$ ise, $e_1 < e_2$
2. Eğer $i_1 < j_1 \; ve \; j_1 = i_2$ ise, $e_1 < e_2$
3. Eğer $e_1 < e_2 \; ve \; e_2 < e_3$ ise, $e_1 < e_3$.

**Tanım (DFS kodu - DFS code):** Bir $G$ çizgesinin verilmiş DFS kodu $T$ için, düğüm sıralaması $(e_i)$, doğrusal sıralama $Lo$ üzerinden şu şekilde oluşturulursa $(e_i \; Lo \; e_{i+1})$ ve burada $i = 0, \dots, |E| - 1$ ise, $(e_i)$ 'ye DFS kodu denir ve $code(G, T)$ ile gösterilir.

DFS kodunun gösterimi için değişebilen 5 nicelik $(i, j, l_i, l_{(i,j)}, l_j)$ kullanılır. Bunlar:

- $i \; ve \; j$, ilgili kenarın iki ucundaki düğümler,
- $l_i \; ve \; l_j$, kenarların etiketleri,
- $l_{(i,j)}$, düğümün etiketidir.

Şekil 2-11'deki çizgenin DFS kodları Çizelge 2-3'de verilmiştir.

Çizelge 2-3: Şekil 2-11'deki çizgenin DFS kodları

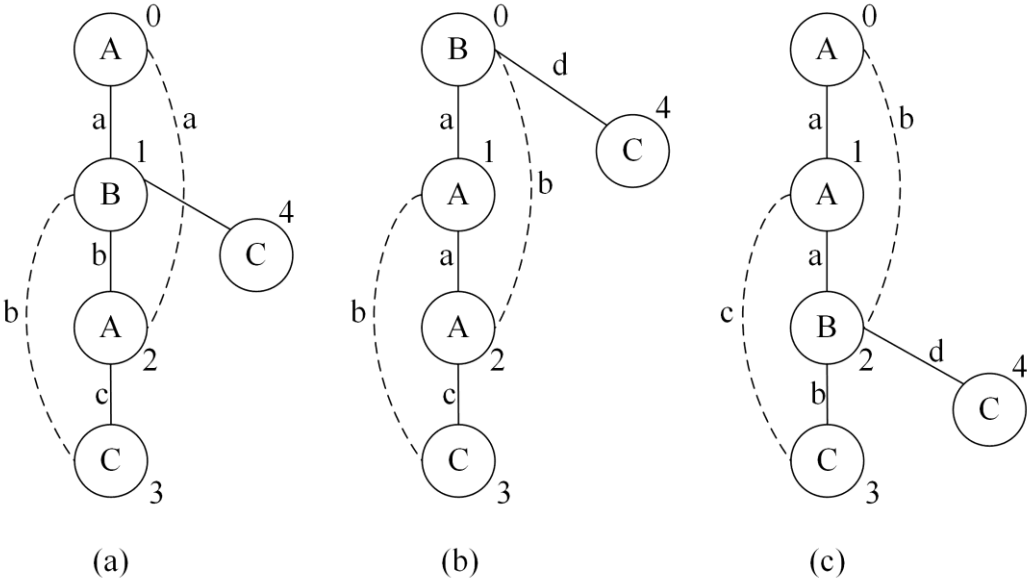
| Başlangıç no | DFS kodu |
|---|---|
| 0 | (0,1,A,a,B) |
| 1 | (1,2,B,b,A) |
| 2 | (2,0,A,a,A) |
| 3 | (2,3,A,c,C) |
| 4 | (3,1,C,b,B) |
| 5 | (1,4,B,d,C) |

Bir çizgenin DFS kodları oluşturulurken, başlangıç yapılan düğümün değişmesi durumunda farklı DFS kodları oluşmaktadır. Tüm bu DFS kodlarının birleşimi ise DFS sözlük sıralamasını oluşturur.

**Tanım (DFS sözlük sıralaması - DFS lexicographic order):** $Z$, bağlı durumdaki etiketlenmiş çizgelerin olası tüm DFS kod listesini içermek üzere, $Z = \{code(G,T)|T\ \ G'nin\ DFS\ ağacıdır\}$ olarak ifade edilir. DFS sözlük sıralaması ise bir doğrusal sıralamadır ve şu şekilde ifade edilir. $\alpha = code(G_\alpha, T_\alpha) = (a_0, a_1, \dots, a_m)$ ve $\beta = code(G_\beta, T_\beta) = (b_0, b_1, \dots, b_n)$, $\alpha, \beta \in Z$ ve $\alpha \leq \beta$ iken şu şartlardan biri doğru olmalıdır.

1. $k < t, a_t\ Lo\ b_t\ için, \exists t, 0 \leq t \leq \min(m,n)\ , a_k = b_k$
2. $0 \leq k \leq m$ ve $n \geq m$ için $a_k = b_k$ .

Şekil 2-12'te, Şekil 2-11'de bulunan çizgenin 3 farklı DFS kod ağacı görünmektedir.




Şekil 2-12: Bir çizgenin üç farklı DFS kod ağacı

DFS sözlük sıralaması sonucu oluşan DFS kodlarından eşleniklik kontrolü yapılması için en küçük DFS kodunun (minimum DFS code) alınması gereklidir.

**Tanım (En küçük DFS kodu – Minimum DFS code):** Bir $G$ çizgesi için, DFS sözlük sıralamasına göre oluşturulan $Z(G) = \{code(G,T)|T\ \ G'nin\ DFS\ ağacıdır\}$

içerisindeki en küçük DFS koduna en küçük DFS kodu denir ve $\min(Z(G))$ olarak ifade edilir. Aynı zamanda $G$'nin kabul görmüş etiketidir.

**Teori:** Eğer verilen $G$ ve $G'$ çizgelerinin en küçük DFS kodları eşit ise, $\min(G) = \min(G')$'dir ve bu durumda çizgeler eşleniktir.

Yukarıdaki tanımlar ve teorinin birleşimi sonucu, alt çizge madenciliği problemi en küçük DFS kodlarının bulunup karşılaştırılması esasına indirgenebilir. gSpan algoritmasınında temelde Şekil 2-10'daki strateji üzerinde çalışmasının nedeni budur. Şekil 2-13'de, Şekil 2-12'deki çizgelerin DFS kodları mevcuttur. Bu kodlar onlarca olabilir. Ancak örnek olması açısından üç tanesi burada verilmiştir. Bu kodlar içerisinde en küçük DFS kodunu bulurken yapılması gereken, DFS kodları arasında birinci sıradan başlayarak sözlük sıralamasına göre karşılaştırma yapılmasıdır. Eğer ilk seviyede karşılaştırma bir sözlük sıralamasını sağlarsa, en küçük DFS kodu bulunmuş demektir. Bulunamaz ise, diğer seviyelerde karşılaştırmalar en küçük olanı bulana dek devam eder.

| **Başlangıç no** | **DFS kodu (a)** | **DFS kodu (b)** | **DFS kodu (c)** |
|---|---|---|---|
| 0 | (0,1,A,a,B) | (0,1,B,a,A) | (0,1,A,a,A) |
| 1 | (1,2,B,b,A) | (1,2,A,a,A) | (1,2,A,a,b) |
| 2 | (2,0,A,a,A) | (2,0,A,b,B) | (2,0,B,b,A) |
| 3 | (2,3,A,c,C) | (2,3,A,c,C) | (2,3,B,b,C) |
| 4 | (3,1,C,b,B) | (3,1,C,b,A) | (3,0,C,c,A) |
| 5 | (1,4,B,d,C) | (0,4,B,d,C) | (2,4,B,d,C) |

Şekil 2-13: Şekil 2-12'deki çizgelerin DFS kodları

En küçük DFS kodları için 0'ıncı düğümden itibaren başlarsak ve üç kodu A, B ve C karşılaştırırsak, sıralamanın $C < A < B$ şeklinde olduğu görülmektedir. Dolayısıyla C kodu, en küçük DFS kodu olarak bu örnekte ortaya çıkmaktadır.

**Tanım (DFS kodlarında üst alt ilişkisi – DFS parent child relation):** Verilen iki DFS kodu $\alpha = (a_0, a_1, \dots, a_m)$ ve $\beta = (a_0, a_1, \dots, a_m, b)$ arasındaki ilişki, $\alpha$ $\beta$'nın üstü yada $\beta$ $\alpha$'nın altı olarak ifade edilir.

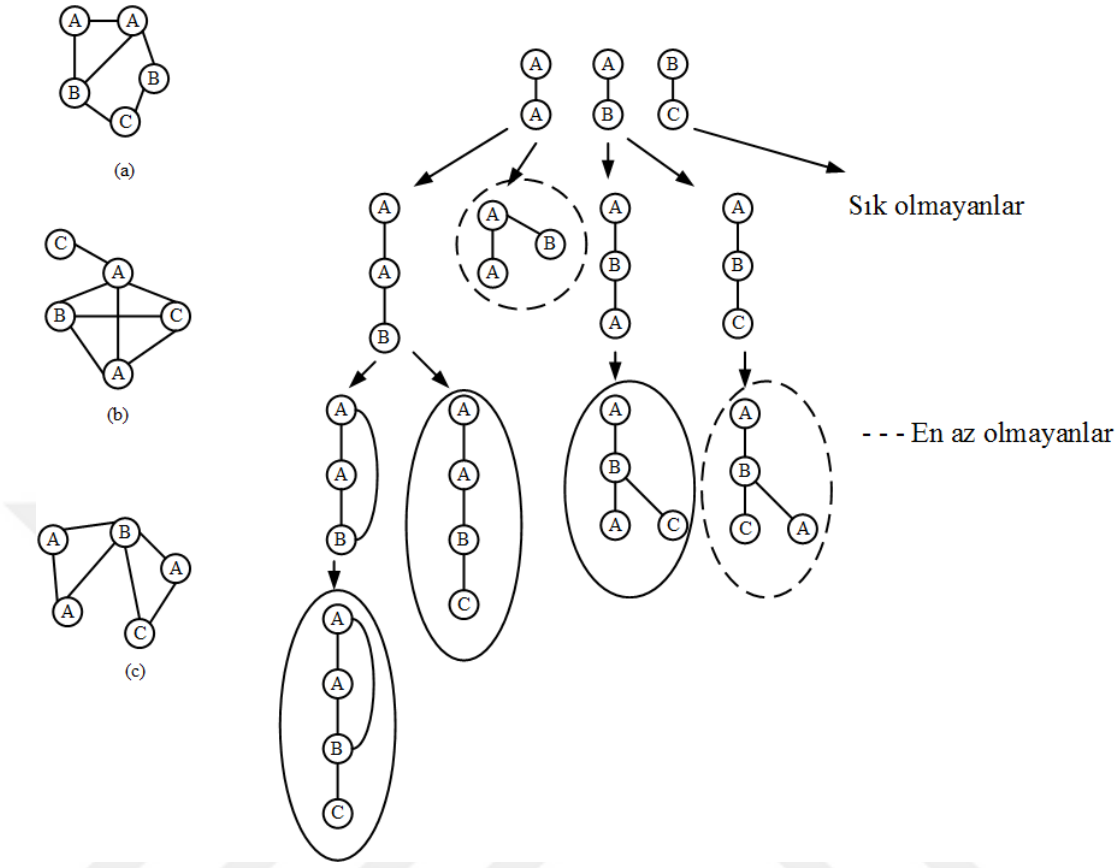
Çizgelerin en küçük DFS kodlarının belirlenmesi sonucu, DFS ağaçları alt üst ilişkisi kapsamında birleştirilerek DFS kod ağacı oluşturulur.

**Tanım (DFS kod ağacı – DFS code tree):** Her bir kenar bir DFS kodunu temsil etmek üzere ve kodlar arasında üst-alt ilişkisi olması koşuluyla sözlük sıralamasına bağlı kalınarak oluşturulan ağaç yapısına DFS kod ağacı denir. [39]

DFS kod ağaçları oluşturulurken boyutunun gereksiz uzunluklara ulaşmasını engellemek amacıyla, oluşturulmaları sırasında destek seviyesinin düşük olduğu belli olan çizgeler ve benzeri zaten var olan yapılar temizlenir (selection ve pruining). Oluşturulan DFS kod ağacında her seviye (n), (n-1) adet düğüm içermektedir.

Şekil 2-14'de, verilen üç örnek çizge için destek seviye 3 olarak belirlenmiş alt çizgelerin bulunması işlemi gösterilmektedir. Burada çizgelerin minimum kodlarının tesbiti sonucu başlangıç düğümleri kalın çizgili daire ile gösterilmiştir.

gSpan algoritması iki method içermektedir. Birinci method, DFS sözlük sıralamasına göre tek düğümlü kodları oluşturur ve ikinci methodu çağırır. İkinci method ise oluşturulan çizgeleri büyüterek diğer çizgelerle karşılaştırır ve birinci methodu özyinelemeli olarak çağırır. Şekil 2-15'de gSpan algoritmasının temsili kodları verilmiştir.




Şekil 2-14: DFS kod ağacı üzerinde destek seviyesi 3 olan alt çizgeler

```
Metod 1: GraphSet_projection(GS, FS)
    sort labels of the vertices and edges in GS by frequency;
    remove infrequent vertices and edges;
    relabel the remaining vertices and edges (descending);
    S^1:= all frequent 1-edge graphs;
    sort S^1 in DFS lexicographic order;
    FS := S^1;
    for each edge e in S^1 do
        init g with e, set g.GS by graphs which contains e.
        Subgraph_mining(GS, FS, g);
        GS := GS - e;
        if |GS| < minSup
          break;
Metod 2: Subgraph_mining(GS, FS, g)
    if g ≠ min(g)
        return;
    FS := FS ∪ {g};
    enumerate g in each graph in GS and count g's children;
    for each c (child of g') do
        if support(c) ≥ minSup
          Subgraph_mining(GS, FS, c);
```

Şekil 2-15: gSpan algoritmasının temsili kodu

## 2.5 Veri Yapısı

Verilerin bilgisayar sistemlerinde tutulmasını ve düzenlenmesini sağlayan veri yapılarını optimize edebilmek için bellekte nasıl tutulduklarını anlamak, hizalama ve boşluk doldurma gibi özelliklerini bilmek gereklidir.

### 2.5.1 Hizalama Nedir

Bilgisayar belleğinde verinin düzenlenme ve saklanma şekline veri yapısı hizalaması (data structure alignment) denir. Veri yapısı hizalama, iki özelliğe bağlı olarak yapılır. Bunlar veri hizalama (data alignment) ve veri yapısı boşluk eklemedir

(data structure padding). Bu iki özellik ile yapılar paketlenerek (structure packing) bellekte tutulurlar.

Günümüz bilgisayar sistemlerinde paketlenmiş veri yapılarının bellekten okunma ve yazılması sözcük uzunluğu yığın miktarı (word size chunk) kadar olmaktadır. Verilerin bu şekilde bellekte saklanmasının amacı, işlemcinin (CPU) bellek alanına hızlı ulaşımını sağlamak dolayısıyla da performans artışı yapabilmektedir.

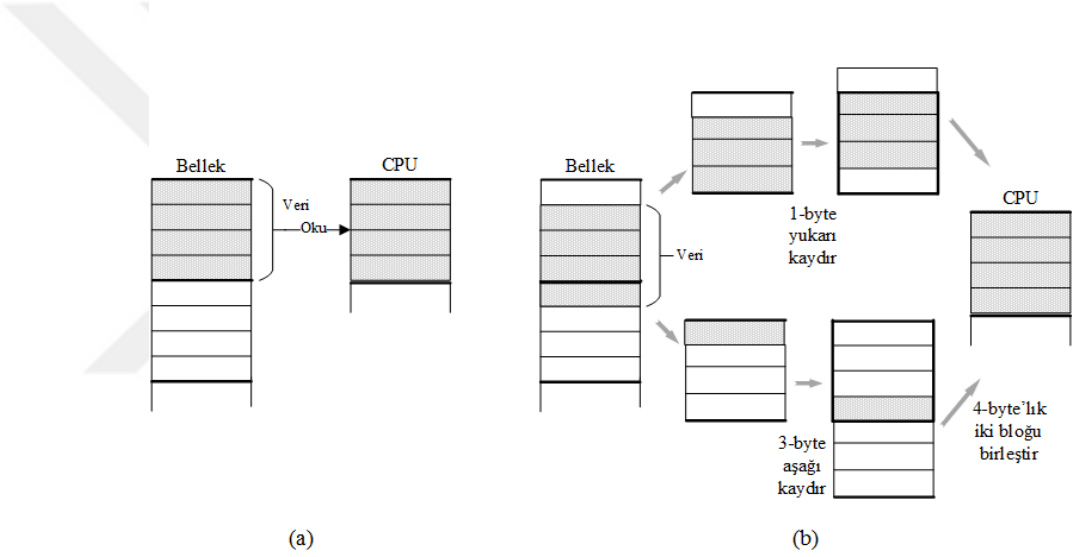
Bilgisayar mimarisine göre sözcük uzunlukları değişkenlik göstermektedir. Örneğin, x32 tabanlı bir bilgisayarda sözcük uzunluğu 4 byte iken, x64 tabanlı bir bilgisayarda sözcük uzunluğu 8 byte'dır [46].

