T.C.
SÜLEYMAN DEMİREL ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

YEŞİL YAZILIMDA ENERJİ TÜKETİMİNİ DÜŞÜRMEYE YÖNELİK KOMBİNATORYAL BİR ALGORİTMA GELİŞTİRİLMESİ

İbrahim ŞANLIALP

Danışman
Prof. Dr. Tuncay YİĞİT

II. Danışman
Doç. Dr. Muhammed Maruf ÖZTÜRK

DOKTORA TEZİ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI
ISPARTA - 2022

İÇİNDEKİLER

ÖZET

Doktora Tezi

YEŞİL YAZILIMDA ENERJİ TÜKETİMİNİ DÜŞÜRMEYE YÖNELİK KOMBİNATORYAL BİR ALGORİTMA GELİŞTİRİLMESİ

İbrahim ŞANLIALP

Süleyman Demirel Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı

Danışman: Prof. Dr. Tuncay YİĞİT

II. Danışman: Doç. Dr. Muhammed Maruf ÖZTÜRK

Kodu yeniden düzenleme, kaynak kodlarında iyileştirmeler yapmak için uygulanan, zaman alan ve yoğun çaba gerektiren bir süreçtir. Son yıllarda yazılım mühendisliği geliştiricileri arasında oldukça ilgi görmüştür. Geliştiriciler için, seçilen yeniden düzenleme tekniğine bağlı olarak yazılım tarafından tüketilen enerjideki değişimi bilmek önemlidir. Bu tez çalışması kod yeniden düzenleme tekniklerinin etkileşimlerinin incelenmesi ve enerji verimliliği için optimal sıranın elde edilmesine yönelik araştırmaların yapılması üzerinedir.

Çalışma kapsamında Prim tabanlı önceliklendirme algoritması, matematiksel model ve Windows tabanlı grafiksel kullanıcı arayüz (GUI) uygulaması geliştirilmiştir. Uygulama öncelikle verilen statik kod metriklerini çıkartmakta ve hangi tür yeniden düzenleme tekniklerinin kaynak kodla uyumlu olduğuna karar vermektedir. Sonrasında geliştirilen kompleks metriğin ölçümünü kullanan Prim tabanlı önceliklendirme algoritması ile yeniden düzenleme tekniklerinin sırasını üretmektedir. Son olarak, algoritmanın ürettiği öncelik sırası uygulandıktan sonra elde edilen yeniden düzenlenmiş kaynak kod ile orijinal kod enerji tüketimi açısından karşılaştırılır.

Sonuçlar, kodun karmaşıklığını ve boyutunu gösteren kriterlerin enerji tüketimi açısından bir öncelik belirleme algoritması tasarlamak için yararlı olduğunu göstermektedir. Yeniden düzenleme teknikleri sırasının esas olarak yararlanılacak yazılım projesinin kaynak koduna bağlı olduğu sonucu çıkarılmıştır. Elde edilen bulgular geliştiricilerin nesne yönelimli programlama dili ile oluşturulan kodlarını yalnızca enerji verimliliği açısından değil, aynı zamanda sürdürülebilirlik açısından da geliştirmelerine yardımcı olacaktır.

Anahtar Kelimeler: Yeniden düzenleme, önceliklendirme, yeşil yazılım, yazılım metriği, enerji tüketimi, yazılım kalitesi, nesne yönelimli programlama.

2022, 129 sayfa

ABSTRACT

Ph.D. Thesis

DEVELOPMENT OF A COMBINATORIAL ALGORITHM TO REDUCE ENERGY CONSUMPTION IN THE GREEN SOFTWARE

İbrahim ŞANLIALP

Süleyman Demirel University
Graduate School of Natural and Applied Sciences
Department of Computer Engineering

Supervisor: Prof. Dr. Tuncay YİĞİT

Co-Supervisor: Asst. Prof. Dr. Muhammed Maruf ÖZTÜRK

Code refactoring is a time-consuming and effort-intensive process that is applied to make improvements to source code. In recent years, it has gained more interest among software engineering developers. For developers, it is essential to know the change in the energy consumed by the software dependent on the chosen refactoring technique. This thesis is about examining the interactions of code refactoring techniques and conducting research on obtaining the optimal order for energy efficiency.

The scope of the study, Prim-based prioritization algorithm, mathematical model, and Windows-based graphical user interface (GUI) application are developed. Firstly, the application extracts the given static code metrics and decides which refactoring techniques are compatible with the source code. Then, it produces a refactoring sequence thanks to the Prim-based prioritization algorithm using the measurement of the developed complex metric. Finally, refactored source code obtained after applying the priority order produced by the algorithm is compared with the original code in terms of energy consumption.

The results show that criteria indicating the complexity and size of the code are useful for designing a prioritization algorithm in terms of energy consumption. It is concluded that the order of refactoring techniques mainly depends on the source code of the software project to be exploited. The findings will help developers improve their code created with the object-oriented programming language, not only in terms of energy efficiency but also in terms of sustainability.

Keywords: Refactoring, prioritization, green software, software metrics, energy consumption, software quality, object-oriented programming.

2022, 129 pages

TEŞEKKÜR

Doktora tez çalışmamda beni yönlendiren, karşılaştığım zorlukları bilgi ve tecrübeleri ile aşmamda yardımcı olan değerli danışman hocalarım Prof. Dr. Tuncay YİĞİT ve Doç. Dr. Muhammed Maruf ÖZTÜRK'e teşekkürlerimi sunarım.

Ayrıca her çalışmamda olduğu gibi bu tez çalışmasında da maddi ve manevi desteğiyle hep yanımda olan sevgili eşime, hayatın her aşamasında beni yalnız bırakmadıkları gibi tezimin her aşamasında da beni yalnız bırakmayarak bugünlere gelmemi sağlayan aileme sonsuz sevgi ve saygılarımı sunarım.

İbrahim ŞANLIALP

ISPARTA, 2022

ŞEKİLLER DİZİNİ

Sayfa

ÇİZELGELER DİZİNİ

SİMGELER VE KISALTMALAR DİZİNİ

G	Graf
D	Altgraf
G_P	İndüklenmiş altgraf
F	Döngüsel olmayan altgraf
E	Grafın ayrıt sayısı
N	Grafın düğüm sayısı
P	Ayrık bağlı bileşen sayısı
e_k	Düğüm çifti
e^*	Minimum ağırlıklı kenar
V	Gerilim (voltaj)
I	Elektrik akımı (amper)
T	Zaman
V_c	Kararsız olmayan kaynak voltajı
P_d	Dinamik güç
C	Kapasitans
A	Aktivite faktörü
U	Arama uzayı
u	Düğüm ağırlığı
f_i	Amaç fonksiyonu
h_j	Maliyet fonksiyonu
g_k	Doğrusal olmayan fonksiyon
V	Programın hacmi
n	Programın kelime hazinesi
w	Ağırlık fonksiyonu
A_G	Komşuluk matrisi
M_G	Bitişiklik matrisi
R_s	Direnç
J	Joule
CPU	Merkezi işlemci birimi
GPU	Grafik işlemci birimi
$SDLC$	Yazılım geliştirme yaşam döngüsü
LOC	Kod satır sayısı
DIT	Kalıtım ağacı derinliği
CC	Çevrimsel karmaşıklık
$SLOC-L$	Mantıksal kod satır sayısı
MI	Bakım yapılabilirlik indeksi
NOM	Fonksiyon sayısı
MYA	Minimum yayılan ağaç
SI	Uluslararası birimler sistemi
SC	Kaynak kod
GUI	Grafiksel kullanıcı arayüz
GT	Grafik teknolojileri