### 2.5.2 Boşluk Doldurma

Bilgisayar sistemlerinde sözcük uzunluk miktarı işletim sistemi tarafından varsayılan değerde (default) verilmektedir. Derleyicilerde bu varsayılan değerler üzerinden kodları bilgisayar diline (executable) çevirmektedir. Bir ya da birden fazla veri belleğe ardışık olarak yazılacağı ya da okunacağı zaman işlemler bir sözcük uzunluğu miktarında yapılmaktadır. Dolayısıyla eğer veri sözcük uzunluğu miktarından küçük ya da büyük ise, sözcük yığını miktarının katı şeklinde olması için eksik kısımlarına boşluk alanlar eklenir ve belleğe yazılır. Böylece bir verinin ardından başlayan yeni bir verinin bitiminin, aynı sözcük yığını içerisinde olması yada yeni bir sözcük yığını içerisinde olması sağlanır. Bu amaçla boşluk alanların verilerin ardına eklemesi işlemine veri yapısı doldurma (data structure padding) denir.

### 2.5.3 Hizalama Hatası

Eğer bir veri bir sözcük uzunluğu boyutundan fazla ise ve bir sonraki sözcük alanına taşıyorsa bu durumda işlemci en az iki kere bellek alanına ulaşmak durumunda kalacaktır. Bu duruma hizalama hatası (alignment fault) denir. Şekil 2-16'de, 4 byte'lık veriye veri hizalaması doğru yapılmış (a) ve veri hizalama hatası olan (b) sistemlerde CPU'nun nasıl eriştiği görülmektedir [47].




Şekil 2-16: Doğru hizalanmış ve hizalama hatası olan 4 byte'lık veriye erişim [47]

### 2.5.4 Hizalama Nasıl Yapılır

Veri hizalamanın nasıl yapıldığını anlamak için, öncelikle temel veri tiplerinin (primitive data types) bellek ihtiyaçlarını inceleyelim. Çizelge 2-4'de temel veri tiplerinin 32 ve 64 bitlik sistemlerdeki veri hizalama ihtiyaçları gösterilmektedir [48].

Çizelge 2-4: Temel veri tipleri ve veri yığın hizalama ihtiyaçları

| **Veri tipi** | **Hizalama (32 bit)** | **Hizalama (64 bit)** |
|---|---|---|
| char | 1 byte | 1 byte |
| short | 2 byte | 2 byte |
| int | 4 byte | 4 byte |
| long | 4 byte | 8 byte |
| float | 4 byte | 4 byte |
| double | 4 byte | 8 byte |
| long long | 8 byte | 8 byte |
| pointer | 4 byte | 8 byte |

**Örnek:** Şekil 2-17'deki veri yapısının 32 bitlik bir sistemde nasıl hizalanacağını inceleyelim.

```
struct {
    int int1; //4 byte
    char c1; //1 byte
    char *p; //4 byte yada 8 byte
    char c2; //1 byte
    short s1; //2 byte
} OrnekYapı;
```

Şekil 2-17: Örnek bir veri yapısı

32 bitlik sistemler varsayılan değer olarak 4 byte'lık sözcük uzunluğunda veri hizalaması kullanırlar. OrnekYapı içerisindeki 5 adet veri yapısının bellekte hizalanması için 4 byte'lık kısıma sığacak şekilde olmaları gerekmektedir. Sığmayan kısımlar hizalama hatası olmaması için boşluk eklenerek bir sonraki sözcük yığını içerisinde olacak şekile çevirilirler. Şekil 2-18'te veri yapısının hizalanmış hali ve hangi sözcük uzunluğu birimi içerisine konulduğu gösterilmektedir.

Eğer, OrnekVeri yapısı *sizeof(OrnekVeri)* komutu ile kontrol edilirse, *12 byte* olarak görünecektir. Fakat gerçekte bellekte kapladığı alanın $4x4 = 16\ byte$ olacaktır.

| Yapı | Sıra |
|---|---|
| **struct** { | |
| **int** int1; //4 byte | 1 |
| **char** c1; //1 byte | 2 |
| **char** pad[3]; //3 byte hizalama boşluğu | 2 |
| **char** *p; //4 byte | 3 |
| **char** c2; //1 byte | 4 |
| **short** s1; //2 byte | 4 |
| **char** pad[1]; //1 byte hizalama boşluğu | 4 |
| } OrnekYapı; | |

Şekil 2-18: Şekil 2-17'deki veri yapısının 4 byte sözcük uzunluklu hizalanışı

Aynı örneği 64 bitlik bir sistemde hizalayacak olursak, bu durumda sözcük uzunluğu 8 byte olacağından Şekil 2-19'daki şekilde olacaktır.

| Yapı | Sıra |
|---|---|
| **struct** { | |
| **int** int1; //4 byte | 1 |
| **char** c1; //1 byte | 1 |
| **char** pad[3]; //3 byte hizalama boşluğu | 1 |
| **char** *p; //8 byte | 2 |
| **char** c2; //1 byte | 3 |
| **short** s1; //2 byte | 3 |
| **char** pad[5]; //1 byte hizalama boşluğu | 3 |
| } OrnekYapı; | |

Şekil 2-19: Şekil 2-17'teki veri yapısının 8 byte sözcük uzunluklu hizalanışı

Şekil 2-19'da görüldüğü üzere aynı yapı 8 byte sözcük uzunluklu veri hizalama yapıldığında 3 sözcük boyutu kadar yer kaplayacaktır. Dolayısıyla bellekte saklanması içinde $3x8 = 24\ byte$ alana ihtiyaç duyulacaktır.

OrnekVeri isimli bu verinin 64 bit sistemdeki kapladığı bellek alanı 32 bit sistemde kapladığı bellek alanından $(1 - \frac{24}{16}) = 15\%$ daha fazladır.

### 2.5.5 Hizalama Optimizasyonu

Veri hizalama CPU'nun erişimi için olumlu etki yapmakla birlikte verilerin bellekte tutulması için gerekli alan ihtiyaçlarını arttırmaktadır. Bu nedenle veri yapıları hizalamanın verimliliğini arttırmak için uygulanabilecek tekniklerin gerekliliği doğmuştur [49].

Veri hizalamanın verimliliğini arttırabilmek ve bellek kazancı sağlayabilmek için uygulanabilecek iki method vardır [50][51].

1. Veri sıralamanın değiştirilmesi (structure reordering)
2. Veri hizalamada kullanılan yığın boyutunun değiştirilmesi (word size chunk)

Veri sıralamasının değiştirilmesi, verilerin en büyük hizalama ihtiyacı olandan daha az hizalama ihtiyacı olana doğru sıralanması ile olmaktadır. Şekil 2-20'de, yeniden hizalama ile Şekil 2-17'deki OrnekVeri yapısının 4 byte ve 8 byte'lık sözcük sıralama uzunluklu sistemlerdeki bellek ihtiyaçlarının nasıl değiştiği görülmektedir.

Veri hizalamada verimlilik sağlayan sözcük uzunluk yığın değeri değiştirilmesinde sözcük uzunlukları 1, 2, 4, 8 ve 16 olarak belirlenebilmektedir. Örneğin veri hizalama için sözcük uzunluğunun 2 byte olduğunu varsayıp OrnekVeri isimli yapıyı yeniden sıralama yapılmamış şekilde 32 bitlik bir sistem için hizalarsak Şekil 2-21'daki şekilde olacaktır.

Yapı paketlemede kullanılan sözcük boyutu yüksek seviye programlama dillerinde Java, C# vb. değiştirilemez. Ancak düşük seviye programlama dillerinde C, C++, Fortran vb. özel derleyici direktifleri ile mümkün olmaktadır. Bir diğer önemli nokta ise sözcük boyutunun azaltılması ile CPU'nun bellek erişim miktarı adet olarak arttığından programların çalışma zamanı performansları negatif olarak etkilenecektir. Bu nedenle sözcük boyutunu değiştirmek konusunda dikkatli olunması gerekmektedir.

| Yapı | Sıra (4 byte) | Sıra (8 byte) |
|---|---|---|
| **struct** {<br>  **char** *p; //4 yada 8 byte<br>  **int** int1; //4 byte<br>  **short** s1; //2 byte<br>  **char** c1; //1 byte<br>  **char** c2; //1 byte<br>} OrnekYapı; | <br>1<br>2<br>3<br>3<br>3 | <br>1<br>2<br>2<br>2<br>2 |
| Toplam bellek ihtiyacı → | 12 $byte$ | 16 $byte$ |

Şekil 2-20: Şekil 2-17'deki veri yapısının tekrar hizalanışı

| Yapı | Sıra |
|---|---|
| **struct** {<br>  **int** int1; //4 bytes<br>  **char** c1; //1 byte<br>  **char** pad[1]; //1 byte hizalama boşluğu<br>  **char** *p; //4 bytes<br>  **char** c2; //1 bytes<br>  **char** pad[5]; //1 byte hizalama boşluğu<br>  **short** s1; //2 bytes<br>} OrnekYapı; | <br>1,2<br>3<br>3<br>4,5<br>6<br>6<br>7 |
| Toplam bellek ihtiyacı → | 14 $byte$ |

Şekil 2-21: Şekil 2-17'deki veri yapısının 2 byte sözcük uzunluklu hizalanışı

### 2.5.6 Yapı Paketleme Direktifleri

C ve C++ programlama dilleri esnek ve güçlü yapılarıyla sayesinde veri yapısı paketleme için kullanılan hizalama miktarının değiştirilmesine izin vermektedir. Bu değişiklik derleme sırasında (compile time) olmakta ve çalıştırma (runtime) sırasında önceden belirlenmiş değerlere göre yapılabilmektedir.

Bu işlem için kullanılan direktif *#pragma pack()* 'dir. Sözcük uzunluğunda yapılacak değişiklik yapının başlangıcında *#pragma pack(n)* direktifi ile belirlenir ve sözcük uzunluğu geçici olarak değiştirilir. Yapının bitimi ardından varsayılan sözcük uzunluğuna dönülmesi için *#pragma pack()* yazılır. Şekil 2-22'de sözcük uzunluğu 2 byte'a geçici olarak sınırlandırılmaktadır. Artık OrnekYapı2 isimli yapı 32 bit ya da

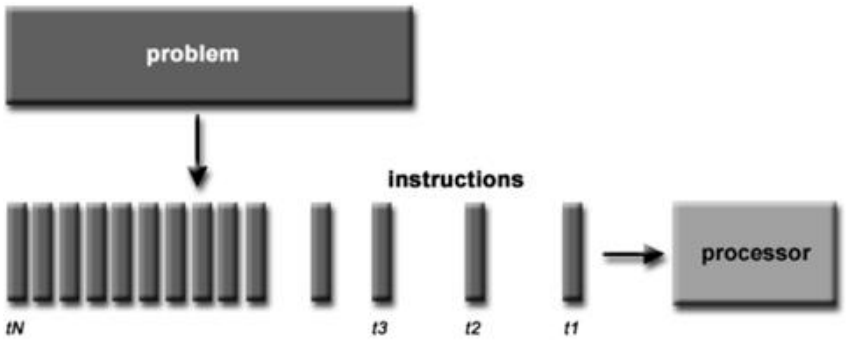
64 bit sistemlerdede kullanılsa sözcük uzunluğu 2 byte üzerinden hesaplanarak veri hizalaması yapılacak ve her durumda Şekil 2-20'deki gibi bellekte 12 byte yer kaplayacaktır [52].

```
#pragma pack(2)
struct {
      int int1; //4 byte
      char c1; //1 byte
      char *p; //4 byte
      char c2; //1 byte
      short s1; //2 byte
} OrnekYapı2;
#pragma pack()
```

Şekil 2-22: Yapı paketleme boyunu 2 byte'a değiştirilmiş bir veri yapısı

## 2.6 Paralelleştirme

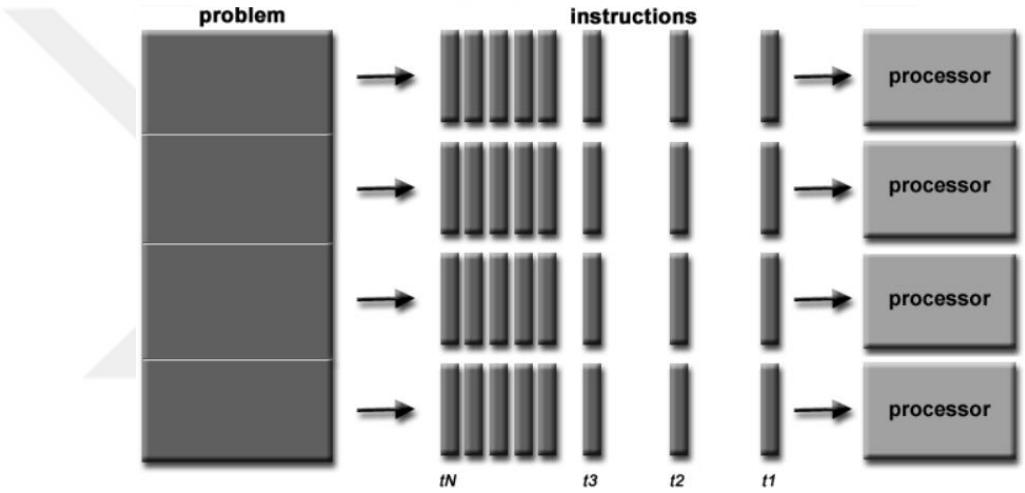
Geleneksel yazılımlar seri (serial) bir yapıda tek bir iş parçacığı üzerinden programın çalışma süreci boyunca çalışıp sonlanırlar. Var olan bir problem birden fazla adım seti olarak belirlenir ve sırası gelen adım iş parçacığı aracılığı ile çalıştırılıp sonlandırılır. Şekil 2-23'da örnek bir durum gösterilmektedir.




Şekil 2-23: Tek iş parçacıklı sistemde bir programın çalıştırılması [39]

Program parçası içerisindeki birbirinden bağımsız çalışabilen işlemlerin farklı iş parçacıkları ile çalışabilir hale getirilmesi ile birden fazla işlemci üzerinde aynı anda işlem yapılabilmesi durumuna uygulamanın paralelleştirilmesi (parallel programming) denir. Şekil 2-24'de örnek bir paralelleştirilmiş programın çalışması gösterilmektedir.

Parallelleştirilmiş uygulamalarda her bir iş parçacığı bir işlemci ya da çekirdekte çalıştırılmaktadır. Bazı durumlarda verilerin çoklanması gerekebilmektedir. Bu nedenle değişik paralelleştirme methodları mevcuttur.




Şekil 2-24: Paralelleştirilmiş bir programın çalıştırılması [39]

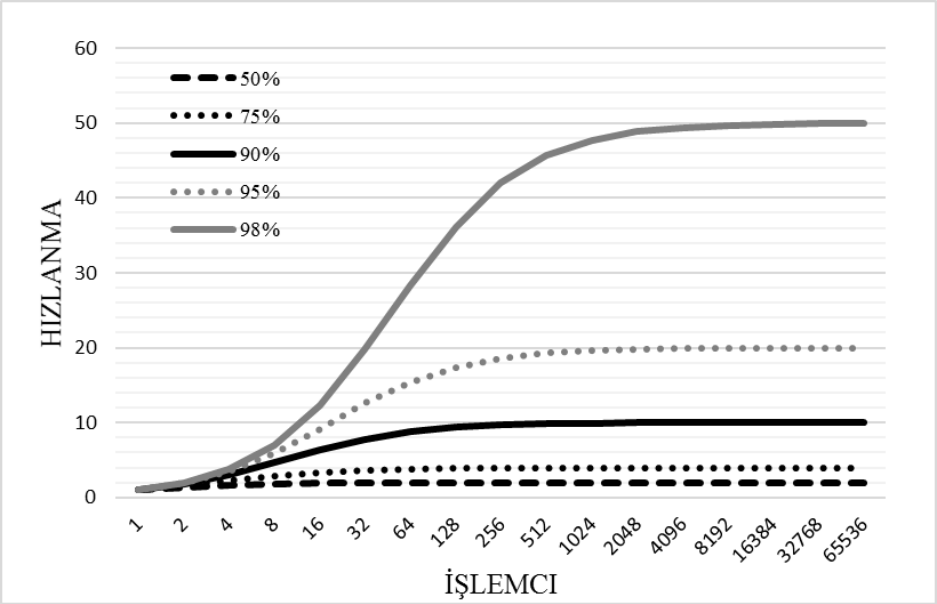
### 2.6.1 Amdahl Kanunu

Bir programın paralelleştirilmiş halinin hızlanması, Amdahls kuralı (Amdahl's Law) ile belirlenebilir. Bu kurala göre bir programın maksimum hızlanması seri çalışan kısmına F, kullanılan işlemci sayısına *n* demek koşuluyla Denklem 2.2'deki şekilde hesaplanabilir.

$$Hızlanma_{max} = \frac{1}{F + \frac{(1-F)}{n}} \tag{2.2}$$

Bir programın en yüksek hızlanma miktarı, paralelleştirilmiş kısmı ve kullanılan işlemci sayısına bağlı olarak Şekil 2-25'deki gibi olacaktır.

Grafikte görüldüğü üzere, programın paralelleştirilebilen kısmı ne kadar yüksek olursa olsun belli bir hızlanma oranından sonra işlemci sayısının artması hızlanmada neredeyse yok denecek kadar az etkilidir.




Şekil 2-25: Paralelleşme miktarına bağlı en yüksek hızlanma eğrisi.

Paralelleştirilmiş bir programın etkinliği Denklem 2.3'deki şekilde, paralelleştirmede elde edilen gerçek hızlanmanın kullanılan işlemci sayısına bölümü ile bulunur. Etkinlik oranı 1'e yaklaştıkça paralelleştirme etkinliği yükselir.

$$0 < Etkinlik = \frac{Hızlanma}{n} < 1 \qquad (2.3)$$

### 2.6.2 Paralelleştirme Methodları

Paralelleştirmede günümüzde kullanılan standart tekniklere bakıldığında üç method öne çıkmaktadır.