1. GİRİŞ

Günümüzde enerji tüketimi ve enerjiye bağımlılık tüm dünya ülkelerini ilgilendiren güncel konulardan biridir (Tolga ve Turgut, 2018). Enerji tüketiminin giderek artması sera etkisi yapan gazların atmosferdeki miktarında önemli artışlar yapmaktadır. Bu gazların atmosfere salınımının artması sonucunda küresel ısınma ve küresel iklim değişiklikleri meydana gelmektedir. Atmosferde meydana gelen artışlara en büyük katkıyı yapan sera gazı karbondioksit gazıdır ve karbondioksit merkezli toplam sera gazı emisyonu hesaplanmasında karbon ayak izi kullanılır (Civelekoğlu ve Bıyık, 2020). Karbon ayak izi işlemlerin yaşam boyunca çevreyi kirleticilikleri ile ilgili olup insan faaliyetlerinin çevreye verdiği zararın birim karbondioksit veya karbon cinsinden ölçülen miktarı olarak tarif edilmektedir (Yılmaz, 2014).

Dünyada artan populasyon ile birlikte iletişim ve bilgi teknolojilerinin kullanımı da artmaktadır. Bu tür teknolojilerin devamlılığını sürdürebilmesi için sağlam altyapılar kullanılması kaçınılmazdır. Altyapı kullanımı sonucunda enerji tüketimi büyük miktarda artmakta ve bunun sonucunda dünyada her geçen gün karbon ayak izlerinin giderek arttığı görülmektedir. Bu yüzden kullanılan enerji tüketiminin azaltılması gerekmektedir (Agarwal vd., 2012).

Genel olarak dünyada enerji tüketimi ile ilgili belirli bir farkındalığın oluşması, yazılım enerji tüketimi ile ilişkili çalışmaların araştırmacıların ilgisini çekmesini sağlamıştır. Uygulayıcılar tarafından yeşil yazılım mühendisliği olarak adlandırılan yaklaşım (Manotas vd., 2016), gelişmekte olan bir paradigmadır ve çevre üzerindeki olumsuz etkileri azaltmak için yeşil özellikli yazılımlar geliştirmeyi amaçlamaktadır. Yazılım mühendisliğinin alt araştırma alanlarından biri olan yeşil yazılım (Shenoy ve Eeratta, 2011), enerji tüketimi konusu kapsamında değerlendirilebilir. Bu kapsamda, kod işlevini bozmadan enerji tüketimini düşürmeye yönelik hem donanımsal hem de yazılımsal çözümler sunan çalışmalar mevcuttur (Hsu ve Kremer, 2003). Donanımsal çözümler genellikle merkezi işlemci birimi (CPU) enerji tüketim gözlemine ve bu tüketim değerinin tahminine veya kullanılan çevre birim elemanlarının ayarlarına yönelik iyileştirmeleri içerir. Yazılım enerji tüketim çalışmaları genellikle mobil programlama noktasında yoğunlaşmıştır. Bunun temel nedeni mobil cihazlardaki sık

şarj ihtiyacı olarak görülebilir. Ancak yazılım ölçeklerinin büyüme hızı ve bulut bilişimdeki gelişmeler, kullanıcıların kişisel bilgisayarlardaki enerji tüketim düzeylerini de yükseltmektedir. Bu yüzden yazılımsal çözümler ise kod işlevini bozmadan yazılım için izlenen yazılım geliştirme süreci, kullanılan yeniden düzenleme tekniği gibi konularda yapılan yenilik ve geliştirmeleri gözlemlemeye dayanır.

Yazılım geliştirme süreçlerinde önerilen iyileştirmeler ile yazılım enerji tüketimi düşürülmeye çalışılmıştır. Konuyla ilgili olarak çevik yöntemlerdeki yenilikler göze çarpmaktadır. Bu yapılırken, özellikle CPU veya grafik işlemci birimi (GPU) üzerindeki işlem yükü azaltılmaya çalışılmıştır. İlgili işlemler; etkili algoritma geliştirmeyi, çoklu-iş parçacığı yapısını, veri akışı üzerindeki düzenlemeleri, kod vektörleştirmeyi ve ön bellek yapısının etkili kullanılmasını kapsar. Bunların dışında yeniden düzenleme teknikleri kullanılarak kodun işlevi değiştirilmeden yapılan işlemler ile de enerji tüketimi azaltılabilmektedir.

1.1. Motivasyon

Yazılım mühendisliğinde yazılım maliyet bütçesinin yüzde ellisinden fazlası bakım için ayrılmaktadır (Turver ve Munro, 1994; Ahn vd., 2003). Bu bakım maliyeti çıkan hatalar ile ilgili olmakla birlikte yazılım özelliklerinin yenilenmesi ve güçlendirilmesi ile de ilgilidir. Özellikle mobil cihazlarda ve yüksek düzeyli bilgisayarlarda enerji tüketimi, yönetilmesi ve kontrol edilmesi gereken bir unsurdur. Yazılım enerji tüketiminin düşürülmesi hem hesaplama karmaşıklığı yükünü hafifletmekte hem de yeşil yazılım olarak adlandırılan konuda ilerlemeyi sağlamaktadır. Bu noktada, yazılım enerji tüketimine yönelik her çalışma yazılım ekonomisi kapsamında değerlendirilebilir. Yazılım ekonomisini daha iyi yönetmeye yardımcı olan enerji tüketimi konusu daha çok mobil yazılımlar için vazgeçilmezdir. Ancak yazılım mimarileri ve çevre birimlerindeki değişim ile artan depolama ihtiyacı, hesaplama sistemlerinin daha fazla enerji tüketmesine neden olmaktadır.

Kod yapısına ve yazılım geliştirme modeline göre enerji tüketim ipuçları belirleyen çalışmalar literatürde mevcuttur. Ancak bu popülariteye rağmen sürekli değişen ve gelişen yeniden düzenleme tekniklerinin enerji tüketimi açışından tek tek incelendiği