1. MPI: Bir ya da birden çok bilgisayarın ağ üzerinden birbirlerine bağlanarak oluşturduğu bilgisayar kümeleri üzerinde kullanılan, temeli mesaj iletimine dayanan paralelleştirme methodudur. Bellek paylaşımlı yapılardada çalışabilmektedir ancak yapısı her programın birden fazla şekilde çalıştırılmasına dayandığından bilgisayar kümeleri üzerinde kullanımı tercih edilmektedir. Çoklu platform desteği derleyicilerin desteğine bağlı olarak mevcuttur [54].
2. POSIX iş parçacıkları: C, C++, Fortran gibi dillerde kullanılan en temel iş parçacıklı paralelleştirme methodudur. Bellek paylaşımlı sistemlerde kullanılmaktadır. Çoklu platform desteği derleyicilerin desteğine bağlı olarak mevcuttur. İş parçacıklarının oluşturulması, eşlenmesi, durumlarının kontrolleri ve sonlandırılmaları gibi tüm yaşam döngü kontrollerinin geliştirici tarafından yapılması gereklidir [55].
3. OpenMP: POSIX thread gibi bellek paylaşımlı sistemlerde kullanılan ve son yıllarda kullanımı hızla artan bir teknolojidir. Geliştiricinin POSIX iş parçacıklı sistemde yapmak durumunda olduğu bazı iş parçacığı yönetimlerini kendisi yapmakta ve program yazabilmeyi kolaylaştırmaktadır. Bu tez çerçevesinde geliştirilen ANPTR methodu OpenMP teknolojisini kullandığından, teknolojinin bu çalışma için gerekli olan kısımları Bölüm 2.6.4'te detaylandırılmıştır [56].

### 2.6.3 Verimlilik Ölçütleri

Bir programın paralelleştirilmesi sırasında aşağıdaki ölçütlerin incelenmesi ve ilgili çözümlerin uygulanması, paralelleştirme veriminin arttırılması için gereklidir.

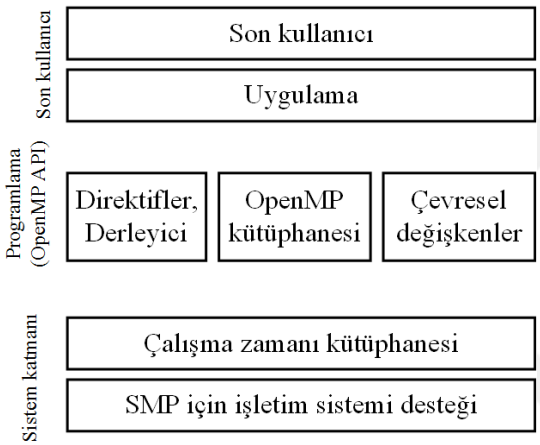
- İş parçacığı yönetimi için hangi teknolojinin kullanılacağı,
- İş parçacığı sayısının kullanılan teknolojiye bağlı olarak otomatik mi yoksa manuel olarak program geliştirici tarafından mı yapılacağı,
- İş parçacıklarının kullandığı veri setinin ortak mı yoksa ayrı mı olacağı,
- Problem alt işlemlerinin belirlenmesi ve bu işlemlerin birbirlerinden mümkün olduğunca ayrık şekilde oluşturulması,
- İş parçacıkları arası eşlenim için eş zamanlı ya da zaman uyumsuz eşlenim modellerinden hangisinin kullanılacağı,
- İş parçacıkları arası iletişim dağılımlarında noktadan noktaya (point-to-point) ya da kollektif (collective) iletişim türünün mü seçileceği,
- İş parçacıkları arasındaki eşleme için bariyer (barrier) ve kilit yapıların (lock/semaphore) minimum seviyede ve doğru kullanımları,
- İş parçacıkları arasında yük dağılımının nasıl sağlanacağı,
- Paralelleştirme sırasında oluşan problemlerin tesbiti için uygulamanın nasıl analiz (debugging) edileceği,
- Paralelleştirilmiş uygulamanın hedef ölçüm ölçütlerinin belirlenmesi ve bu ölçümlerin nasıl yapılacağı.

### 2.6.4 OpenMP Teknolojisi

OpenMP, derleyici direktifleri yardımıyla çok iş parçacıklı programlar yazmayı sağlayan programcı kütüphaneleri bütünüdür. Herhangi bir donanım ortamında derleyici desteğinin olması çalışması için yeterlidir. Bu nedenle çapraz platformlarda

çalışabilmektedir. C, C++ ve Fortran dilleri için desteklenmektedir. OpenMP'nin mimari katman yapısı Şekil 2-26'daki gibidir.

OpenMP kullanabilmek için, *#include<omp.h>* ifadesi ile gerekli kütüphane programa dahil edilir. Parallelleşecek program parçacığı içerisine Şekil 2-27'deki direktif eklenerek paralelleştirilme sağlanır [57].




Şekil 2-26: OpenMP mimari katmanları

```
#pragma omp parallel [clause ...]  newline
                    if (scalar_expression)
                    private (list)
                    shared (list)
                    default (shared | none)
                    firstprivate (list)
                    reduction (operator: list)
                    copyin (list)
                    num_threads (integer-expression)
structured block
```

Şekil 2-27: OpenMP #parallel direktifi

Paralel kısım direktifinde kullanılan önemli argümanları sırasıyla inceleyelim.

**Private:** Liste olarak verilen değişkenler, yalnızca bu paralel kısım içerisinden erişilebilir durumdadır ve iş parçacıkları arasında paylaşılmayacaktır. Değişkenlerin ilk değeri olmayacaktır.

**Shared:** Liste olarak verilen değişkenler, bu paralel kısım içerisindeki tüm iş parçacıkları arasında paylaşılacak aynı zamanda paralel kısım dışarısındanda erişilebilir olacaktır.

**Firstprivate:** Liste olarak verilen değişkenler, bu paralel kısım içerisine girilirken ilk değerleri atanacak ve bundan sonra yalnızca paralel kısım içerisinden erişilebilir olacaktır.

**Num_threads:** Paralel kısımın çalışacağı iş parçacığı sayısı belirtilmektedir. Eğer belirtilmezse, OpenMP uygun iş parçacığı sayısını dinamik olarak kendi belirlemektedir.

Paralel kısımdan önce *omp_set_num_threads(n)* komutu ile belirtilen *n* değeri kadar iş parçacığı oluşturulması sağlanabilir. Paralel kısım içerisinde *omp_get_num_threads()* komutu ile kullanılan toplam iş parçacığı sayısı alınabilir ya da o an için program kısmının çalıştığı aktif iş parçacığı numarası *omp_get_thread_num()* komutu ile öğrenilebilir. Şekil 2-28'de paralelleştirilmiş bir program parçacığı görünmektedir.

OpenMP'de işlerin iş parçacıklarına dağıtılması için kullanılan iş paylaşım yapıları (work-sharing construct) vardır. Bunlar do/for loop, sections, tasks ve single/master'dır. ANPTR çalışması kapsamında kullanılan Do/for loop yapısının nasıl çalıştığını inceleyecelim.

**Do/For loop:** Temeli *for loop* döngüsüne benzemektedir. Programın paralel çalıştırılabilecek kısmındaki işlerin bir döngü içerisinde OpenMP tarafından otomatik olarak belirtilmiş bir zaman çizelgesi (schedule) kapsamında dağıtılmasını sağlar. Direktifin kullanım argümanları Şekil 2-29'de mevcuttur.

```
#include <omp.h>
 main(int argc, char *argv[]) {
       int nthreads, tid, sid;
      sid = 7;
       /* paralel kısım başlıyor, 4 iş parçacığı oluşturulacaktır, tid değişkeni private,
      sid değişkeni public olarak tanımlanıyor */
       #pragma omp parallel private(tid) shared(sid) num_threads(4)
        {
          /* iş parçacığı numarası bulunuyor ve ekrana yazılıyor */
          tid = omp_get_thread_num();
          printf("Merhaba dünya, bu mesaj iş parçacığı = %d 'den geliyor \n", tid);
          printf("sid değişkeninin değeri = %d \n", sid);
          if (tid == 0) { /* yalnızca ana iş parçacığı bu işi yapıyor */
            nthreads = omp_get_num_threads();
            printf("Toplam iş parçacığı sayısı = %d\n", nthreads);
          }
        }
      /* Tüm iş parçacıkları ana iş parçacığına katılıyor, paralel kısım sonlanıyor */
 }
```

Şekil 2-28: OpenMP #parallel direktifi ile örnek

```
#pragma omp for [clause ...]  newline
            schedule (type [,chunk])
            ordered
            private (list)
            firstprivate (list)
            lastprivate (list)
            shared (list)
            reduction (operator: list)
            collapse (n)
            nowait
  for_loop
```

Şekil 2-29: OpenMP # parallel do/for direktifi

Argümanlardan bu tez kapsamında kullanılanları sırasıyla inceleyelim.

**Schedule (type [,chunk]):** Döngü içerisindeki işlerin iş parçacıklarına nasıl dağıtılacağını belirler. Tipine göre çalışma şekilleri şöyledir.

- STATIC, belirtilen *chunk* değeri kadar iş, iş parçacıklarına round-robin mantığıyla dağılır.
- DYNAMIC, belirtilen *chunk* değeri kadar iş, iş parçacıklarındaki iş bittiği durumda dinamik olarak dağıtılır. Varsayılan *chunk* değeri 1'dir.
- GUIDED, belirtilen chunk değeri üzerinden DYNAMIC yapıya benzer şekilde ancak azalan iş parçacığı sayısıyla işler dağıtılır. İlk iş, *toplam_iş/iş_parçacığı_sayısı* ve ardından diğer işlerde *kalan_toplam_iş/iş_parçacığı_sayısı* denklemi doğrultusunda dağıtılırlar.
- RUNTIME, programın çalışması sırasında OMP_SCHEDULE çevresel veri değeri ile dağıtılır.

**Nowait:** Bu ifade kullanılmadığında iş parçacıkları işlerini tamamladıklarında döngüden ayrılmadan internal bir *barrier* varmış gibi beklemede kalırlar ve tüm iş parçacıkları işlerinin bitiminde katılım (join) işlemini gerçekleştirirler. Nowait kullanımı ile iş parçacıkları katılım işlemini beklemeden devam ederler.

**Private, firstprivate, shared**: #parallel ifadesindeki ile aynı mantıkta çalışırlar.

Do/for yapısının kullanımına Şekil 2-30'de örnek bir program parçacığı verilmiştir. Do/for direktifi ile iş dağıtılması OpenMP'ye bırakılmaktadır. Bununla birlikte #parallel direktifi ilede aynı iş dağıtımı daha fazla kullanıcı kontrolü ve kod ile yapılabilir. Şekil 2-31'de seri bir kodun #parallel ve #do/for yapısı kullanılarak paralelleştirilmesine örnek verilmiştir.

```
#include <omp.h>
#define N 1000
#define CHUNKSIZE 100
main(int argc, char *argv[]) {
      int i, chunk;
      float a[N], b[N], c[N];
      /* Temel değerlerin atanması */
      for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
      chunk = CHUNKSIZE;
      #pragma omp parallel shared(a,b,c,chunk) private(i)
      {
        #pragma omp for schedule(dynamic,chunk) nowait
        for (i=0; i < N; i++)
          c[i] = a[i] + b[i];
      }  /* paralel kısım sonu */
}
```

Şekil 2-30: OpenMP #do/for direktifi ile örnek

| Seri kod | for(i=0;I<N;i++) { a[i] = a[i] + b[i];} |
|---|---|
| OpenMP paralel alan kullanımı | **#pragma omp parallel**<br>**{**<br>int id, i, Nthrds, istart, iend;<br>id = omp_get_thread_num();<br>Nthrds = omp_get_num_threads();<br>istart = id * N / Nthrds;<br>iend = (id+1) * N / Nthrds;<br>for(i=istart;I<iend;i++) { a[i] = a[i] + b[i];}<br>**}** |
| OpenMP do/for iş paylaşımı kullanımı | **#pragma omp parallel**<br>**#pragma omp for schedule(static)**<br>for(i=0;I<N;i++) { a[i] = a[i] + b[i];} |

Şekil 2-31: Seri bir kodun iki farklı OpenMP paralelleştirmesi

İş parçacıkları arasında kullanılan *shared* tanımlı değişkenlerin üzerinde yapılacak işlemler sırasında birden fazla iş parçacığının değişiklik yapmasının önüne geçmek için bu tez kapsamında kullanılan iki OpenMP senkronizyon mekanizması vardır.

1. Critical, bir öbek olarak çalışır ve *critical* olarak belirtilen bir öbek içerisine aynı anca yalnızca bir iş parçacığının girmesine izin verir. Diğer iş parçacıkları giriş sırasında beklerler ve sırası gelen öbek yapı içerisine girebilir.
2. Atomic**,** critical ile aynı mantıkta çalışır ancak daha hızlıdır. Yalnızca tek bir işlemi bir değişken için yapmaya olanak sağlar, toplama, çıkarma gibi.

Şekil 2-32'de *critical* ve *atomic* direktiflerinin kullanımına örnek verilmiştir.

| **Critical** | **Atomic** |
|---|---|
| **#include <omp.h>**<br>main(int argc, char *argv[]) {<br>int x = 0;<br>**#pragma omp parallel shared(x)**<br>**{**<br>**#pragma omp critical**<br>**{**<br>x = x + 1;<br>/* Ek işlemler yapılabilir*/<br>**}**<br>**}** /* paralel kısım sonu */<br>} | **#include <omp.h>**<br>main(int argc, char *argv[]) {<br>int x = 0;<br>**#pragma omp parallel shared(x)**<br>**{**<br>**#pragma omp atomic**<br>x = x + 1;<br>**}** /* paralel kısım sonu */<br>} |

Şekil 2-32: Critical ve atomic direktiflerinin kullanımı

OpenMP, paralel kısımların birden çok kere iç içe çalışmasını (nested parallelism) desteklemektedir. Her iç içe çalışmada LOP değeri bir artmaktadır. Paralel çalışan iş parçacığı sayısı olan DOP değeri ise, yeni çalışan paralel kısımdaki iş parçacığı sayısı oranındadır. Dolayısıyla herhangi bir LOP değeri için DOP değeri farklı olabilir.

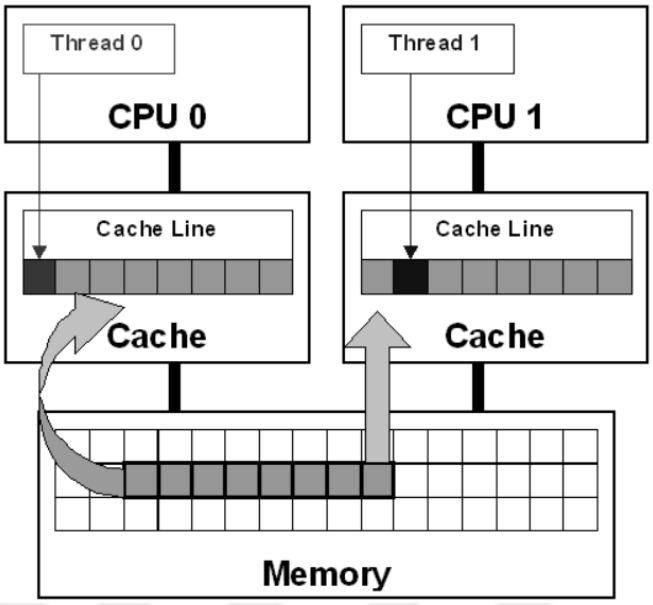
İç içe paralelleşmenin ilgili paralelleşme direktifinden önce aktif hale getirilmesi gereklidir. *omp_set_nested(n)* komutu ile en fazla kaç seviye paralelleşme yani LOP değeri belirlenir. *omp_get_nested()* komutu o anda çalışan iş parçacığının LOP değerini verir. Şekil 2-33'deki örnekte iki seviye LOP yapılmakta ve her seviyede DOP değeri 2 olarak atanmaktadır.

```
#include <omp.h>
#include <stdio.h>
void report_num_threads(int level) {
    #pragma omp single
    {
      printf("LOP %d için, DOP=- %d\n", omp_get_num_threads());
    }
}
int main() {
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(2) /* Paralelleştime */
    {
      report_num_threads(1);
      #pragma omp parallel num_threads(2)  /* Birinci seviye paralelleştime */
      {
        report_num_threads(2);
        #pragma omp parallel num_threads(2) /* İkinci seviye paralelleştime */
        {
          report_num_threads(3);
        }
      }
    }
    return(0);
}
```

Şekil 2-33: İç içe paralelleştirme örneği

OpenMP ile paralelleştirilmiş bir uygulama yazarken performans arttırmaya yönelik aşağıdaki ölçütlere dikkat edilmelidir.

- İş parçacıkları arasındaki paylaşımlı verilerde değişiklik yapılmasını zorunlu tutacak durumların dolayısıyla barrier, critical ve atomic ifadelerinin kullanımlarının azaltılması,
- Paylaşımlı bir verinin birden fazla iş parçacığı tarafından sıklıkla değiştirilmesi gerekiyor ve değişen kısımlar CPU'da aynı ön bellek yapısına sığabiliyorsa, yanlış paylaşım (False sharing, Şekil 2-34'de) adı verilen performans düşürücü problem oluşabilir. Bu durumların tesbit edilmesi ve önüne geçilmesi,




Şekil 2-34: False Sharing'in oluşması [58]

- OpenMP iş parçacığı havuzundaki iş parçacığı miktarının yeterli olmayacağı durumlarda bu havuz değerinin gerekli miktarda yükseltilmesi [59],
- Öbek yapılarda iş parçacıkları arası eşlenimin gerekmediği durumlarda *nowait* argümanının kullanılması [60],
- Zaman çizelgesi tiplerinden STATIC ya da GUIDED'ın yüksek *chunk* miktarı ile kullanımının öncelikle tercih edilmesi, DYNAMIC'in dengeli dağılmamış yüklerin bulunduğu durumlarda kullanılması [61],
- Paralelleştirilmiş öbekleri birleştirmek ve sayısını azaltmak,
- İç içe paralelleştirmelerde en dış seviyelerde iyi bir yük dağılımı ile yüksek DOP kullanımının sağlanması ve LOP değerinin düşük tutulması [62].

## 2.7 Kullanılan Araçlar

### 2.7.1 Kaynak Kodlar, Geliştirme ve Test Ortamı

gSpan algoritmasının kaynak kodu yazarların kendi sitesinden alınmıştır. Orijinal kodlar Linux üzerinde C proglama dili kullanılarak 32 bit olarak geliştirilmiştir.

Geliştirme ortamı olarak, 4 çekirdekli 64 bitlik standart bir masa üstü bilgisayar kullanılmıştır. Test ortamı olarak Microsoft Azure üzerinde A11 kodlu yüksek performans testleri için ayrılmış bir makine kullanılmıştır. Bu makinenin donanım özellikleri, 16 çekirdekli Intel® Xeon® E5-2670 @ 2.6 GHz işlemci ve 112 GB hafızadan oluşmaktadır [40].

### 2.7.2 Yazılım Araçları

Bu tez çalışmasındaki geliştirmeler için öncelikle gSpan algoritma kaynak kodları C++'ın 64 bitlik gcc derleyicisi ile çalışır hale getirilmiştir. Geliştirme ortamında Linux'un Fedora 21 sürümü, kullanıcı arayüz desteğinin kolaylığı nedeniyle tercih edilmiştir. Test ortamlarında ise CentOS kullanılmıştır. Yazılım geliştirme ortamı olarak Eclipse C/C++ Development Tooling (CDT) kullanılmıştır [41][42].

Alan verimliliği çalışmasında, hafıza profilleme (memory profiling) aracı olarak Valgrind araçları kullanılmıştır. Zaman verimliliği çalışması sırasında çoklu iş izleme ve paralel iş parçacıklarının analizi için Intel® VTune™ Amplifier XE 2016 aracı kullanılmıştır [43][44].

### 2.7.3 Veri Setleri

Deneysel çalışmalar sırasında kullanılan veri setleri Çizelge 2-5'da özellikleri ile birlikte verilmiştir. 4, 5 ve 6 nolu veri setleri yapay olarak elde edilmiş veri setleridir. Diğer veri setleri gerçek yaşamdan alınmış verilerden oluşmaktadır. Tablodaki ilk 6 veri seti alan verimliliği çalışmasındaki testlerde diğer 4 veri seti ise zaman verimliliği çalışmasındaki testlerde kullanılmıştır [45].

Çizelge 2-5: Kullanılan deneysel veri setleri ve özellikleri

|G̓|: Veri setinin içerdiği toplam çizge sayısı
$|V_{MAX}(G')|$: Çizgeler içerisindeki en büyük düğüm sayısı
$|E_{MAX}(G')|$: Çizgeler içerisindeki en büyük kenar sayısı
$|L_{MAX\text{-}V}|$: Çizgeler içerisindeki en büyük düğüm etiketi
$|L_{MAX\text{-}E}|$: Çizgeler içerisindeki en büyük kenar etiketi

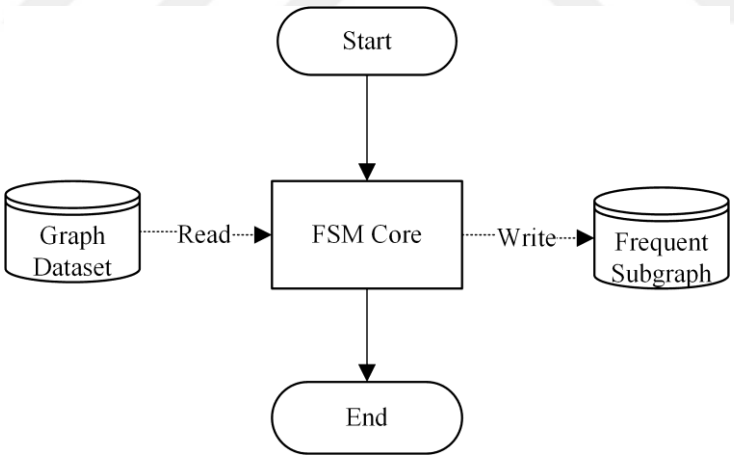
| **No** | **Dataset G̓** | **\|G̓\|** | **$\|V_{MAX}(G')\|$** | **$\|E_{MAX}(G')\|$** | **$\|L_{MAX\text{-}V}\|$** | **$\|L_{MAX\text{-}E}\|$** |
|---|---|---|---|---|---|---|
| 1 | NCI | 20586 | 112 | 119 | 64 | 3 |
| 2 | AIDS | 56213 | 221 | 247 | 61 | 3 |
| 3 | T10KV5KE14K | 10317 | 5747 | 14267 | 88 | 3 |
| 4 | T58KV100E100 | 58242 | 112 | 119 | 63 | 3 |
| 5 | T114KV200E200 | 114455 | 221 | 247 | 63 | 3 |
| 6 | MCF-7 | 25475 | 243 | 245 | 46 | 3 |
| 7 | OVCAR | 38436 | 243 | 245 | 65 | 3 |
| 8 | SN12C | 38048 | 243 | 245 | 65 | 3 |
| 9 | NCI-H23 | 38295 | 243 | 245 | 65 | 3 |

# 3 ALAN VERİMLİLİĞİ ÇALIŞMASI

Bu bölümde SAÇM algoritmalarının alan verimliliğini arttırmaya yönelik geliştirilen PDSSP methodu hakkında bilgi verilmiştir. Methodun alt bileşenlerinin neler olduğu ve nasıl çalıştığı aktarılmış, uygulanması sonucu gSpan algoritmasının çalışma zamanındaki bellek değişimi profilleme üzerinden gösterilmiştir. Son kısımda veri setleri üzerinde yapılan deneysel sonuçlar paylaşılmıştır.

## 3.1 PDSSP Methodu

SAÇM algoritmaları veri setlerini dış bir kaynaktan (disk, ağ vb.) okuyarak belleğe alırlar ve işlemlerini tamamladıktan sonra çıktıyı ekrana ya da dış bir kaynağa yazarlar. Genel mimarileri Şekil 3-1'de gösterildiği gibidir.




Şekil 3-1: SAÇM algoritmalarının genel mimarisi

Giriş veri setleri incelendiğinde sayısallaştırılmış verilerden oluştuğu ve algoritmaların çalışmaları sırasında yalnızca sayısal veriler üzerinden işlem yaptığı tesbit edilmektedir. Sayısal verilerin yapı paketlemesi üzerinde iyileştirme yapılması temeline dayalı olarak geliştirilmiş olan "öngörülü değişken uzunluklu yapı

paketleme" kısaca PDSSP methodu SAÇM algoritmalarının içerisine enjekte edilerek herhangi bir algoritmik değişiklik yapmadan algoritmalarının çalışması sırasında daha az bellek alanına ihtiyaç duymalarını sağlar. Şekil 3-2'de PDSSP'nin bir SAÇM algoritmasına uygulandıktan sonraki mimari yapısı görünmektedir.

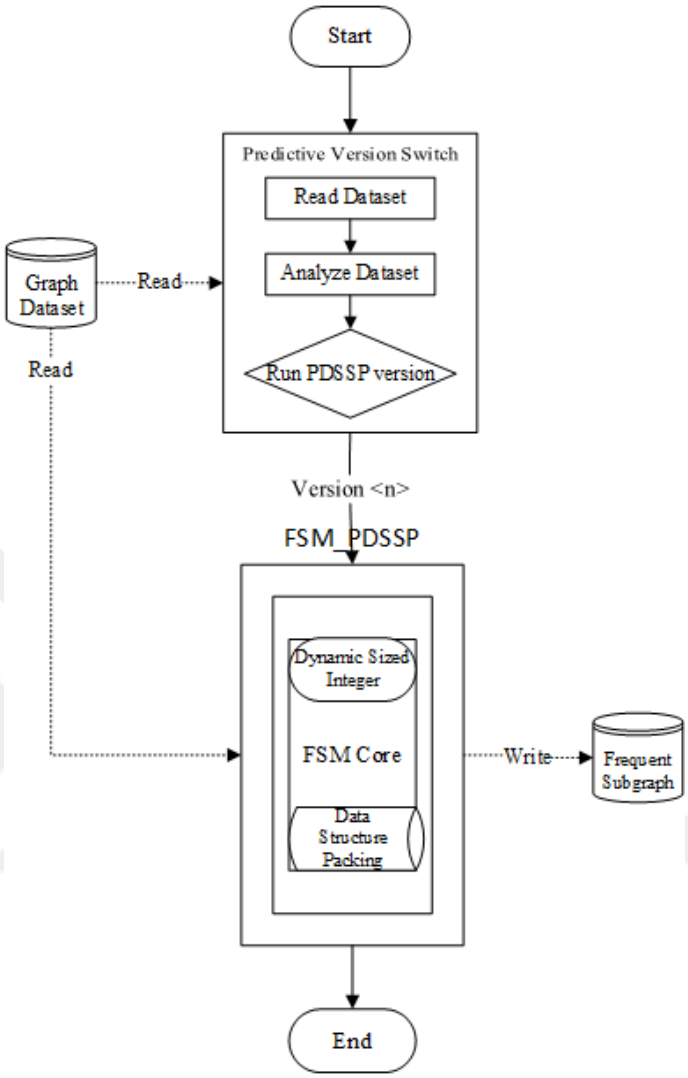
PDSSP'nin çalışması şu temel fikire dayanmaktadır. Eğer bir programın çalışması sırasında ihtiyaç duyabileceği en yüksek sayısal (integer) değerler program çalışmadan önce bilinirse, bu sayısal değerleri tutabilecek en küçük hizalama alanı ihtiyacı olan sayısal veri tipi tanımlanarak veriler bu tipler içerisinde tutulabilir. SAÇM algoritmalarının çalışması sırasında kullandığı sayısal değerlerin işaretsiz (unsigned) sayısal veri tipleri olması nedeniyle, PDSSP methodu işaretsiz sayısal veriler için geliştirilmiştir. Çizelge 3-1'de işaretsiz sayısal veriler için bellek ihtiyaçları ve alabildikleri değer aralıkları gösterilmektedir.

Çizelge 3-1: İşaretsiz sayısal verilerin bellek ihtiyacı ve değer aralıkları

| **Veri tipi** | **Bellek alanı** | **Değer aralığı** |
|---|---|---|
| Unsigned char | 1 byte | 0 - 255 ($2^8$) |
| Unsigned short | 2 byte | 0 - 65,535 ($2^{16}$) |
| Unsigned int | 4 byte | 0 - 4,294,967,295 ($2^{32}$) |
| Unsigned long | 8 byte | 0 - 18,446,744,073,709,551,616 ($2^{64}$) |

PDSSP'yi oluşturan üç ana parça vardır. Bu parçalar;

1. Öngörülü sürüm anahtarı (PVS)
2. İşaretsiz değişken uzunluklu sayısal veri tipi (ds_Int)
3. Veri paketleme tekniğinin uygulanması




Şekil 3-2: PDSSP uygulanmış SAÇM algoritma mimarisi

### 3.1.1 Öngörülü Sürüm Anahtarı (PVS)

PVS, DIMACS biçimindeki (Şekil 3-3) giriş veri setini inceleyerek algoritmanın çalışması sırasında ihtiyaç duyabileceği en yüksek hareket, düğüm, kenar ve etiket sayısal değerlerini tesbit eder. Ardından ilgili FSM_PDSSP sürümünü çalıştırır [53].

```
t # <graph_id>
      v <vertex_id> <vertex_label>
      e <edge_from> <edge_to> <edge_label>
<next_graph_or_end_of_file>
```

Şekil 3-3: DIMACS biçimindeki giriş veri seti

PVS'in hangi sürümdeki SAÇM algoritmasının çalıştırılacağına karar verilebilmesi için SAÇM algoritması çalıştırılmadan önce çalıştırılması gereklidir. PVS algoritmasının temsili kodu Şekil 3-4'de verilmiştir.

```
PVS (A dataset file ds_file, an algorithm selection alg_select, a minimum support level
min_sup, an output file out_file)
Set TransactionCount, maxVertex, maxEdge, maxLabel to zero;
Read graphs from ds_file into Dataset
for each row in Dataset do
  if type of row is transaction then
    Increase TransactionCount by 1
    else if type of row is label and label_ID of row is greater than maxLabel then
      Set maxLabel to label_ID
    else if type of row is vertex and vertex_ID of row is greater than maxVertex then
      Set maxVertex to vertex_ID
    else if type of row is edge and edge_ID of row is greater than maxEdge then
      Set maxEdge to edge_ID
  end if
end for
if alg_select is gSpan then
  Run "gSpan_PDSSP" with ds_file, min_sup, out_file
  else if alg_select  is Gaston then
    Run "Gaston_PDSSP" with ds_file, min_sup, out_file
end if
```

Şekil 3-4: PVS algoritmasının temsili kodu

### 3.1.2 İşaretsiz Değişken Uzunluklu Sayısal Veri Tipi (ds_Int)

İşaretsiz sayısal veri tipleri incelendiğinde (Çizelge 3-2), $2^{16}$, $2^{32}$ ve $2^{64}$ değerleri arasında büyük aralıklar olduğu görülmektedir. Örneğin, $2^{18}$ karşılığındaki sayısal veriyi depolamak için $2^{32}$'lik veri tipinin kullanılması ya da $2^{34}$ karşılığındaki sayısal veriyi depolamak için $2^{64}$'lük veri tipinin kullanılması gerekmektedir. Bu durum ihtiyaç olmayan fazladan alanların boşta kalmasına ve tanımlandıkları içinde bellekte gereksiz yer tutulmasına neden olmaktadır.

Bu probleme çözüm bulmak amacıyla PDSSP çözümünün bir parçası olarak temel işaretsiz sayı tiplerinin desteklemediği ara sayı tiplerinide destekleyebilecek "işaretsiz değişken yapılı sayısal veri tipi" olarak *ds_Int* veri tipi geliştirilmiştir.

*ds_Int* veri tipi $2^8$ karşılığındaki sayısal veri tipinden $2^{64}$'e karşılık gelen işaretsiz sayısal veri tipine kadar birer byte arttırımlı veri tipi depolama imkanı sunan bir yapıdır (structure). Temel çalışma mantığı, gerekli miktarda diziler oluşturarak kendisine verilen sayıları bit kaydırma (bit shifting) işlemi yardımıyla bu diziler içerine ikilik tabanda depolamak şeklindedir. Okumak için bu ikilik tabandaki sayıları tekrar bit kaydırma işlemleri ile birleştirerek ondalık hale getirir. Şekil 3-5'de *ds_Int* algoritmasının temsili kodu gösterilmektedir.

Çizelge 3-2'de *ds_Int* veri tipi ile temel veri tiplerinin depolama alanı ihtiyaçları karşılaştırması ve *ds_Int* kullanımının sağladığı bellek alanı kazancı gösterilmektedir.

```
struct ds_Int (An unsigned integer data InputData, size of ds_Int value byte_size)
  Set StoredData with an empty unsigned char array in byte_size size
  function get () returns integer
    Set OutputData to zero
    for i from 0 to byte_size do
      Set OutputData with OutputData & (i × 8-byte left shifted StoredData[i])
    end for
    return OutputData
  end function
  function set (InputData) returns nothing
    for j from 0 to byte_size do
      Set StoredData[j] to StoredData & (j × 8-byte right shifted InputData)
    end for
  end function
```

Şekil 3-5: ds_Int algoritmasının temsili kodu

Çizelge 3-2: ds_Int veri tipi ve temel veri tiplerinin karşılaştırması

| **Değer aralığı** | **İlkel veri tipi / bellek ihtiyacı (byte)** | **ds_Int veri tipi / bellek ihtiyacı (byte)** | **Bellek kazancı (byte)** |
|---|---|---|---|
| 0 - $2^8$ | unsigned char / 1 | ds_Int<1> / 1 | 0 |
| 0 - $2^{16}$ | unsigned short int / 2 | ds_Int<2> / 2 | 0 |
| 0 - $2^{24}$ | unsigned int / 4 | ds_Int<3> / 3 | 1 |
| 0 - $2^{32}$ | unsigned int / 4 | ds_Int<4> / 4 | 0 |
| 0 - $2^{40}$ | unsigned long int / 8 | ds_Int<5> / 5 | 3 |
| 0 - $2^{48}$ | unsigned long int / 8 | ds_Int<6> / 6 | 2 |
| 0 - $2^{56}$ | unsigned long int / 8 | ds_Int<7> / 7 | 1 |
| 0 - $2^{64}$ | unsigned long int / 8 | ds_Int<8> / 8 | 0 |

*ds_Int* veri tipi kullanımı ile bellek kazancının nasıl sağlanabileceğini bir örnek üzerinden gösterelim.