çalışmalar da mevcuttur. Nitekim geliştiriciler açışından seçilen yeniden düzenleme tekniğine bağlı olarak yazılımın tükettiği enerjinin değişimini bilmek önemlidir (Park vd., 2014). Dahası, yeniden düzenleme etkinliğinin uygulamalardaki farklılığa bağlı olarak tutarlı olmadığı tespit edilmiştir (Sahin vd., 2014). Bu da deney veri seti ölçeğinin olabildiğince geniş olması gerektiği sonucunu vermektedir. Bunun dışında, bir veya iki yeniden düzenleme tekniğinin kullanımının enerji tüketimini ne kadar azalttığı da daha önce araştırılmıştır (Kim vd., 2018). Kim ve arkadaşları, yedi farklı yeniden düzenleme tekniği ile yapılan kombinasyonları gömülü sistem kodları üzerinde denemişlerdir. Yeniden düzenleme tekniklerini bireysel uygulamak yerine ikili kombinasyon şeklinde çalıştırmak enerji tüketimini daha çok düşürmüştür. Ancak çalışmanın kapsamı üç farklı kombinasyon ile sınırlandırılmıştır. Ayrıca üç ve daha fazla sayıda kombinasyon göz önüne alınmamış, ikili kombinasyonlar benzerliğe dayalı üretilmiştir. Diğer taraftan, son zamanlarda araştırmacılar optimum yazılım sürdürülebilirliğini elde etmeye yardımcı olan yeniden düzenleme tekniklerinin sırasını tespit etmeye (Tarwani ve Chug, 2020; Gupta ve Chug, 2020) ve yazılımın sürdürülebilirliği ile enerji tüketimi arasındaki ilişkiyi analiz etmeye odaklanmaktadırlar (Mancebo vd., 2021a). Araştırmalar mobil cihazlarda ve yüksek düzeyli bilgisayarlarda enerji tüketiminin yönetilmesi ve kontrol edilmesi gereken bir unsur olduğunu göstermektedir. Dahası kişisel bilgisayarın yılda yaklaşık 175 kg karbondioksit salınım oluşturduğu göz önüne alındığında kodun enerji tüketimini azaltan yöntemlere odaklanmak anlamlı hale gelmektedir.

1.2. Bilimsel Katkı

Tez çalışması kapsamında, kod yeniden düzenleme tekniklerinin açık kaynaklı nesne yönelimli programlama dilleri ile yazılmış kaynak kodlar üzerindeki enerji etkisi araştırıldı. Yazılım sistemlerinin enerji tüketimini azaltmaya yönelik olarak en iyi yeniden düzenleme tekniklerinin sırasını elde eden bir önceliklendirme algoritması ve otomasyon yazılımı geliştirildi. Yazılım metrikleri ve grafik tabanlı arama yaklaşımı, metrik değerler yardımıyla düğüm maliyetini hesapladıktan sonra yeniden düzenleme sırasını bulmak için kullanılır. Bu sıra, yazılım sürdürülebilirliği ve enerji açısından en uygun sıra olarak kabul edilir. Tez ile ilgili çalışmalar nesne yönelimli programlama dillerine ait bulguları kapsayacaktır. Bunun dışındaki programlama yapıları örneğin endüstriyel otomasyon yazılımları ve mantıksal programlama tezin kapsamı dışındadır.

Tezin katkıları şu şekilde sıralanabilir: 1) Yeniden düzenleme tekniklerine öncelik vermek için bir kompleks metrik tasarlanmıştır. 2) Yeniden düzenleme tekniklerinin optimum sırasını elde etmek için Prim tabanlı bir algoritma geliştirilmiştir. 3) Bu çalışma ayrıca statik kod analizini ve önerilen yöntemi içeren bir otomasyon yazılımı sunmaktadır. 4) Bulgular, geliştiricilerin nesne yönelimli kodlarını yalnızca enerji verimliliği açısından değil, aynı zamanda sürdürülebilirlik açısından da geliştirmelerine yardımcı olacaktır.

2. KAYNAK ÖZETLERİ

Opdyke ve Johnson (1993) yaptıkları çalışmada, nesne yönelimli programlamada bir dizi somut sınıfın soyut üst sınıfını yapmak için yeniden düzenleme diye adlandırılan bir tür yapı iyileştirme dönüşümü üzerine odaklanmışlardır. Programları aşamalı olarak yeniden yazma ve yapılarını iyileştirme teknikleriyle ilgilenmişlerdir. Ayrıca nesne yönelimli dillerde yeniden düzenleme işlemlerinin otomasyonunun olmaması dolayısıyla maliyetin yüksek olduğunu ifade etmişlerdir.

Tiwari vd. (1994) çalışmalarını kod derleme sürecinin bir mikroişlemci üzerinde çalışan yazılımın enerji tüketimini nasıl etkileyebileceği üzerine yapmışlardır. Geleneksel derleme tekniklerinin enerji azaltımı üzerindeki etkisini tartışmışlar ve bu konuda faydalı olabilecek teknikleri gözden geçirmişlerdir. Switch işlemini azaltmak için komutları tekrardan düzenleme işleminin enerji tüketimini azaltacağını ifade etmişlerdir. Ayrıca en az içerik kaydedecek şekilde fonksiyon çağrılarını düzenleyen derleyici politikaları kullanmak hafıza operandlarının kullanım sayısını düşüreceği için enerji tasarrufu sağlanabileceğini vurgulamışlardır. Yazarlara göre dinamik Ram'de (sayfalama modu) meydana gelen sayfa kayıpları enerji tüketimini artırmaktadır. Bu yüzden bellek erişimi ve tahsisi yeniden düzenlenerek sayfa isabet (page hit) oranının yükseltilmesi ile enerji tasarrufunun sağlanacağını belirtmişlerdir.

Moore (1996) nesne yönelimli programların çoğunun kusurlu biçimde kalıtım hiyerarşileri ve kusurlu olarak ayrışmış yöntemler içerdiğini ve bu kusurlar bakımla artma eğiliminde olduğunu ifade etmiştir. Yaptığı çalışmasında, kalıtım hiyerarşilerinin ve programlardaki yöntemlerin otomatik olarak yeniden yapılandırılması için Guru diye adlandırdığı bir prototip aracı sunmuştur. Yeniden düzenleme sürecinin otomasyonuna yönelik katkılar sağlamıştır.

Demeyer vd. (2000) yaptıkları çalışmada, nesne yönelimli yazılım uygulamasındaki yeniden düzenleme tespitini tersine mühendislik çabalarıyla tespit etmeye çalışmışlardır. Uygulamada nerede değişiklik olduğunu bulabilmek için kod değişiklik metriklerini (sınıf sayısı, metot sayısı, her sınıf için kalıtım sayısı) kullanmışlar ve sezgisel tarama ile yazılım sisteminin ilgili bölümlerine odaklanmışlardır.

Tokuda ve Batory (2001) nesne yönelimli tasarımların yeniden düzenleme teknikleri ile evrimleştirilmesi konusunda çalışmışlardır. İnceledikleri uygulamanın, yazılım mühendisleri tarafından manuel olarak geliştirildiğini; fakat uyguladıkları yeniden düzenleme teknikleri ile daha hızlı ve daha ucuz bir şekilde üretilebileceğini göstermişlerdir. Ayrıca nesne yönelimli programlamalardaki ilerlemelerin yeniden düzenleme tekniklerinin artmasını sağladığından bahsetmişlerdir.

Kataoka vd. (2002) yaptıkları çalışmada, yeniden düzenlemenin sürdürülebilirlik artırıcı etkisini ölçmek için nicel bir değerlendirme yöntemi önermişlerdir. Yeniden düzenlemenin etkisini değerlendirmek için birleştirici metrikler üzerine odaklanmışlardır. C++ ile yazılmış en fazla 5 yıl süre sürdürülebilirliği olduğunu ifade ettikleri yazılım üzerinde birleştirici diye ifade ettikleri metrikler göz önüne alınarak yeniden düzenleme tekniklerini uygulamışlar ve yeniden düzenlemenin yazılımın sürdürülebilirliğine etkisini gözlemlemişlerdir.

Mens ve Tourwé (2004), yeniden düzenleme ile yazılımın yeniden yapılandırılması alanında mevcut araştırmalara kapsamlı bir bakış sunmuşlardır. Bu araştırmada, bir dizi farklı kriterlere dayanarak (özel teknikler, faaliyetleri desteklemek için kullanılan formalizmler vs.) karşılaştırılmalar yapılmış ve yeniden düzenlemenin yazılım sürecine etkisi tartışılmıştır.

Higo vd. (2004) yaptıkları çalışmada, bir kaynak dosya içerisinde, diğerine özdeş veya benzer bir kod parçacığı olarak adlandırılan kod klonlarını kaldırmak için "Extract Method" ve "Pull Up Method" yeniden düzenleme tekniklerini de kullanan bir yöntem sunmuşlardır. Çalışmaları sonucunda uyguladıkları kaynak dosyada 2 adet klonlanmış sınıf tespit etmişler ve uyguladıkları yeniden düzenleme teknikleri ile kodun fonksiyonelliğini değiştirmeden klonlanmış kodları indirgemişlerdir.

Binkley vd. (2006) yaptıkları çalışmada, nesne yönelimli programlama (OOP) ile geliştirilen programın bağlam yönelimli (aspect-oriented) programlamaya (A0P) transferi sorununa otomatik bir yaklaşım sunmuşlardır. Nesne yönelimli programladan bağlam yönelimli programlamaya transfer için AOP-Migrator isimli bir araç kullanmışlardır. 6 adet yeniden düzenleme tekniği tanımlanmış ve bu araca otomatik

olarak verilmiştir. Çalışma sonuçları, transferin büyük ölçüde otomatik şekilde OOP'den AOP'ye geçmenin mümkün olduğunu fakat yeniden düzenleme tekniklerin her zaman doğrudan uygulanabilir olmadıklarını göstermiştir.

Taina ve Pohjalainen (2009) çalışmalarını yeşil yazılımın mihenk taşlarından olan "yeşil metrikler" üzerine yapmışlardır. İklim değişikliğine karşı savunmada yazılımların önemli rol oynayacağı görüşünü belirtmişlerdir. Sera gazı salınımını azaltmak için yazılımlarda üretilen karbon ayak izlerini ve işlemcilerin cycle kullanımı ile tüketilen enerjilerini vurgulamışlardır.

Taina (2010) çalışmasında, yazılım karbon ayak izlerini analiz etmek için bir yaklaşım sunmuştur. Sunulan yaklaşım tipik bir yazılım yaşam döngüsünü adım adım analiz ederek her adımın ne kadar büyük karbon ayak izleri üretildiğini tahmin etmektedir. Yeşil yazılım açısından bakıldığında, yazılımın nasıl geliştirileceği, nasıl dağıtılacağı ve nasıl kullanılacağının önemli olduğunu vurgulamıştır. Yeşil olduğunu iddia eden bir yazılım satıcısının yazılımının küçük bir karbon ayak izi olduğunu göstermesi gerektiğini ifade etmiştir.

Silva vd. (2010) yaptıkları çalışmada, "inline refactoring" yöntemini Java dili ile yazılmış gömülü yazılım için geliştirilmiş uygulamaların kodlarına uygulamışlardır. Yöntemin performans ve enerji tüketimi üzerindeki etkilerini incelenmişlerdir. Address Book, Sokoban ve Mpeg Decoder uygulamalarını kullanmışlardır. Yöntemin Addres Book ve Sokoban uygulamalarında performansın arttığını ve enerji tüketiminin azaldığını gözlemlemişlerdir. Fakat en karmaşık analiz uygulaması olan Mpeg Decoder aynı sonuçları vermemiştir. Çünkü uygulama çok karmaşık yöntemler içermesinin yanı sıra yöntemlerin değişken havuzunda adreslenecek çok sayıda değişkene sahip olması durumunda bir kazanca yol açmadığını ifade etmişlerdir.

Naumann vd. (2011) yaptıkları çalışmada, "Yeşil ve Sürdürülebilir Yazılım" ve "Yeşil ve Sürdürülebilir Yazılım Mühendisliği" terimlerinin tanımlarını önermişler, daha sonra kavramsal referans model olan GreenSoft Modeli'nden ana hatlarıyla bahsetmişlerdir. Bu model yazılım ürünleri için ekonomik, toplumsal ve ekolojik etkilerin ve tüm yaşam

döngüsü boyunca üründen kaynaklanan insan üzerindeki etkilerinin mümkün olduğunca küçük olması gerektiği anlamına gelmektedir.

Shenoy ve Eeratta (2011) çalışmalarında, yeşil yazılım ve geliştirme sürecinde çevre üzerindeki zararlı etkileri en aza indirmek için yazılım geliştirme aşamalarında iyileştirmeler önermektedirler. Ayrıca daha düşük karbon emisyonlarına, güç, kâğıt kullanımına yol açabilecek ve kuruluşların daha çevreci ve sürdürülebilir yazılım geliştirmeye doğru ilerlemesine yardımcı olabilecek uygun yeşil yazılım parametreleri sunmaktadırlar.

Chen ve Wang (2012) yaptıkları çalışmada, grafiksel kullanıcı arayüz test scriptleri içerisindeki problemli bölümler diye tanımlanan kötü kokuları tespit etmişler ve bunlara yeniden düzenleme teknikleri uygulamışlardır. Çalışma sonucunda kötü kokuları gidermede yeniden düzenleme tekniklerinin yararlı olduğunu gözlemlemişlerdir.

Gottschalk vd. (2012) çalışmalarını çevresel kaynakları korumanın yanında bilgi ve iletişim teknolojisinin kabul edilebilir bir enerji tüketimini sürdürmek için mobil cihazların enerji tüketimi üzerine yapmışlardır. Mobil cihazlardaki uygulama yazılımlarının kaynak kodları üzerinde tespit edilen kötü kokuları yeniden düzenleme teknikleri ile optimize etmişler ve böylece uygulamanın enerji tüketimini azaltmışlardır. Ayrıca kötü koku çeşitlerinden olan “loop bug”, “dead code”, “moving too much data”, “immortality bug” için çözüm önerileri sunmuşlardır.

Agarwal vd. (2012) yaptıkları çalışmada, yazılımın sürdürülebilirlik yönlerini ele almışlardır. Ayrıca Yazılım Geliştirme Yaşam Döngüsü (SDLC) modelleri, yazılımın ve geliştirme sürecinin sürdürülebilirliği ve daha yeşil yönlerini ele almadıklarını veya bunların üzerinde durmadıklarını vurgulamışlardır. Yazılım geliştirmenin daha yeşil yönlerini de içerecek şekilde SDLC modellerinde iyileştirmeler üzerine odaklanmışlardır.

Mahmoud ve Ahmad (2013) yaptıkları çalışmada, yazılım mühendislerinin yazılım geliştirme sürecine yardımcı olmak ve yazılımın çevre güvenliğini sağlamak için sürdürülebilir bir yazılım modeli oluşturma üzerine odaklanmışlardır. Mühendislik

sürecinde her bir yazılım aşaması için yeşil yönergeler veya yeşil süreçler önermişlerdir. Böylece yeşil ve sürdürülebilir bir yazılıma ulaşmayı hedeflemişlerdir.