**Örnek:** En yüksek değerleri 240, 123.500, 17.250.500.300 olacağını öngördüğümüz üç sayıyı depolamak için bir yapıyı temel veri tipleri ve ds_Int ile oluşturup bellek alanı ihtiyaçlarını Şekil 3-6'daki gibi karşılaştıracak olursak, *ds_Int*'in $\left(1 - \frac{13}{9}\right) = 44\%$ oranında bellek kazancı sağladığı görülmektedir.

| Yapı (temel veri tipi) | Yapı (ds_Int) |
|---|---|
| unsigned char i1;<br>unsigned int i2;<br>unsigned long int i3; | ds_Int<1> i1;<br>ds_Int<3> i2;<br>ds_Int<5> i3; |
| 13 $byte$ | 9 $byte$ |

Şekil 3-6: Örnek bir yapıda ds_Int ile bellek kazancı

### 3.1.3 Veri Yapısı Paketleme

PDSSP çözümü içerisindeki temel bileşen olan ds_Int veri tipinin bir önceki bölümde gösterilen Şekil 3-6 örneğindeki uzunluk miktarında veri paketlemedeki hizalama boyutu göz ardı edilmiştir. Aynı durumu veri hizalamanın 4 byte sözcük uzunluklu olduğunu varsayıp hesapladığımızda, Şekil 3-7'te görüldüğü üzere bellek ihtiyacının arttığı görülmektedir.

| Yapı (temel veri tipi) | Sıra | Yapı (ds_Int) | Sıra |
|---|---|---|---|
| **struct** {<br>  **unsigned char** i1;<br>  **char** pad[3];<br>  **unsigned int** i2;<br>  **unsigned long int** i1;<br>} | <br>1<br>1<br>2<br>3,4 | **struct** {<br>  **ds_Int<1>** di1;<br>  **ds_Int<1>** pad[3];<br>  **ds_Int<3>** di2;<br>  **ds_Int<1>** pad[1];<br>  **ds_Int<5>** di3;<br>} | <br>1<br>1<br>2<br>2<br>3,4 |
| Toplam bellek ihtiyacı → | 16 $byte$ | | 16 $byte$ |

Şekil 3-7: Veri hizalama uzunluğu 4 byte olarak Şekil 3-6'nın hesaplanması

Veri hizalama nedeniyle gerçekte bellek alanı ihtiyacı ds_Int ve temel veri tipleri için aynı görünmektedir. Bu noktada bellek kazancını sağlayabilmek için verilerin tekrar sıralamasını sağladığımız durumda, Şekil 3-8'de görüldüğü şekilde bellek ihtiyacı $\left(1 - \frac{16}{12}\right) = 25\%$ oranında azalmış ve *ds_Int* ile bellek kazanımı sağlanmış olacaktır.

| Yapı (temel veri tipi) | Sıra | Yapı (ds_Int) | Sıra |
|---|---|---|---|
| **struct** {<br>**unsigned long int** i1;<br>**unsigned int** i2;<br>**unsigned char** i1;<br>} | <br>1,2<br>3<br>4 | **struct** {<br>**ds_Int<5>** di3;<br>**ds_Int<3>** di2;<br>**ds_Int<1>** di1;<br>} | <br>1,2<br>2<br>3 |
| Toplam bellek ihtiyacı → | 16 *byte* | | 12 *byte* |

Şekil 3-8: Yeniden sıralama sonrası Şekil 3-6'nın tekrar hesaplanması

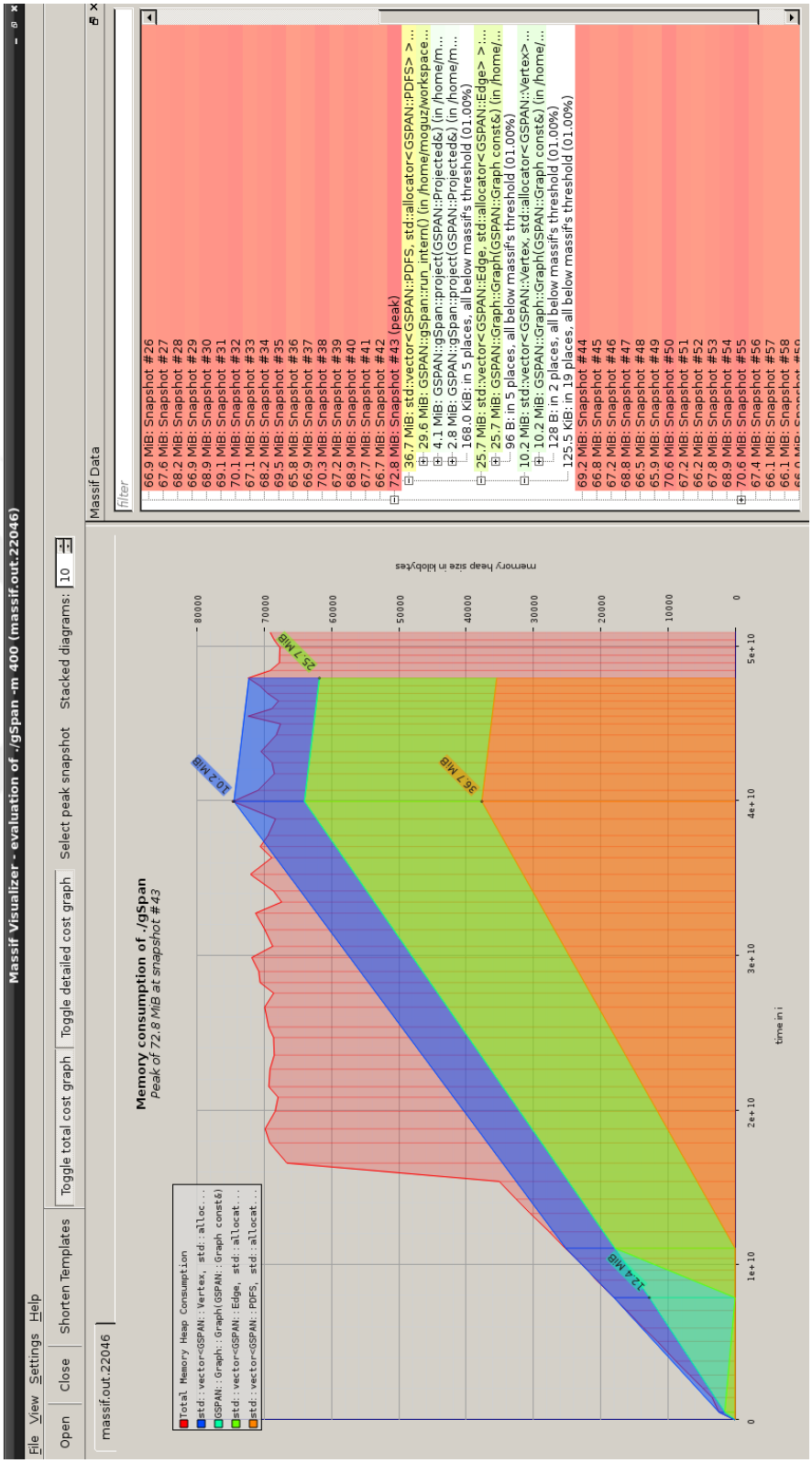
Eğer ds_Int ile veri sıralaması değişimi sonrası paketleme boyununda değişmesi gerekirse, bu durumda *#pragma pack(n)* ile bu değişimde sağlanabilir.

### 3.1.4 gSpan'e Uygulanması

PDSSP methodunun nasıl uygulandığı gSpan algoritmasının bu tez kapsamında kullanılan kodları üzerinden anlatılacaktır.

gSpan uygulamasının çalışması sırasında kullandığı bellek ihtiyacının analiz edilmesi ve bu sırada en yüksek bellek ihtiyacı duyulan zaman diliminde hafızada bulunan yapılar incelenerek PDSSP methodunun kullanılacağı kısımlar belirlenmiştir. gSpan, örnek bir veri seti ile çalıştırılarak bellek profillemesi yapılmış ve Şekil 3-9'daki şekilde görülmüştür.

gSpan'in en yüksek bellek kullanımı (peak) sırasında iki vector yapısının bellekte bulunduğu görülmüştür. Bu yapılardan bir tanesi düğüm yapılarının tutulduğu vector, diğeri ise DFS kodlarının tutulduğu vector yapıdır. Şekil 3-10'da bu yapılar 64 bit ortamdaki bellek ihtiyaçları ile birlikte görülmektedir.




Şekil 3-9: gSpan'nin çalışma zamanı sırasındaki bellek profillemesi

| **Yapı 1 (Düğüm)** | **Yapı 2 (DFS kodları)** |
|---|---|
| **struct** Edge {<br>**int** from;<br>**int** to;<br>**int** elabel;<br>**unsigned int** id;<br><br>//diğer kodlar<br>}; | **struct** PDFS {<br>**unsigned int** id;<br>**Edge** *edge;<br>**PDFS** *prev;<br><br><br>//diğer kodlar<br>}; |
| 16 *byte* | 20 *byte* |

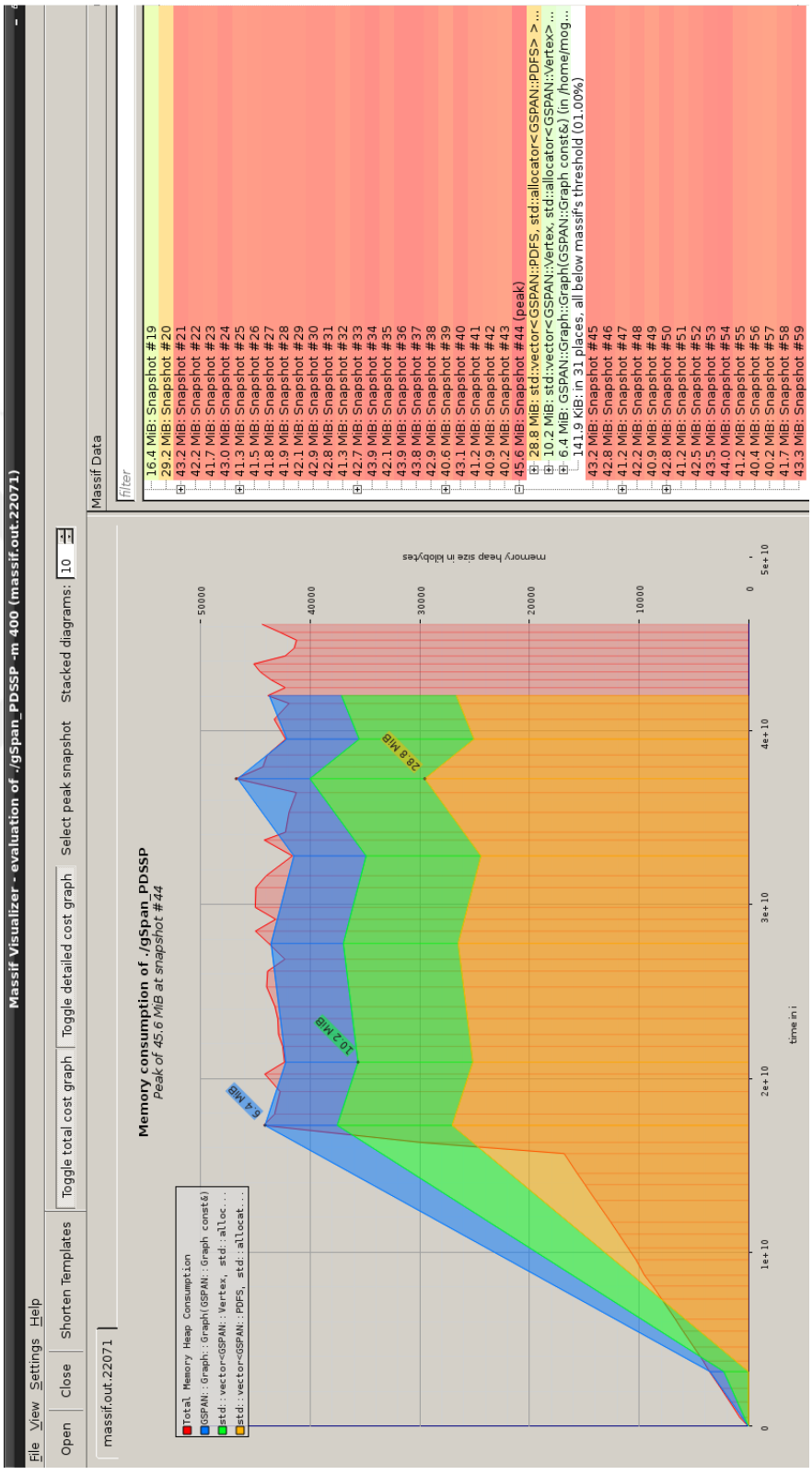
Şekil 3-10: gSpan içerisindeki yüksek bellek ihtiyacı olan veri yapıları

*ds_Int*'in uygulanacağı bu yapıların tesbitinden sonra uygulamada bu yapılar Şekil 3-11'de göründüğü şekillere dönüştürülmüştür. PVS'nin veri setlerini analiz etmesi sonrası gerekli dört sayısal değer tid_max (en yüksek işlem parçacık değeri), v_max (en yüksek düğüm değeri), e_label_max (en yüksek kenar etiken değeri) ve e_max (en yüksek kenar değeri) belirlenir. Ardından ilgili *ds_Int<n>* değerlerini içeren gSpan_PDSSP sürümü çalıştırılır.

| **Yapı 1 (Düğüm)** | **Yapı 2 (DFS kodları)** |
|---|---|
| *#pragma pack(n)*<br>**struct** Edge {<br>**ds_Int<v_max>** from;<br>**ds_Int<v_max>** to;<br>**ds_Int<e_label_max>** elabel;<br>**ds_Int< e_max >** id;<br><br>//diğer kodlar<br>};<br>*#pragma pack()* | *#pragma pack(n)*<br>struct PDFS {<br>**ds_Int< tid_max >** id;<br>**Edge** *edge;<br>**PDFS** *prev;<br><br><br>//diğer kodlar<br>};<br>*#pragma pack()* |

Şekil 3-11: Şekil 3-10'un ds_Int veri tipi ile oluşturulmuş şekli

Yukarıdaki uygulama sonrası gSpan_PDSSP'nin bellek profillemesi Şekil 3-9'daki değerler için tekrar yapıldığında maksimum bellek kullanımının 72MB değerinden 45MB değerine indiği, bellek kullanım eğrisinin farklılaştığı Şekil 3-12'deki gibi olduğu gözlemlenmiştir.




Şekil 3-12: gSpan_PDSSP'nin çalışma zamanı sırasındaki bellek profillemesi

### 3.1.5 Teorik Sınırları

PDSSP methodunun uygulanması sırasında bu tez çalışmasının esas aldığı aşağıdaki kuralların her durumda doğruluğu varsayılmış, geliştirme ve testler bu temeller esas alınarak yapılmıştır.

- *ds_Int* veri tipi içerisinde barındırdığı bit kaydırma operasyonları için bilgisayar mimarisinden bağımsız olarak Intel, ARM, Itanium vb. için dizi yapıları içerisinde aynı şekilde çalışmaktadır.
- İlkel veri tiplerinin veri hizalaması ds_Int içerisindeki dizi yapısında kullanılan veri hizalama ile aynı şekilde çalışmaktadır.
- Örnek veri setlerinin barındırdığı çizge sayıları, çizgeler içerisindeki düğüm, kenar ya da etiket sayılarının en yüksek değerleri unsigned integer değerinin altında kalmaktadır. Eğer bu değer unsigned long integer veri tipi değerine ihtiyaç duyarsa ds_Int'in sağladığı bellek kullanım kazancı artacaktır.
- *ds_Int*<*n*> yapısı şablon sınıf (template class) olarak oluşturulmuştur ve C++ standartları nedeniyle parametre olarak verilen *n* değeri derleme zamanında belirlenmelidir. Bu nedenle gSpan_PDSSP'nin farklı sürümlerinin oluşturulması gereklidir.
- *ds_Int* temel veri tipleri ile kullanım karşılaştırması yapıldığında daha yavaş yazma ve okuma zamanına sahiptir. Ancak kullanıldığı algoritma yazma, okuma ve silmeyi çok sık yapmıyorsa bu durumda çalışma hızındaki yavaşlık yukarıdaki deneylerde olduğu gibi görünmeyecektir.

### 3.1.6 Deneysel Çalışmalar

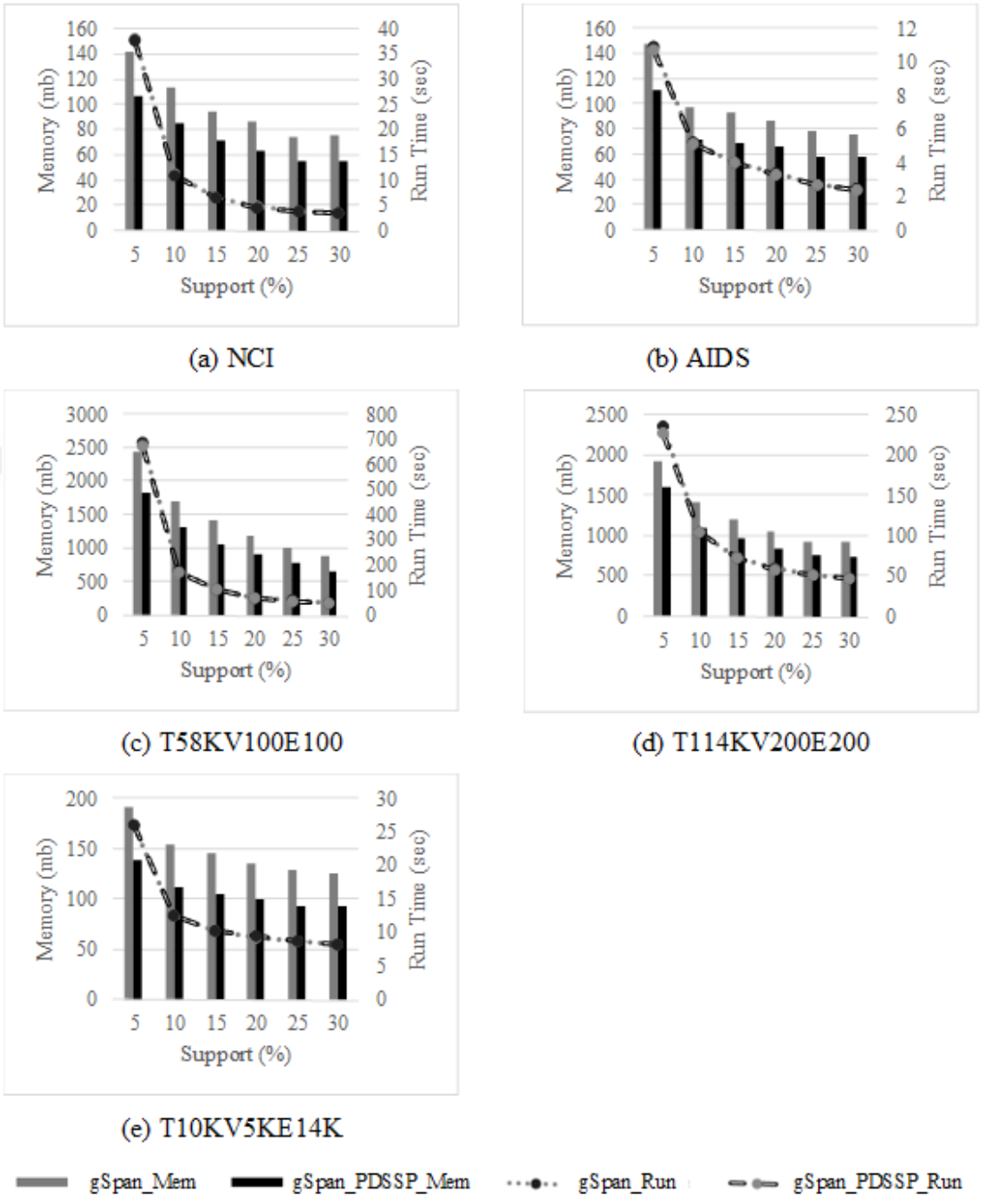
gSpan ve gSpan_PDSSP'nin Çizelge 2-5'daki 1, 2, 3, 4 ve 5 nolu veri setleri ile yapılan deneysel çalışmalarından elde edilen çalışma zamanları ve en yüksek bellek kullanım karşılaştırmaları Şekil 3-13'de verilmiştir. Bellek kullanımları soldaki dikey

eksene karşılık MB cinsinden ve grafik içerisinde dikey barlarla gösterilmiş, çalışma zamanları sağdaki dikey eksene karşılık saniye cinsinden ve grafik içerisinde çizgilerle gösterilmiştir.

Grafikler incelendiğinde şu sonuçlara varılmaktadır:

- Destek seviyesi azaldıkça bellek kullanım ihtiyacı gSpan ve gSpan_PDSSP için yükselmektedir.
- gSpan ve gSpan_PDSSP'nin çalışma zamanları karşılaştırıldığında, gSpan_PDSSP en az gSpan kadar hızlı çalışmaktadır.
- gSpan ve gSpan_PDSSP'nin bellek kullanımları karşılaştırıldığında, deneyde kullanılan tüm veri setleri için gSpan_PDSSP gSpan'e göre 25%'lere varan oranda daha az bellek kullanmaktadır.

Veri setlerinin içerisindeki en yüksek sayısal değerler unsigned integer veri tipinin altında kaldığından bu deneylerin $2^{32}$'den yüksek çizge sayısı ile test edilmesi durumunda gSpan_PDSSP'nin bellek ihtiyacındaki azalma 25%'den daha yüksek olacaktır.




Şekil 3-13: gSpan ve gSpan_PDSSP'nin farklı veri setleri ile karşılaştırılması

# 4 ZAMAN VERİMLİLİĞİ ÇALIŞMASI

Bu bölümde SAÇM algoritmalarının zaman verimliliğini arttırmaya yönelik geliştirilen ANPTR methodu hakkında bilgi verilmiştir. Methodun nasıl çalıştığı örnek üzerinden anlatılmış, temsili kodu verilmiş, uygulanması sonucu gSpan algoritmasının çalışma zamanı değişimi profilleme üzerinden gösterilmiştir. Son kısımda veri setleri üzerinde yapılan deneysel sonuçlar paylaşılmıştır.

## 4.1 ANPTR Methodu

ANPTR, gSpan algoritmasının OpenMP teknolojisi ile paralelleştirilmiş bir şeklidir. SAÇM algoritmaları genelinde olan iki probleme çözüm bulmayı amaçlamaktadır. Bu problemler yük dağılımının eşit olmaması nedeniyle tek seviye paralelleştirmenin yetersiz kalması ve çok seviye paralelleştirmenin nasıl ve kaç seviye yapılacağı ile buna bağlı iş parçacıklarının nasıl dağıtılacağıdır.

### 4.1.1 Methodun Yapısı

gSpan algoritmasının Şekil 2-15'deki orijinal temsili kodu üzerinde uygulanan ANPTR methodu sonucu oluşan paralelleştirilmiş temsili kodu Şekil 4-1'deki gibidir.

Graphset_projection methodu her bir kenar için işleme başlayacağı zaman iş parçacığı yönetim methodu olan Get_AvailableThread() ile kontrol yapılmaktadır. Bu kontrol sonucu paralelleştirmeye izin verilir ya da verilmez. Genel çalışma mantığı aşağıdaki şekildedir.

1. İç içe paralelleştirme seviyesi CPU çekirdek sayısının iki katını aşmıyorsa,

   a. Boşta duran iş parçacığı sayısı 1 adetse, paralelleştirme 2 iş parçası ile yapılacaktır,
   b. Boşta duran iş parçacığı sayısı 2 adetten fazla ise, paralelleştirme 3 iş parçacığı ile yapılacaktır,
2. Diğer durumda 1 değeri dönülecek ve paralelleştirmeye izin verilmeyecektir.

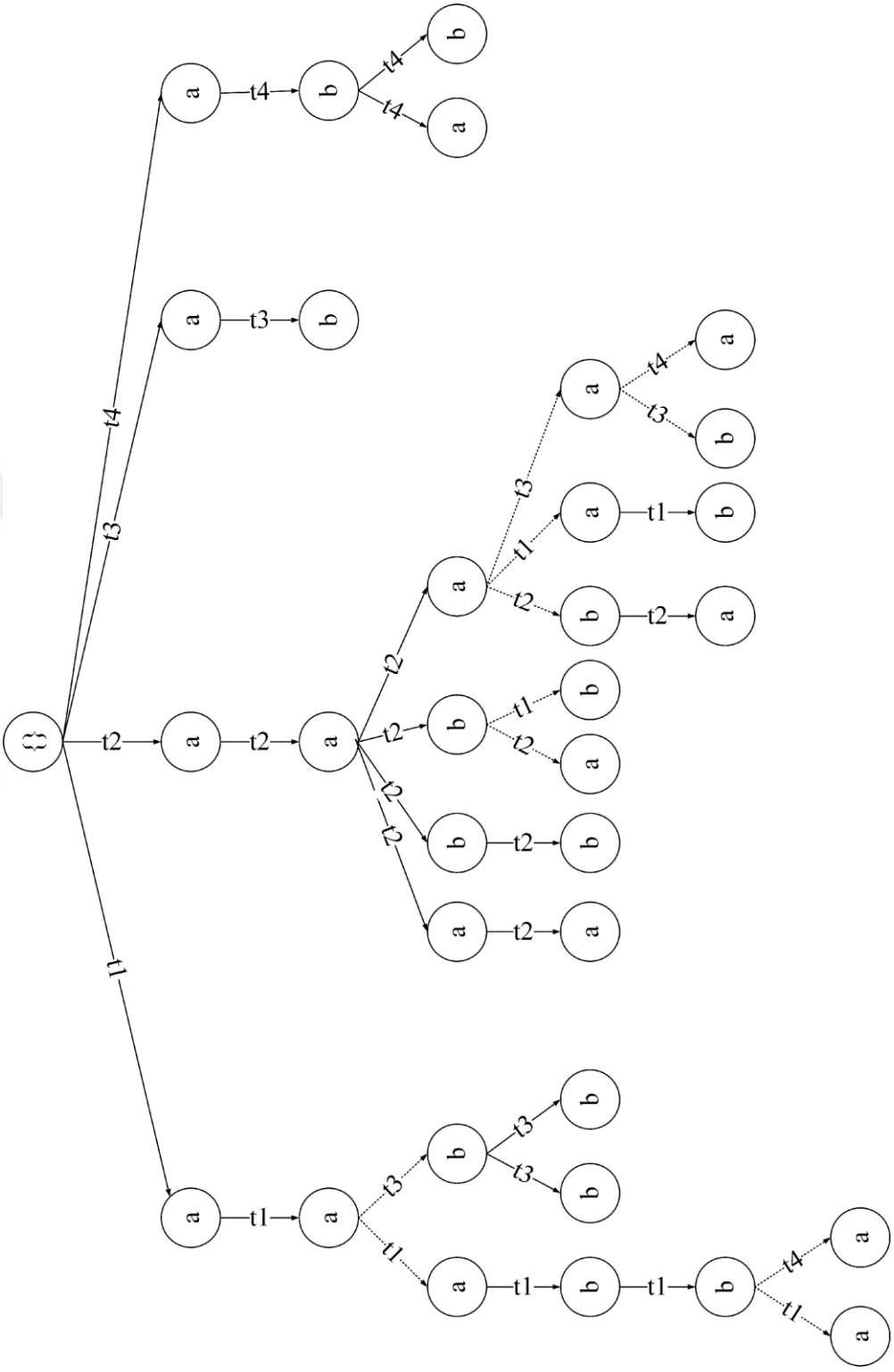
ANPTR'nin nasıl çalıştığını anlamak için Şekil 4-2'de örnek DFS ağacını 4 iş parçacıklı bir sistemde işlediğimizi varsayarsak methodun çalışmasını aşağıdaki şekilde özetleyebiliriz.

1. Örnekte paralelleştirme yapılan kısımlar noktalı, seri çalışmaya devam eden kısımlar ise düz çizgi olarak gösterilmiştir. Tüm DFS ağacı gSpan çalıştıkça yukarıdan aşağıya doğru dinamik olarak oluşmaktadır. Her oluşan düğüm noktasında destek seviyesi kontrolü yapılmaktadır.
2. Her bir düğümde işlemin 1 birim zaman aldığını varsayalım.
3. Tüm DFS ağacı bir iş parçacıklı sistemde çalıştırılırsa 33 birim zaman alacaktır.
4. ANPTR ile ilk seviyede maksimum iş parçacığı kullanımı ile 4 iş parçacığı (t1, t2, t3 ve t4) birer DFS ağaç dalını alıp işlemeye başlayacaktır.
5. Eğer tek seviye paralelleştirme yapılırsa buradaki DFS ağacının en hızlı üretilebildiği zaman ağaçtaki en uzun dalın (örnekte 2 numaralı ağaç dalı) bitirilme süresi olmaktadır. Bu durumda tüm iş 17 birim zamanda bitirilebilmekte ve $\frac{33}{17} = 1.9X$ hızlanma sağlanabilmektedir.
6. ANPTR methodu çok seviyeli paralelleştirme ile boşa çıkan her iş parçacığını alt seviye dallarda yeni bir paralelleştirme için kullanmaya devam etmektedir. Şekildede görüldüğü üzere en uzun dalda alt seviyelerde iş parçacıkları kullanılarak paralelleştirme devam etmektedir.
7. Tüm işlerin ANPTR methodu ile paralelleştirilmesi sonucu DFS ağacının üretilebilme zamanı 11 birim zaman almaktadır. Bu durumda $\frac{33}{11} = 3X$ hızlanma elde edilebilmektedir.

8. Çok seviyeli paralelleştirmede belli bir seviyeden sonra boşta iş parçacığı bulunsa bile yeni alt dallarda paralelleştirme yapılmamaktadır. Çünkü LOP değeri arttıkça iş parçacığı yönetim yükü artmaktadır ve bu yük nedeniyle yeni seviye paralelleştirmenin sağladığı kazanç kaybedilen zamandan az olmaktadır. Örnekte bu durumun oluşabilmesi için 4 iş parçacığının iki katı olan 8 seviyesinde bir paralelleştirme için dallar yeterince uzun değildir. Bu bölüm sonundaki deneysel çalışmalarda bu seviyeler fazlasıyla aşıldığı için ideal LOB değerinin bulunması ve sınırlamanın uygulanması yapılmıştır.

```
Set AvailableThreadCount to CPU core count
Metod 1: GraphSet_projection(GS, FS)
    sort labels of the vertices and edges in GS by frequency;
    remove infrequent vertices and edges;
    relabel the remaining vertices and edges (descending);
    S^1:= all frequent 1-edge graphs;
    sort S^1 in DFS lexicographic order;
    FS := S^1;
    n := Get_AvailableThread ()
    for each edge e in S^1 do, in parallel for n thread if n is greater than 1
        init g with e, set g.GS by graphs which contains e.
        Subgraph_mining(GS, FS, g);
        GS := GS - e;
        if |GS| < minSup
          break;
        Increase AvailableThreadCount
Metod 2: Subgraph_mining(GS, FS, g)
    if g ≠ min(g)
        return;
    FS := FS ∪ {g};
    enumerate g in each graph in GS and count g's children;
    for each c (child of g') do
        if support(c) ≥ minSup
        Subgraph_mining(GS, FS, c);
Metod 3: Get_AvailableThread ()
    if active nested parallelism level is lower than "2XCPU core count"
        if AvailableThreadCount is greater than 1
            decrease AvailableThreadCount by 2
            return 3;
        if AvailableThreadCount is greater than 0
            decrease AvailableThreadCount by 1
            return 2;
    return 1;
```