Dick vd. (2013) yaptıkları çalışmada, yazılımda yeşil ve sürdürülebilir geliştirmeleri Scrum'a entegre ederek, genel bir süreç modeli sunmuşlardır. Böylece Scrum çevik geliştirme modelinde geliştirme yapılarak enerji tüketiminin düşürülmesini amaçlamışlardır. Ayrıca, geliştirilen model ile yazılım mühendislerinin yeşil ve sürdürülebilir yazılım ürünleri geliştirmede desteklenmesi öngörülmüştür.

Gottschalk vd. (2013), çalışma kapsamında Android uygulamaları için güç tüketimi konusunu ele almışlardır. Bir mobil uygulama için güç atığı koşullarını tanımlamışlar ve sonra atık koşullarını çözmek için bir kod yeniden düzenleme tekniği önermişlerdir. Ayrıca önerilen tekniğin enerji verimliliğini arttırmaya uygun olduğunu belirtmişlerdir. Ancak detay iyileştirme sürecini başkalarının kullanımına yetecek kadar açıklanmamıştır.

Kwon ve Tilevich (2013), yaptıkları çalışmada, enerji tüketimini azaltmak için mobil uygulamaların bakımına yönelik adaptive multi-target cloud offloading adını verdikleri yeni bir yaklaşım sunmuşlardır. Cloud offloading, uygulamanın mobil cihazının pilini boşaltmadan uygulamanın enerji yoğun işlevselliğini yürütmeyi mümkün kılan bir mobil uygulama optimizasyon tekniğidir. Kaynak kodlarını değiştirmeden mobil uygulamaların enerji tüketimini işlemci ve ağ iletişimi (Wi-fi, 4G) açısından incelemişlerdir. Gözlemleri sonuçlarının umut verici olduğunu bildirmişlerdir.

Dinita vd. (2013) yaptıkları çalışmada, bir bulut içinde sanallaştırılmış sunucuların mobilitesi aracılığıyla internet / ağ kullanımının optimizasyonuna ve donanım güç tüketimini geliştirmeye odaklanmışlardır. Enerji tüketimine yönelik farklı işlemci yük oranları ile testler yapmışlar ve hızın kritik olmadığı durumlarda clock hızını azaltmanın enerji tüketimini düşürdüğünü tespit etmişlerdir.

Park vd. (2014) çalışmalarında, mevcut yeniden düzenleme tekniklerinin verimli enerji yazılım üretimini destekleyip desteklemediğini araştırmayı amaçlamışlardır. Mevcut teknikler tarafından üretilen yeniden düzenlenmiş kodların orijinal kodlardan daha fazla

güç tüketebileceklerini çünkü yeniden düzenleme süreçlerindeki güç tüketimini dikkate almadıklarını belirtmişlerdir. Bu yüzden, Martin Fowler'ın 63 adet yeniden düzenleme tekniği için güç tüketimini hesaplamışlar ve analiz etmişlerdir. Sonuçta, bunlar arasında 33 tekniğin verimli enerji yeniden düzenleme tekniği olduğu tespit edilmiştir.

Parsai vd. (2015) yaptıkları çalışmada, yazılım test kodlarına yeniden düzenleme teknikleri uyguladıktan sonra test kodlarının davranışlarına yönelik mutasyon testleri uygulamışlardır. Mutasyon testleri, yazılıma test öncesi hatalar yükleyip sonrasında testin bu hataların ne kadarını yakaladığına yönelik bir tür test tekniğidir. Çalışmalarının, test kodları üzerinde uygun olmayan şekilde yapılan yeniden düzenleme kısımlarını da aydınlatmaya yönelik olduğunu belirtmişlerdir.

Xuan vd. (2016), yazılım test kodlarındaki dinamik analiz işlemlerinin geliştirilmesine yönelik yeni bir yeniden düzenleme yaklaşımı üzerine çalışmışlardır. B-Refactoring olarak adlandırdıkları, test edilebilirliğini değiştirmeden test durumlarını daha küçük test durumlarına bölme prensibine dayanan bu teknik ile dinamik analiz için daha iyi kontrol akışı sağlamayı hedeflemişlerdir.

Manotas vd. (2016), yazılım uygulayıcıların yazılımlarını geliştirirken gereksinimleri belirledikleri sırada, tasarım yaptıkları sırada, test ettikleri sırada ve yazılımlarına bakım yaptıkları sırada enerji kavramı hakkında ne düşündükleri üzerine yapılan ilk deneysel çalışmayı sunmuşlardır. Deney Google, Microsoft, ABB ve IBM çalışanlarından oluşan 464 kişilik yazılım uygulayıcılarından oluşmaktadır. Toplanan verilerden elde ettikleri sonuçlar, mevcut yeşil yazılım mühendisliği araştırmasıyla bağdaşmakta ve uygulayıcıların uygulamalarının enerji kullanımını iyileştirmelerine yardımcı olacak stratejiler ve araçlar geliştirmeyi amaçladıklarını göstermektedir.

Morales vd. (2017) yaptıkları çalışmada, An Energy-Aware Refactoring Approach for Mobile Apps isimli enerji tüketimini kontrol ederken mobil uygulamaları için kod yeniden düzenlemeye yönelik yeni bir model (anti-desen) doğrulama yaklaşımı önermişlerdir. Modeli evrimsel çokamaçlı optimizasyon algoritmaları kullanarak test etmişlerdir. Bu yaklaşım, enerji tüketimini düşürebilecek yeniden düzenleme

tekniklerini önermektedir. Çalışmaları sonucunda varsayılan ayarlarda mobil telefonun batarya ömrünü % 29 artırmayı başarmışlardır.

Barack ve Huang (2018) Greenup, Powerup ve Speedup metriklerini kullanarak bir mobil yazılım sistemindeki kod yeniden düzenleme tekniklerinin enerji verimliliği ve performans açısından değerlendirmesine yönelik bir çalışma sunmuşlardır. Fowler'ın çalışmalarını kullanarak örnek kodu çalıştıran bir Android uygulaması geliştirmişler ve mobil uygulama yazılımına 21 kod yeniden düzenleme tekniğini uygulamışlardır. Akabinde, kod yeniden düzenleme tekniğini on kategoriden birine sınıflandırmak için sonuçlara önerilen metrikleri uygulamışlardır. İlk 5 kategori enerji tüketimini düşüren yeşil alanı sonraki 5 ise enerji tüketimini artıran kırmızı alanı temsil etmektedir. Yeşil alanda 9 yeniden düzenleme tekniği ve kırmızı alanda 12 yeniden düzenleme tekniği olacak şekilde sınıflandırmışlardır. Elde edilen sonuçlar, her kod yeniden düzenleme tekniği için performans ve enerji verimliliği arasındaki karşılıklı ilişkiyi göstermektedir.

Kim vd. (2018) çalışmalarında, gömülü sistemlerden elde edilen kodlarda yedi farklı yeniden düzenleme tekniğinin ve bunların bazı ikili kombinasyonlarının enerji tüketimindeki etkisini araştırmışlardır. % 2.4 oranında enerji tüketiminin düşürülebildiği çalışmanın tek programlama dilindeki kodlar üzerinde yapıldığı görülmektedir. Çalışmada vurgulanan önemli noktalardan biri yeniden düzenleme tekniklerinin farklı kombinasyonlarının da nasıl seçilmesi gerektiğidir. Bunun da bir başka araştırma konusu olabileceği vurgulanmıştır.