Şekil 4-1: gSpan_ANPTR algoritmasının temsili kodu

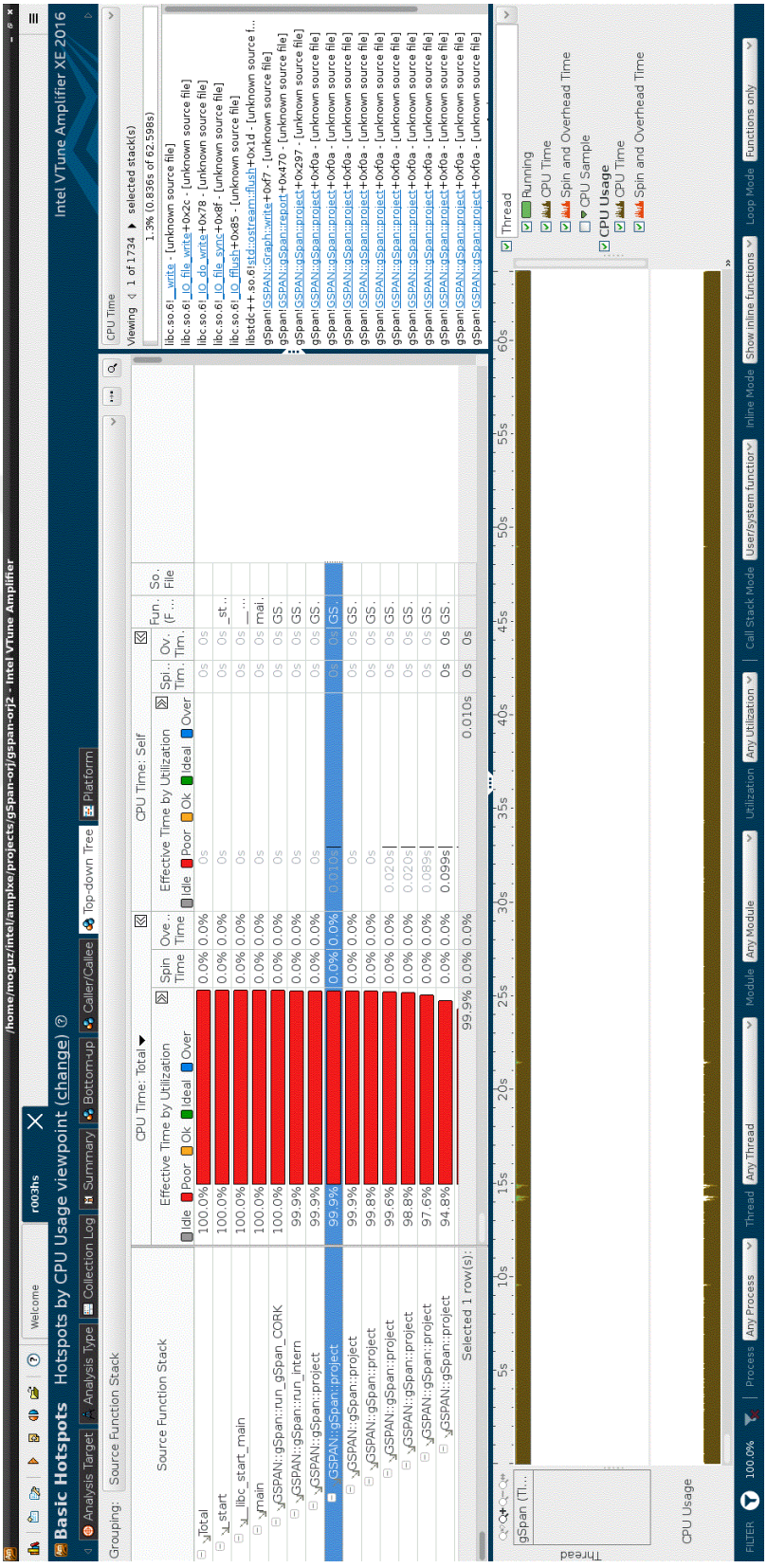



Şekil 4-2: gSpan_ANPTR'nin çalışma örneklemesi

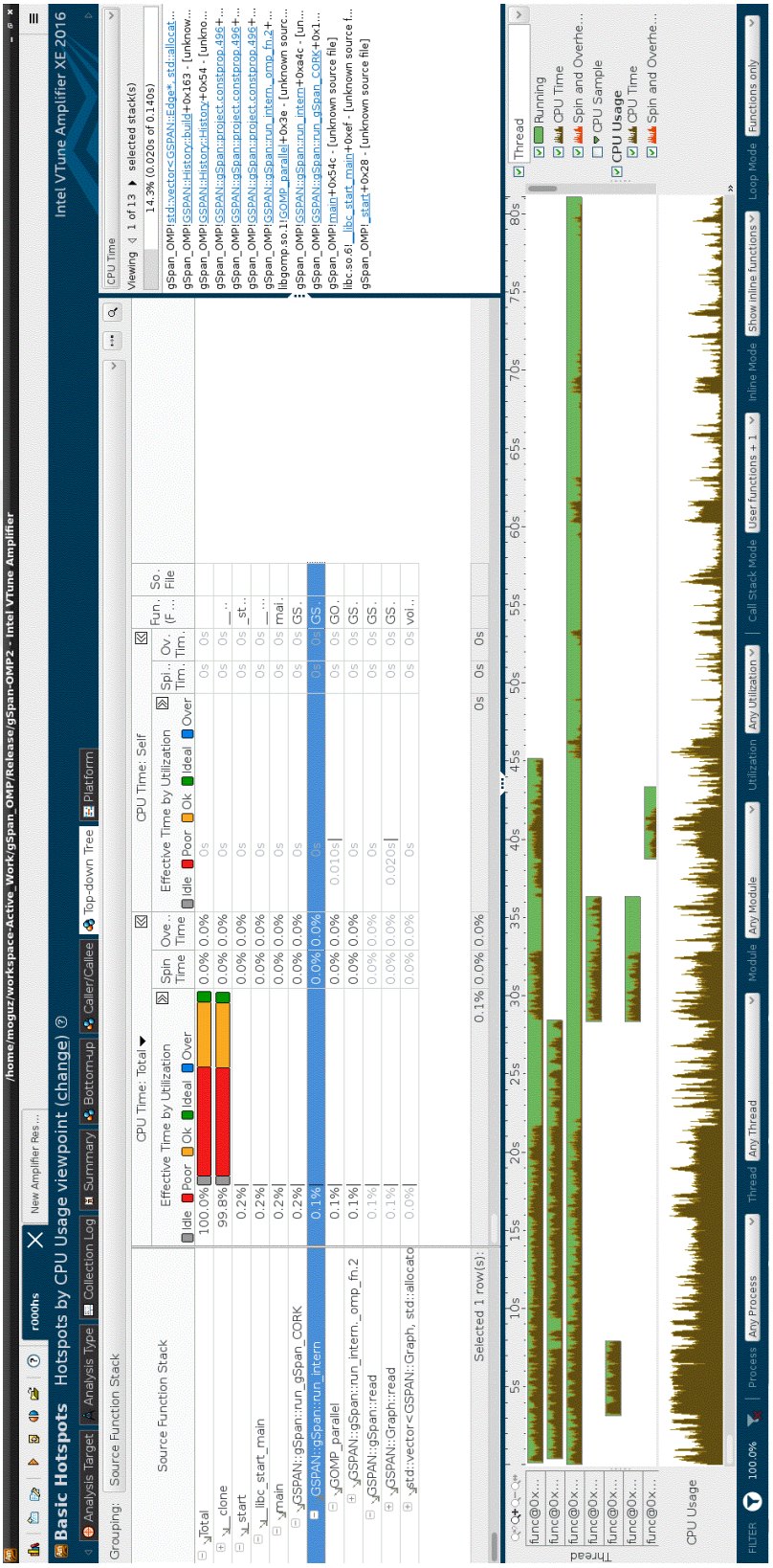
### 4.1.2 gSpan'e Uygulanması

gSpan kodunun çalışma profillemesi Şekil 4-3'deki gibidir. Ana method *run_gSpan_CORK()* çalıştıktan sonra *run_intern()* methodu çağırılır ve özyinelemeli şekilde *Project()* isimli method çağırılarak program devam eder. Şekil 2-15'deki temsili kodlarla karşılaştırıldığında *run_intern()* methodu *GraphSet_projection* methoduna, *project()* methodu ise *Subgraph_mining* methoduna karşılık gelmektedir. Şekilin alt kısmında programın tek bir iş parçacığı ile çalıştığı, tüm süreyi bu tek iş parçacığının aldığı ve CPU kullanım oranının tüm yaşam süreci boyunca yüksek olduğu görülmektedir.

gSpan_ANPTR ile ilk seviye paralelleştirme *run_intern()* methodu içerisinde yapılmıştır. Ardından özyinelemeli olarak çağırılan *project()* methodu içerisinde ileri düğüm (forward edge) ve geri düğüm (reverse edge) oluşturmak için methodun tekrar çağırılması kısımlarıda ayrıca paralelleştirilmiştir. Bu şekilde çok seviyeli bir paralelleştirme elde edilmiştir. Programın dört iş parçacıklı bir sistemde çalışma zamanı profillemesi yapıldığında Şekil 4-4'deki durum görülmektedir. Paralelleştirme *run_intern()* methodu içerisinde başlamaktadır. Oluşan iş parçacıklarına bakıldığında bazı parçacıkların uzun süre çalıştığı, birçok iş parçacığının ara ara iş yapıp sonlandığı, CPU çalışma zamanının genel olarak yüksek olduğu gözlenmektedir. Ekran resmi belirli bir kısmı gösterebildiğinden toplam iş parçacığı sayısı gösterilememiştir. Ancak Şekil 4-3'de tek bir iş parçacığı oluşması ile yapılan işlem, burada toplam 9,479 adet iş parçacığının kullanımı ve aynı anda 4 adet iş parçacığının çalıştırılması ile paralelleştirilmiştir.




Şekil 4-3: gSpan kodunun çalışma profillemesi



Şekil 4-4: gSpan_ANPTR kodunun çalışma profillemesi

### 4.1.3 Teorik Sınırları

ANPTR methodunun uygulanması sırasında bu tez çalışmasının esas aldığı aşağıdaki kuralların doğruluğu varsayılmış, geliştirme ve testler bu temeller esas alınarak yapılmıştır.

- OpenMP teknolojisi derleyici destekli bir teknoloji olduğundan donanım mimarisindan bağımsız olarak C, C++ derleyici desteğiyle tüm donanımlarda çalışabilir.
- Kullanılan veri setleri farklı düğüm, kenar, etiket gibi bilgiler içerdiğinden, yapılan testlerdeki verim en yüksek değerlerin alınması ile elde edilmiştir.
- gSpan algoritmasının gerçek hayata geçirilmiş hali olan C kodları algoritmanın bire bir özelliklerini barındırmaktadır.
- Belirlenen ideal LOP değeri eldeki veri setleri üzerinde yapılan deneylere dayalıdır. Veri setlerinin değişmesi durumunda, daha verimli bir paralelleştirme için farklı LOP değerinin kullanılması gerekebilir.

### 4.1.4 Deneysel Çalışmalar

Çizelge 2-5'deki 6, 7, 8 ve 9 nolu veri setleri kullanılarak yapılan deneylerde öncelikli amaç ideal LOP değerinin belirlenmesidir. Çizelge 4-1'de gSpan'in 5% ile 20% arasında değişen destek seviyeleri için yapılan deneysel test sonuçları görünmektedir. Bu çizelgedeki sonuçlar gSpan_ANPTR çalışma sonuçları karşılaştırması için temel alınmıştır.

Çizelge 4-1: gSpan algoritmasının çalışma zamanları (saniye)

| **Veri seti** | **Destek seviyesi** | | | |
|---|---|---|---|---|
| | **20%** | **15%** | **10%** | **5%** |
| **mcf-7** | 107 | 199 | 611 | 10563 |
| **ovcar** | 161 | 311 | 896 | 11374 |
| **sn12c** | 159 | 300 | 892 | 11669 |
| **nci-h23** | 159 | 300 | 865 | 11370 |

İdeal LOP değerini bulabilmek için, 16 iş parçacıklı sistemde 5% ile 20% arasında değişen destek seviyeleri için yapılan test sonuçları Çizelge 4-2'de görünmektedir. Bu sonuçlara göre, destek seviyesi yüksek ve çalışma zamanı düşük olduğu durumlarda maksimum iş parçacığı sayısının yüksek ya da düşük katı miktarında LOP değeri kullanımı benzer sonuçları vermektedir. Destek seviyesinin düşmesi ve algoritma çalışma zamanının yükselmesi durumları için en tutarlı ve yüksek paralelleştirme verimi elde edilen LOP değeri, maksimum iş parçacığı sayısının iki katı olduğu durumlardır. Buradan Denklem 4.1'i ANPTR methodu için oluşturabiliriz.

$$ANPTR - LOP_{max} = 2\times\text{maksimum iş parçacığı sayısı} \qquad (4.1)$$

Çizelgedeki tutarlı sonuçlar için tüm testler üçer kez yapılmış ve Denklem 4.2'deki şekilde standart sapma değerleri üzerinden standart sapmaları düşük olan test sonuçları tutarlı bir paralelleştirme sonucu için seçilmiştir.

$$\sigma = \sqrt{\frac{1}{N} \times \sum_{i=1}^{N} (x_i^2 - N\tilde{x}^2)} \tag{4.2}$$

İdeal LOP değerinin belirlenmesi sonucunda gSpan_ANPTR algoritmasının 2 ile 16 arası değişen iş parçacıklı sistemde, 5% ile 20% arasında değişen destek seviyeleri için yapılan test sonuçlarının çalışma süreleri ve hızlanma zamanları Şekil 4-5'de verilmiştir.

Grafikler incelendiğinde şu sonuçlara varılmaktadır:

- Paralelleştirmede 16 iş parçacıklı sistemde 13X'e yaklaşan hızlanma elde edilmiştir.
- Destek seviyesi azaldıkça programın çalışma zamanı artmaktadır. Çalışma zamanının artması sırasında gSpan_ANPTR'nin paralelleştirme verimide yükselmektedir.
- Amdahl yasasına göre, gSpan'in paralelleşen kısmı 95%'in üzerindedir.
- gSpan_ANPTR'nin algoritma etkinliği $\frac{13}{16} = 81\%$'dir.
- Şekil 2-28 ve Şekil 4-5b karşılaştırıldığında, iş parçacığı arttırmanın hızlanmaya etkisinin neredeyse olmadığı noktaya gSpan_ANPTR henüz yaklaşmamıştır. Testler 16 işlemciden daha yüksek bir sistemde yapılırsa, paralelleşmede 13X'in üzerine çıkılabilecektir.

Çizelge 4-2: gSpan_ANPTR için ideal LOP değeri bulunması

| LOP | Dataset | 20% | | | | | 15% | | | | | 10% | | | | | 5% | | | | | Max SU |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Min | Max | Avg | SD | Avg SU | Min | Max | Avg | SD | Avg SU | Min | Max | Avg | SD | Avg SU | Min | Max | Avg | SD | Avg SU | |
| 64 | mcf-7 | 16 | 22 | 19 | **3** | **5.6** | 34 | 34 | 34 | **0** | **5.9** | 66 | 96 | 81 | **15** | **7.5** | 893 | 944 | 919 | 26 | 11.5 | 12 |
| | ovcar | 27 | 27 | 27 | **0** | **6.0** | 43 | 47 | 45 | **2** | **6.9** | 132 | 134 | 133 | **1** | **6.7** | 1017 | 1161 | 1089 | 72 | 10.4 | 11 |
| | sn12c | 18 | 27 | 23 | **5** | **7.1** | 36 | 55 | 46 | **10** | **6.6** | 123 | 128 | 126 | **3** | **7.1** | 1011 | 1094 | 1053 | 42 | 11.1 | 12 |
| | nci-h23 | 18 | 22 | 20 | **2** | **8.0** | 43 | 54 | 49 | **6** | **6.2** | 96 | 108 | 102 | **6** | **8.5** | 1130 | 1187 | 1159 | 29 | 9.8 | 10 |
| 32 | mcf-7 | 15 | 17 | 16 | 1 | 6.7 | 33 | 36 | 35 | **2** | **5.8** | 75 | 80 | 78 | **3** | **7.9** | 864 | 917 | 891 | **27** | **11.9** | **12** |
| | ovcar | 29 | 33 | 31 | 2 | 5.2 | 40 | 56 | 48 | **8** | **6.5** | 109 | 120 | 115 | **6** | **7.8** | 1011 | 1079 | 1045 | **34** | **10.9** | 11 |
| | sn12c | 22 | 27 | 25 | 3 | 6.5 | 41 | 55 | 48 | **7** | **6.3** | 123 | 144 | 134 | **11** | **6.7** | 932 | 1066 | 999 | **67** | **11.7** | **13** |
| | nci-h23 | 23 | 26 | 25 | 2 | 6.5 | 44 | 54 | 49 | **5** | **6.1** | 109 | 120 | 115 | **6** | **7.6** | 1072 | 1092 | 1082 | **10** | **10.5** | 11 |
| 24 | mcf-7 | 19 | 19 | 19 | 0 | 5.6 | 31 | 36 | 34 | 3 | 5.9 | 80 | 95 | 88 | 8 | 7.0 | 865 | 885 | 875 | **10** | **12.1** | **12** |
| | ovcar | 19 | 22 | 21 | 2 | 7.9 | 44 | 52 | 48 | 4 | 6.5 | 117 | 141 | 129 | 12 | 6.9 | 945 | 1142 | 1044 | **99** | **10.9** | **12** |
| | sn12c | 22 | 29 | 26 | 4 | 6.2 | 45 | 61 | 53 | 8 | 5.7 | 101 | 140 | 121 | 20 | 7.4 | 1065 | 1106 | 1086 | **21** | **10.7** | 11 |
| | nci-h23 | 18 | 29 | 24 | 6 | 6.8 | 50 | 50 | 50 | 0 | 6.0 | 101 | 118 | 110 | 9 | 7.9 | 970 | 1087 | 1029 | **59** | **11.1** | **12** |
| 16 | mcf-7 | 15 | 16 | 16 | 1 | 6.9 | 30 | 37 | 34 | 4 | 5.9 | 74 | 98 | 86 | **12** | **7.1** | 922 | 923 | 923 | 1 | 11.5 | 11 |
| | ovcar | 24 | 32 | 28 | 4 | 5.8 | 48 | 52 | 50 | 2 | 6.2 | 101 | 134 | 118 | **17** | **7.6** | 1000 | 1087 | 1044 | 44 | 10.9 | 11 |
| | sn12c | 26 | 27 | 27 | 1 | 6.0 | 44 | 58 | 51 | 7 | 5.9 | 114 | 134 | 124 | **10** | **7.2** | 1032 | 1105 | 1069 | 37 | 10.9 | 11 |
| | nci-h23 | 22 | 29 | 26 | 4 | 6.2 | 45 | 51 | 48 | 3 | 6.3 | 114 | 116 | 115 | **1** | **7.5** | 992 | 1063 | 1028 | 36 | 11.1 | 11 |
| 8 | mcf-7 | 12 | 21 | 17 | **5** | **6.5** | 25 | 38 | 32 | **7** | **6.3** | 82 | 86 | 84 | **2** | **7.3** | 1011 | 1017 | 1014 | 3 | 10.4 | 10 |
| | ovcar | 18 | 25 | 22 | **4** | **7.5** | 54 | 57 | 56 | **2** | **5.6** | 105 | 121 | 113 | **8** | **7.9** | 1063 | 2080 | 1572 | 509 | 7.2 | 11 |
| | sn12c | 21 | 26 | 24 | **3** | **6.8** | 38 | 50 | 44 | **6** | **6.8** | 110 | 125 | 118 | **8** | **7.6** | 1096 | 1258 | 1177 | 81 | 9.9 | 11 |
| | nci-h23 | 20 | 28 | 24 | **4** | **6.6** | 45 | 55 | 50 | **5** | **6.0** | 109 | 135 | 122 | **13** | **7.1** | 1078 | 1102 | 1090 | 12 | 10.4 | 11 |
| 4 | mcf-7 | 37 | 38 | 38 | 1 | 2.9 | 52 | 57 | 55 | 3 | 3.7 | 156 | 238 | 197 | 41 | 3.1 | 1904 | 5272 | 3588 | 1684 | 2.9 | 6 |
| | ovcar | 44 | 55 | 50 | 6 | 3.3 | 72 | 105 | 89 | 17 | 3.5 | 243 | 361 | 302 | 59 | 3.0 | 2534 | 2818 | 2676 | 142 | 4.3 | 4 |
| | sn12c | 40 | 45 | 43 | 3 | 3.7 | 127 | 162 | 145 | 18 | 2.1 | 213 | 216 | 215 | 2 | 4.2 | 6249 | 8050 | 7150 | 901 | 1.6 | 2 |
| | nci-h23 | 44 | 62 | 53 | 9 | 3.0 | 74 | 120 | 97 | 23 | 3.1 | 210 | 247 | 229 | 19 | 3.8 | 2234 | 2759 | 2497 | 263 | 4.6 | 5 |

| Dataset | Threads vs Run Times (sec) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
| mcf-7 | 10563 | 5621 | 2976 | 2163 | 1584 | 1404 | 1113 | 1013 | 864 |
| ovcar | 11374 | 9750 | 4310 | 2540 | 1989 | 1676 | 1305 | 1179 | 1011 |
| sn12c | 11669 | 8350 | 3662 | 2454 | 1913 | 1536 | 1242 | 1139 | 932 |
| nci-h23 | 11370 | 6552 | 3584 | 2413 | 1844 | 1549 | 1304 | 1099 | 1072 |




(a)

| Dataset | Threads vs Speedup (X) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
| mcf-7 | 0.0 | 1.9 | 3.5 | 4.9 | 6.7 | 7.5 | 9.5 | 10.4 | 12.2 |
| ovcar | 0.0 | 1.1 | 2.6 | 4.5 | 5.7 | 6.8 | 8.7 | 9.6 | 11.3 |
| sn12c | 0.0 | 1.3 | 3.2 | 4.8 | 6.1 | 7.6 | 9.4 | 10.2 | 12.5 |
| nci-h23 | 0.0 | 1.6 | 3.2 | 4.7 | 6.2 | 7.3 | 8.7 | 10.3 | 10.6 |




(b)

Şekil 4-5: gSpan_ANPTR için çalışma zamanları ve hızlanma oranları

# 5 SONUÇLAR VE DEĞERLENDİRME

SAÇM algoritmaları günümüzde bir çok alanda büyük veri setleri içerisinde farklı amaçla benzerlik tesbiti için kullanılmaktadır. Veri setlerinin büyüklüğü, yapılan işlemler sırasında yüksek bellek alanı ihtiyacının oluşmasına ve işlem sürelerinin saatler, günler ve hatta bazı durumlarda haftalar almasına neden olmaktadır. Bu nedenle SAÇM alanında yeni algoritmaların bulunmasına, var olan algoritmaların yapısal değişikliklerle zaman ve alan verimliliklerinin iyileştirilmesine ihtiyaç vardır.