Anwar vd. (2019) çalışmalarında, Android açık kaynak uygulamalarının önce tür başına ayrı ayrı ve sonra permütasyonlu bir şekilde beş kod koku türü yeniden düzenlemelerinin enerji tüketimine etkisini araştırmışlardır. Ayrıca yeniden düzenlemeler kullanmanın yerel Android açık kaynak uygulamalarının yürütme süresi üzerindeki etkisini de incelemişlerdir. Sonuç olarak "Yinelenen kod" ve "Tip Kontrolü" için kaydedilen maksimum enerji azaltımının sırasıyla % 10,8 ve % 10,5 olduğu görülmektedir. Kod kokusu yeniden düzenlemelerinin belirli permütasyonları, enerji tüketimi etkileri seçilen Android uygulamaları arasında büyük ölçüde farklılık gösterebileceğinden dikkatli kullanılması yönünde görüş bildirmişler ve seçili Android uygulamalarında yürütme süresinde önemli bir artış veya azalma gözlemlememişlerdir.

Connolly Bree ve Cinnéide (2020) çalışmalarında, temsilci ve kalıtımdan oluşan iki yeniden düzenlemenin bir programın enerji tüketimi üzerindeki etkisini karşılaştırmak için bir kavram kanıtı deneyi gerçekleştirmişler. Aynı programın biri kalıtımla, diğeri temsilciyle uygulanan iki versiyonu yazılmıştır. Kalıtımdan yararlanan program sürümünün çalışma süresinde % 77'nin üzerinde bir azalma sağlanmış ve güç tüketimi, temsilciye kıyasla % 4'ün üzerinde azalmıştır.

Sehgal vd. (2020), yeniden düzenleme işleminin güç kullanımının azaltılmasında nasıl önemli bir role sahip olduğunu araştırmışlardır. Bu çalışma, uygun yeniden düzenleme tekniklerinin uygulanması ile pil tüketimini azalttığını ortaya koymuştur. Sonuç olarak, yeniden düzenleme tekniği ile kodun iyileştirilmesinin, enerji verimliliğine yönelik iyi bir adım olduğu vurgusu yapılmıştır.

Mancebo vd. (2021a), yazılımın enerji tüketimi ile sürdürülebilirliği arasında bir ilişki olup olmadığını test etmek için ampirik bir çalışma gerçekleştirmişlerdir. Bu çalışma bakım yapılabilirlik, kod satırı sayısı ve yazılımın karmaşıklığı gibi farklı metrikler aracılığıyla değerlendirilmiştir. Elde edilen sonuçlar, kod satır sayısının hem işlemcinin enerji tüketimini hem de yazılımın çalıştırıldığı bilgisayarın toplam enerji tüketimini etkilediğini göstermiştir.

3. GENEL BİLGİLER

Yeniden düzenleme kavramı, yeniden düzenleme teknikleri, yazılım metrikleri, metin madenciliği, metin madenciliği alanları, kavramları ve teknikleri, kombinatoryal optimizasyon, graf teorisi, minimum yayılan ağaç ve algoritmaları, enerjinin genel tanımı ve bu araştırmada kullanılan diğer teknikler ve araçlar ile ilgili genel bilgiler bu bölümde açıklanmaktadır. Bahsi geçen kavram ve tekniklerin açıklanmasının ardından, tez kapsamıyla ilgili diğer çalışmalar ele alınmaktadır.

3.1. Yeniden Düzenleme

Yazılım sisteminindeki kodun dış davranışını değiştirmeyecek şekilde iç yapısında iyileştirmelere gidilerek kodu değiştirmek gerekebilir. Kodun dışsal davranışını değiştirmeden sürdürülebilirliğini, okunabilirliğini artırmak ve karmaşıklığını düşürmek için kod üzerinde yapılan değişiklik kod yeniden düzenleme olarak adlandırılmaktadır (Fowler vd., 1999).

Kod yeniden düzenleme, mevcut olan kodun gövdesini yeniden yapılandırarak kodun dış davranışını değiştirmeden iç yapısını değiştirmek için kullanılan disiplinli bir tekniktir. Bu teknik, her bir bilgisayar programının kaynak kodunda işlevsel olmayan gereksinimlere uygunluğunu değiştirmeden ufak değişiklikler içeren bir dizi yeniden düzenleme yoluyla yapılandırma sürecini kapsar (Kaur ve Kaur, 2016). Kısaca, kodun kalitesini artırmak için yazılım sisteminin işlevsel olmayan özelliklerini iyileştirmeye yönelik gerçekleştirilmektedir.

Martin Fowler (Fowler, 2018), yeniden düzenlemenin geliştiricilerin daha hızlı programlama yapmalarına, hataları bulmalarına ve yazılım tasarımını iyileştirmelerine yardımcı olduğunu belirtir. Yeniden düzenleme, kaynak kodun sürdürülebilirliğini iyileştirmek için geliştirilmiş kod okunabilirliği ve azaltılmış karmaşıklığın yanı sıra genişletilebilirliği iyileştirmek için daha etkileyici bir iç dizayn elde etmeyi hedefleyen bir tekniktir. Ayrıca, yazılımın performansı veya güvenilirliği gibi diğer yazılım niteliklerini geliştirmek için de kullanılmaktadır (Kwon vd., 2013; Park ve Hong, 2013). Bu niteliklerin yanında yeni bir yazılım kalite faktörü olarak değerlendirilen enerji

verimliliği açısından da incelenmektedir (Park vd., 2014; Kim vd., 2018; Sehgal vd., 2020). Yazılım mühendisleri sürdürülebilirliği iyileştirmek için eski kodlarını yeniden düzenlediklerinde, yeniden düzenleme, enerji tüketimi açısından orijinal eski koddan daha fazla enerji tüketen herhangi bir kod üretebilir (Park vd., 2014).

Kod üzerinde yeniden düzenleme ile değişiklikler yapılırken dikkat edilmesi gereken en önemli nokta, kodun temeldeki işlevselliğini etkilememesi gerektiğidir. Bu yüzden, yapılan değişiklikler tam anlamıyla kodu kolaylaştırmak içindir. Manuel yeniden düzenleme genellikle hataya açıktır ve zaman alır (Roberts, 1999). Örneğin, bir yöntemi yeniden adlandırmak, yöntemin yeni adının henüz kullanımda olmadığının kontrol edilmesini ve tüm çağrıların güncellenmesini gerektirir. Açıkça zaman alıcı olmasının yanı sıra, bu işlem aynı zamanda hataya da meyillidir çünkü polimorfizm, unutulmuş bir güncellemenin doğru bir şekilde derlenmesine neden olabilir, ancak yazılımın davranışını yanlışlıkla değiştirebilir. Bu, bakımcının değiştirilen işlevselliği manuel olarak bulmasını ve atlanan çağrıyı güncellemesini gerektirir. Böyle durumlarda yeniden düzenleme genellikle gerçekleştirilmez ve yazılımın yapısı, işlevsel değişikliklerin bir sonucu olarak bozulur. Fakat otomatik yeniden düzenleme araçları sayesinde bu tür sorunlar ortadan kaldırılabilir. Çünkü otomatik yeniden düzenleme, ön koşulların kullanılmasıyla mümkün kılınmıştır ve bunların yerine getirilmesi sonucunda, yeniden düzenlemenin davranışları koruyacağını garanti eder (Kaur ve Kaur, 2016).