Bu tez kapsamında geliştirilen PDSSP ve ANPTR methodları bu sorunların çözüme yönelik olarak gSpan algoritması üzerine kurgulanmış ve sağladıkları verimlilik iyileştirmeleri deneysel çalışmalarla gözlemlenmiştir. gSpan algoritmasının temel yapısına dokunulmamış, yalnızca bilgisayar bilimleri alanındaki teknikler kullanılmıştır. Tekniklerin dayandığı temeller üçüncü ve dördüncü bölümlerde bulunan teorik sınırlar başlığı altında sıralanmıştır. gSpan algoritmasının alan karmaşıklığı ve zaman karmaşıklığı sınıflandırmasında bir değişiklik yapılmamıştır, en kötü durum için ( $c_{worst}(n)$) verimlilik iyileştirmesi yapılmıştır.

Alan verimliliği üzerine yapılan PDSSP geliştirmesinin gSpan üzerine inşa edilmesi sonucu deneysel çalışmalar 25% oranında alan verimliliği artışı sağlandığını göstermiştir. Deneysel çalışmalarda veri setlerinin boyutları 32 bitlik işaretsiz sayısal verinin limitleri altında kalmıştır. İhtiyaç duyulan veri boyunun 32 bitlik değeri aşması durumunda, alan verimliliği artışı orijinal algoritmada 100%'e yakın oranda olacaktır. gSpan_PDSSP'de bellek artış oranının veri setleri içerisindeki düğüm, kenar, etiket sayısı gibi değerlere bağlı olarak daha az olacağı ve alan verimliliğindeki iyileştirme oranınında 40%'lara varabileceği öngörülmektedir. Methodun diğer bir avantajı ise, alan verimliliği sağlarken zaman verimliliği anlamında neredeyse herhangi bir negatif etki oluşturmamasıdır. Literatürde benzeri bir çalışma olmadığından bir karşılaştırma bu tez kapsmında yapılamamıştır.

Zaman verimliliği artışı üzerine yapılan ANPTR geliştirmesi, literatürde OpenMP teknolojisi kullanılarak yapılan çok seviyeli paralelleştirmenin gSpan üzerine uygulandığı ilk çalışma olarak görünmektedir. Deneysel çalışmalardaki sonuçlar benzeri çalışmalarla karşılaştırıldığında 8 iş parçacıklı sistemde 6.5X hızlanma sağlayarak literatürde bilinir FPGA paralelleştirmesi ile elde edilen 6.2X hızlanmanın üzerinde sonuç sağlanmış, 12 iş parçacıklı sistemde 10X'e varan hızlanma sağlayarak literatürde bilinen en yüksek değer olan 11X'lik hızlanmaya çok yakın sonuçlar elde edilmiş ve 16 iş parçacıklı bir sistemde 13X'e varan hızlanma sağlayarak literatürde görünen bir çok çalışmadan daha iyi sonuç sağlanmıştır. 13X'lik bir hızlanmanın anlamı yaklaşık 3 saatte biten bir işin 15 dakikadan az sürede tamamlanabilmesidir. ANPTR'nin önemli bir diğer özelliği ise elde edilen sonuçların farklı veri setleri ile yaklaşık değerlere ulaşmasıdır. gSpan_ANPTR algoritmasının paralelleştirme etkinliğinin yüksek olması daha yüksek işlemcili sistemlerde ve büyük veri setleri ile yapılacak deneysel sonuçlarda hızlanma oranının artacağını göstermektedir.

Bu çalışmaların devamı olarak, SMP sistemlerde üzerindeki paralelleştirmenin MPI ve OpenMP birleşimi karma (hybrid) hale getirilmesi ile bilgisayar kümesi üzerinde çalışmasının sağlanması, veri setlerinin her bilgisayar için çoklanması gerekliliği nedeniyle de PDSSP methodunun kullanılarak alan verimliliğinde iyileştirmenin sağlanması düşünülmektedir. Bu şekilde yüksek genişletilebilirliğe sahip, alan ve zaman verimliliği daha yüksek bir SAÇM algoritması elde edilebilir.

# KAYNAKLAR


1. Yan X., Jiawei H., "gspan: Graph-based substructure pattern mining." Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on. IEEE, 2002.
2. Anastasiu D.C., Jeremy I., Shaden S., Karypis G. "Big data frequent pattern mining." In Frequent Pattern Mining, pp. 225-259. Springer International Publishing, 2014.
3. Aggarwal C.C., Bhuiyan M.A., Al Hasan M., Frequent pattern mining algorithms: A survey. In: Aggarwal CC, Han J, editors. Frequent Pattern Mining. Switzerland: Springer International Publishing, pp. 19–64, 2014.
4. Nijssen S., Kok J.N., The Gaston tool for frequent subgraph mining. Electronic Notes in Theoretical Computer Science 2005;127;77–87, 2005.
5. Lakshmi K., Meyyappan DT. A comparative study of frequent subgraph mining algorithms. IJITCS 2012; 2; 2, 2012.
6. Fatta G.D., Berthold M.R., Dynamic load balancing for the distributed mining of molecular structures. IEEE Transactions on Parallel and Distributed Systems, 17(8):773– 785, 2006.
7. Wang C., Parthasarathy S., Parallel algorithms for mining frequent structural motifs in scientific data. In Annual International Conference on Supercomputing, ICS ’04, pages 31–40, NewYork, NY, USA, 2004.
8. Liu Y.,, Jiang X., Chen H., Ma J., Zhang X.,Mapreduce-based pattern finding algorithm applied in motif detection for prescription compatibility network. In Advanced Parallel Processing Technologies, pages 341–355. Springer, 2009.
9. Cook D.J., Holder L.B., Galal G., Maglothin R., Approaches to parallel graph-based knowledge discovery. Journal of Parallel and Distributed Computing, 61(3):427– 446, 2001.
10. Ray A., Holder L.B., Efficiency improvements for parallel subgraph miners. In Florida Artificial Intelligence Research Society Conference, FLAIRS ’12, 2012.

11. Reinhardt S., Karypis G., “A Multi-Level Parallel Implementation of a Program for Finding Frequent Patterns in a Large Sparse Graph,” In Proceedings of IPDPS, pp.1-8, 2007.
12. Robert K., Talukder N., Anchuri P., Zaki M., "Parallel Graph Mining with GPUs." In BigMine, pp. 1-16. 2014.
13. Stratikopoulos A., Chrysos G., Papaefstathiou I., Dollas A., HPC-gSpan: An FPGA-based parallel system for frequent subgraph mining. In Field Programmable Logic and Applications (FPL). In IEEE 2014 24th International Conference; Munich, Germany, IEEE. pp. 1-4, 2-4 Sep 2014.
14. Gregory B., Parthasarathy S., Nguyen A., Kim D., Chen Y.K., Dubey P., Parallel graph mining on shared memory architectures. Technical report, Columbus, OH, USA, 2005.
15. Meinl T., Worlein M., Fischer I., Philippsen, M., Mining molecular datasets on symmetric multiprocessor systems. In Systems, Man and Cybernetics, 2006. SMC'06. IEEE International Conference on (Vol. 2, pp. 1269-1274), IEEE. October 2006.
16. Fei W., Dong J., Yuan B., Graph-based substructure pattern mining using CUDA dynamic parallelism. In Intelligent Data Engineering and Automated Learning–IDEAL 2013, pp. 342-349. Springer Berlin Heidelberg, 2013.
17. Levitin A., Mukherjee S., Introduction to the design & analysis of algorithms. Reading, MA: Addison-Wesley, 2012.
18. OpenMP org, http://openmp.org/wp/presos/omp-in-action-SC05.pdf (01.01.2016)
19. Aridhi S., Doktorate Thesis, Distributed frequent subgraph mining in the cloud, 2014
20. Fayyad U., Knowledge discovery in databases: An overview, In Relational Data Mining, pp. 28-47. Springer Berlin Heidelberg, 2001.
21. Aggarwal C.C., Wang H., Managing and mining graph data. Vol. 40. New York: Springer, 2010.
22. Goto S., Okuno Y., Hattori M., Nishioka T., Kanehisa M., LIGAND: database of chemical compounds and reactions in biological pathways, Nucleic acids research 30, no. 1 (2002): 402-404, 2002.

23. Borgelt C., Berthold M.R., Mining molecular fragments: Finding relevant substructures of molecules, In Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on, pp. 51-58. IEEE, 2002.
24. Saidi R., Aridhi S., Nguifo E.M., Maddouri M., Feature extraction in protein sequences classification: a new stability measure. In Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine, BCB '12, pages 683–689, New York, NY, USA, 2012.
25. Pizzuti C., Rombo S.E., Marchiori E., Complex detection in protein-protein interaction networks: a compact overview for researchers and practitioners. In Proceedings of the 10th European conference on Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics, EvoBIO'12, pages 211–223, Springer-Verlag, Berlin, Heidelberg, 2012.
26. Bonchi F., Castillo C., Gionis A., Jaimes A., Social Network Analysis and Mining for Business Applications. ACM volume 2, pages 22:1–22:37, New York, NY, USA, May 2011.
27. Aggarwal C.C., Han J., Book, Frequent pattern mining. Springer, 2014.
28. Yan X., Han J., CloseGraph: Mining closed frequent graph patterns. In: ACM SIGKDD 2003 International Conference on Knowledge Discovery and Data Mining; Washington, DC, USA: ACM. pp. 286–295, 24–27 Aug 2003.
29. Vanetik N., Gudes E., Shimony S.E., Computing frequent graph patterns from semistructured data. In: IEEE 2002 International Conference on Data Mining, Maebashi City, Japan: IEEE. pp. 458–465, 9–12 Dec 2002.
30. Sun Z., Wang H., Wang H., Shao B., Li J., Efficient subgraph matching on billion node graphs. VLDB Endowment 2012; 5: 788–799, 2012.
31. Rehman S.U., Asghar S., Zhuang Y., Fong S., Performance evaluation of frequent subgraph discovery techniques. Mathematical Problems in Engineering 6, 2014.
32. Borgelt C., Berthold M.R., Mining molecular fragments: Finding relevant substructures of molecules. In IEEE International Conference on Data Mining, ICDM 2002, pages 51–58. IEEE, 2002.

33. Inokuchi A., Washio T., Motoda H.,An apriori-based algorithm for min- ing frequent substructures from graph data. In Principles of Data Mining and Knowledge Discovery, pages 13–23. Springer, 2000.
34. Kuramochi M, Karypis G., Frequent subgraph discovery. In Proceedings of the 2001 IEEE International Conference on Data Mining, ICDM 2001, pages 313–320. IEEE, 2001
35. Kuramochi M., Karypis G., Finding frequent patterns in a large sparse graph. Data Mining and Knowledge Discovery, 11(3):243–271, 2005.
36. Holder L.B., Cook D.J., Djoko S., Substucture discovery in the subdue system. In AAAIWorkshop on Knowledge Discovery in Databases, KDD-94, pages 169–180, 1994.
37. Krishna V., Suri N., Athithan G., A comparative survey of algorithms for frequent subgraph discovery. Current Science(Bangalore), 100(2), 190-198, 2011.
38. Lu W., Chen G., Tung A.K.H., Zhao F., Efficiently extracting frequent sub- graphs using mapreduce. In 2013 IEEE International Conference on Big Data, pages 639–647. IEEE, 2013.
39. Barney B., Livermore L., Introduction to Parallel Computing, National Laboratory, https://computing.llnl.gov/tutorials/parallel_comp/#Concepts (01.01.2016)
40. Microsoft Corp., https://azure.microsoft.com/en-us/pricing/details/virtual-machines/ (01.05.2016)
41. Fedora Org., https://getfedora.org/en_GB/workstation/ (01.01.2016)
42. Eclipse Org., https://eclipse.org/downloads/ (01.01.2016)
43. Valgrind Org., http://valgrind.org/ (01.01.2016)
44. Intel Corp., https://software.intel.com/en-us/intel-vtune-amplifier-xe (01.01.2016)
45. Thoma M., Cheng H., Gretton A., Han J., Kriegel H.P., Smola A., Song L., Yu P.S., Yan X., Borgwardt K.M. Discriminative frequent subgraph mining with optimality guarantees. Statistical Analysis and Data Mining. The ASA Data Science Journal; 3: 302–18, 2010.
46. Wikipedia Org., https://en.wikipedia.org/wiki/Data_structure_alignment (01.01.2016)
47. http://www.songho.ca/misc/alignment/dataalign.html, (01.01.2016)

48. Horton I. Working with fundamental data types. In: Anglin S, lead editor. Beginning C++. New York, NY, USA: Apress Press, pp. 55–77, 2014.
49. Microsoft Corp., https://msdn.microsoft.com/en-us/library/ms253935, (01.01.2016)
50. Microsoft Corp., https://blogs.msdn.microsoft.com/ce_base/2008/06/27/when-not-to-pack-structures/ (01.01.2016)
51. Intel Corp., https://software.intel.com/en-us/articles/coding-for-performance-data-alignment-and-structures 01.01.2016
52. Gnu Org, https://gcc.gnu.org/onlinedocs/gcc-4.8.5/gcc/Structure-Packing-Pragmas.html (01.01.2016)
53. Bader D.A., Meyerhenke H., Sanders P., Wagner D., Benchmarking for graph clustering and partitioning. Encyclopedia of Social Network Analysis and Mining, pp. 73–84, 2012
54. Wikipedia Org., https://en.wikipedia.org/wiki/Amdahl%27s_law (01.01.2016)
55. Yang L.T., Guo M., High-performance computing: paradigm and infrastructure. Vol. 44. John Wiley & Sons, 2005.
56. Hager G., Wellein G., Introduction to high performance computing for scientists and engineers. CRC Press, 2010.
57. Barney B., Livermore L., OpenMP, National Laboratory https://computing.llnl.gov/tutorials/openMP/#DO (01.05.2016)
58. Intel Corp., https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads (01.01.2016)
59. OpenMP Org., http://openmp.org/mp-documents/openmp-examples-4.0.2.pdf (01.01.2016)
60. Oracle Corp., http://docs.oracle.com/cd/E19059-01/stud.10/819-0501/2_nested.html (01.01.2016)
61. Intel Corp., https://software.intel.com/en-us/articles/performance-obstacles-for-threading-how-do-they-affect-openmp-code (01.01.2016)
62. Jin H., Jespersen D., Mehrotra P., Biswas R., Huang, L., Chapman B., High performance computing using MPI and OpenMP on multi-core parallel systems. Parallel Computing, 37(9), 562-575, 2011.

# ÖZGEÇMİŞ

Murat Oğuz, 1977 yılında İstanbul/Şişli'de doğdu. İlk ve ortaöğrenimini sırasıyla Yıldıztepe İlkokulu ve Bağcılar Teknik Lisesi Bilgisayar Bölümünde tamamlamıştır. 1995-2000 yılları arasında Dokuz Eylül Üniversitesi Bilgisayar Mühendisliği bölümünde lisans eğitimini tamamlamış ardından özel sektörde çalışmaya başlamıştır. Armada Eğitim Merkezi A.S. , Havelsan A.S., Mavi Akademi Eğitim Merkezi A.S, Anka Eğitim Merkezi A.S., Simetri Yazılım Evi A.S. gibi değişik şirketlerde sistem yöneticiği ve eğitmenlik yaptıktan sonra, 2004 yılında Microsoft Türkiye'de işe başlamıştır ve halen çalışmaya devam etmektedir.

Çalışma yaşamına devam ederken, Haliç Üniversitesi'de Bilgisayar Mühendisliği yüksek lisansını tamamlamış ardından Maltepe Üniversitesi Bilgisayar Mühendisliği'nde doktora yapmaya başlamıştır.