3.1.1. Yeniden düzenleme teknikleri

Yazılım mühendisliğinde bazı yöntemler programın enerji tüketimini ve performansını doğrudan etkileyebilir (Silva vd., 2010). Bunlardan biri de yeniden düzenleme teknikleridir. Kaynak koda doğru şekilde uygulanan yeniden düzenleme teknikleri, yalnızca kodun kalitesini artırmaz aynı zamanda bir uygulama tarafından tüketilen enerjiyi de etkiler (Papadopoulos vd., 2018). Yeniden düzenleme teknikleri uygulanmış kodlar orijinal kodlara göre enerji tüketimini artırabilir (Kim vd., 2018).

Yazılım geliştiricileri, mühendisleri veya araştırmacıları, açık kaynaklı yazılım geliştirmeyi sürdürmek için daha kararlı yeniden düzenleme teknikleri kullanmak istemektedir. Yeniden düzenleme teknikleri, yazılım endüstrisindeki birçok bilgisayar

programında uygulanabilir. Bu programlar nesne yönelimli dillerle yazılabilir. Nesneye yönelik programlamada kullanılabilecek birçok yeniden düzenleme tekniği vardır. Fakat bazı yeniden düzenleme teknikleri enerji tüketimine olumlu katkı sunarken bazıları ise enerji tüketimini olumsuz etkilemektedir (Park vd., 2014).

Verilen bilgiler doğrultusunda, tez kapsamında toplamda 78 adet yeniden düzenleme tekniğini incelenmiş olup yazılımın enerji tüketimi ve kod performansı açısından önemli görülen 40 adet teknik Çizelge 3.1'de verilmektedir. Çalışma kapsamında tekniklerin birbirleri ile etkileşimleri sonucunda enerji verimliliğine olumlu katkı sunan yeniden düzenleme tekniği sayısı projelere göre değişiklik gösterebilmektedir. Bu doğrultuda, tekniklerden önemli görülenlerin bireysel olarak genel çalışma mantıkları aşağıda detaylı bir şekilde anlatılmaktadır.

3.1.1.1. Simplify nested loop

Çift döngü (iç içe döngü) genellikle iki boyutlu bir diziyi işlemek için kullanılır. Bununla birlikte bu teknik, yuvalanmış döngü yapısını tek bir döngüye dönüştürür (Kim vd., 2018). Döngü bloğu içindeki işlemler, iç içe döngü yapısındaki dizi öğelerine erişmek için dizin kullanan ifadelerle açıklanmış ise dizi işaretçisi ve tek döngü kullanılarak bu iç içe döngü yapısının yerine kullanılabilir (Kim vd., 2018). Şekil 3.1'de iki farklı kod parçası görülmektedir. Şekil 3.1(a) orijinal kod parçasını temsil ederken, Şekil 3.1(b) Simplify Nested Loop uygulanmış kod parçasını göstermektedir.

```
#define sizeM = 100;
#define sizeN = 100;

void main() {
        int i, j;
        int arrayA[sizeM][sizeN];

        for(i = 0; i < sizeM; i++){
            for(j = 0; j < sizeN; j++){
                arrayA[i][j] = 100;
                }
            }
        }
```

(a) Orijinal

```
#define sizeM = 100;
#define sizeN = 100;

void main() {
        int i, j;
        int arrayA[sizeM][sizeN];
        int *p = &arrayA[0][0];

        for(i = 0; i < sizeM*sizeN; i++){
            *p++ = 100;
            }
        }
```

(b) Simplify Nested Loop

Şekil 3.1. Orijinal kod parçası ve simplify nested loop uygulanmış kod parçası (Kim vd., 2018)

Çizelge 3.1. Önerilen yeniden düzenleme teknikleri listesi

Yeniden Düzenleme Teknik Adı
• Simplify Nested Loop
• Inline Method
• Replace Temp with Query
• Introduce Explaining Variable
• Split Temporary Variable
• Inline Temp
• Remove Assignment to Parameters
• Replace Method with Method Object
• Substitute Algorithm
• Hide Delegate
• Move Method
• Move Field
• Extract Class
• Inline Class
• Introduce Foreign Method
• Remove Middle Man
• Introduce Local Extension
• Self Encapsulate Field
• Replace Array with Object
• Replace Magic Number with Symbolic Constant
• Change Reference to Value
• Replace Subclass with Fields
• Encapsulate Field
• Encapsulate Collection
• Decompose Conditional
• Consolidate Conditional Expression
• Consolidate Duplicate Conditional Fragments
• Remove Control Flag
• Rename Method
• Add Parameter
• Remove Parameter
• Separate Query from Modifier
• Parameterize Method
• Replace Parameter with Explicit Methods
• Introduce Parameter Object
• Remove Setting Method
• Pull Up Method
• Pull Up Field
• Push Down Field
• Collapse Hierarchy

3.1.1.2. Inline method

Gereksiz yöntemlerin sayısını en aza indirmek, kodu daha sade hale getirecektir (Refactoring, 2021). Böylece kod tekrarının önüne geçilecek ve kodun sürdürülebilirliği artacaktır. Şekil 3.2'de iki farklı kod parçası görülmektedir. Bu kodlar C# dilinde yazılmış ve aynı işleve sahiptir. Şekil 3.2(a) orijinal kod parçasını temsil ederken, Şekil 3.2(b) Inline Method uygulanmış kod parçasını göstermektedir. Şekil 3.2(b)'deki kod satır sayısı daha azdır.

```
class Poliklinik
{
  //...
  int TalepDoktor()
  {
    return HastaSayisiSiniri()? 2 : 1;
  }
  bool HastaSayisiSiniri()
  {
    return  hastaSayisi> 5;
  }
}
```

(a) Orijinal

```
class Poliklinik
{
  //...
  int TalepDoktor()
  {
    return hastaSayisi> 5 ? 2 : 1;
  }
}
```

(b) Inline Method

Şekil 3.2. Orijinal kod parçası ve inline method uygulanmış kod parçası

Yukarıdaki kod satırlarında olduğu gibi yöntemin gövdesi yöntemin kendisinden daha açık olduğunda bu yöntemin kullanılması kodu daha açık hale getirmiş olacaktır.

3.1.1.3. Introduce explaining variable

Karmaşık ifadelerin ve koşullu ifadelerin düşük düzeyde basitleştirilmesi yeniden düzenleme tiplerinin özellikleri arasındadır (Counsell vd., 2015). Bunlardan biri olan Introduce Explaining Variable, karmaşık bir ifadeyi basitleştiren, düşük seviyeli bir kod tabanlı yeniden düzenleme tekniğidir (Counsell vd., 2015). Teknik kodun içerisinde karmaşık bir ifade tespit ederse ifadenin sonucunu veya ifadenin bölümlerini amacı açıklayan bir adla birlikte geçici bir değişkene koyar (Fowler vd., 1999). Böylece koda yeni değişkenlerin tanıtılmasıyla karmaşık ifade basitleştirilir. Şekil 3.3(a) orijinal kod

parçasını temsil ederken, Şekil 3.3(b) Introduce Explaining Variable uygulanmış kod parçasını göstermektedir.

```
double price() {

        return _quantity * _itemPrice -
                Math.max(0, _quantity - 500) * _
                itemPrice * 0.05 +
                Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

(a) Orijinal

```
final double basePrice = _quantity * _itemPrice;
final double quantityDiscount = Math.max(0, _quantity - 500) * _itemPrice * 0.05;
final double shipping = Math.min(basePrice * 0.1, 100.0);

double price() {
                return basePrice - quantityDiscount + shipping;
                }
```

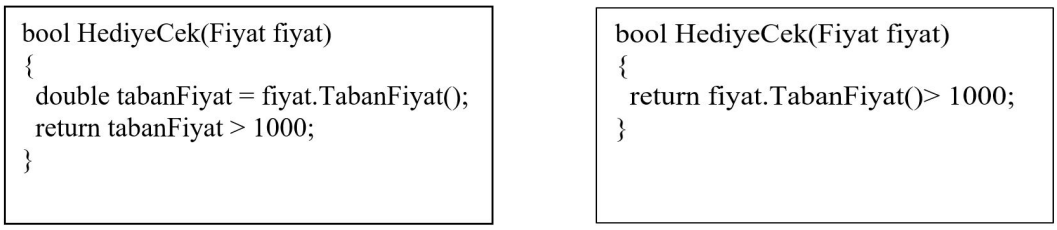
(b) Introduce Explaining Variable

Şekil 3.3. Orijinal kod parçası ve introduce explaining variable uygulanmış kod parçası (Fowler vd., 1999)

Geçici değişkenler, yalnızca bir yöntem bağlamında kullanışlıdır. Bir yöntem, nesnenin tamamında ve diğer nesnelerde kullanılabilir (Fowler vd., 1999). Fakat yerel değişkenlerin Extract Method gibi tekniklerin kullanımını zorlaştırdığı zamanlar yerine bu tekniği kullanmak avantaj sağlayabilir. Teknik bir koşulun her bir cümlesini almanın ve koşulun ne anlama geldiğini iyi adlandırılmış bir geçici ile açıklamanın yararlı olduğu koşullu mantık ile değerlidir (Fowler vd., 1999). Diğer taraftan bu tekniğin uygulandığı yerlerde sınıfların daha büyük ve daha yüksek oranda bağlı olma eğiliminde olduğu bilinmektedir (Counsell vd., 2015).

3.1.1.4. Inline temp

Kod bloğu içerisinde basit bir ifadenin sonucunu atayan ve daha fazlası olmayan geçici bir değişken var ise bu değişkeni ifadenin kendisi ile değiştirmek gerekebilir (Fowler vd., 1999). Böyle durumlarda Inline Temp yeniden düzenleme tekniğini kullanmak gerekir. Şekil 3.4(a) orijinal kod parçasını temsil ederken, Şekil 3.4(b) Inline Temp uygulanmış kod parçasını göstermektedir.



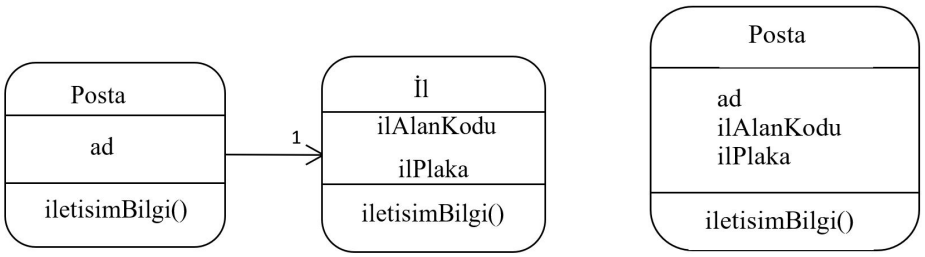
(a) Orijinal (b) Inline Temp

Şekil 3.4. Orijinal kod parçası ve inline temp uygulanmış kod parçası

Kod bloğunda tabanFiyat değişkeninin referansı "fiyat.TabanFiyat()" ifadesinin referansı ile değiştirilmiştir. Geçici değişkenlerin silinmesi, gereksiz geçici değişkeni ana belleklerden ve önbellek belleklerden alma süresini kısaltmaktadır. Bununla birlikte, eğer değişken bir yöntemin sonucuna atanmışsa, gereksiz değişkenden kurtularak programın okunabilirliği marjinal olarak artırılabilir.

3.1.1.5. Inline class

Bir sınıf bazen neredeyse hiçbir işlem yapmaz veya hiçbir seyden sorumlu değildir. Böyle durumlarda sınıfın bütün özelliklerini ilişkili olduğu diğer sınıfa devretmek devredilen sınıfın boyutunu artıracak fakat sınıflar arası bağlantıyı azaltacaktır (Elish ve Alshayeb, 2011). Yeniden düzenleme tekniklerinden biri olan Inline Class, Şekil 3.5(a) orijinal kod parçasının diyagram halini temsil ederken, Şekil 3.5(b) Inline Class uygulanmış hali diyagram üzerinde verilmiştir.



(a) Orijinal (b) Inline Class

Şekil 3.5. Orijinal kod parçası ve inline class uygulanmış kod parçası

Posta sınıfında, İl sınıfında bulunan genel alanları ve yöntemleri oluşturduktan sonra, Posta sınıfının metotu İl sınıfının eşdeğer metotuna karşılık gelecek şekilde

düzenlenmelidir. Gerekli düzenlemeler yapıldıktan sonra programın çalışması test edilmeli ve herhangi bir hata bulunmadığından emin olunmalıdır. Move Method ve Move Field işlemi kullanılarak metotlar ve alanlar Posta sınıfına aktarılır. Bu işleme İl sınıfı boş kalana kadar devam edilir. Sonunda İl sınıfı silinir. Bu teknik ile gereksiz sınıf ortadan kaldırılarak bellekte yer açılmasına ve programın daha iyi performansta çalışmasına olanak sağlanır.

3.1.1.6. Self encapsulate field

Bir sınıf içindeki özel alanlara sınıf içerisinden doğrudan erişim olanağı bulunmaktadır. Bu özel alanlar için bir "getter" ve "setter" oluşturulmaktadır ve sadece alana erişmek için bu yeniden düzenleme tekniğini kullanmak gerekmektedir. Bazen bir sınıf içindeki özel alana doğrudan erişebilmek yeterince esnek değildir. Bu teknik sayesinde ilk sorgulama yapıldığında bir alan değeri başlatabilir veya atandığında alanın yeni değerlerinde belirli işlemler gerçekleştirilebilir. Şekil 3.6(a) orijinal kod parçasının diyagram halini temsil ederken, Şekil 3.6(b) Self Encapsulate Field uygulanmış halini temsil etmektedir.

```
class Mesafe
{
 private int kisa, uzun;

 bool Durum(int gelen)
 {
  return gelen >= kisa && gelen <=
         uzun;
 }
}
```

(a) Orijinal

```
class Mesafe
{
  private int kisa, uzun;

 int Kisa {
  get { return kisa; }
 }
 int Uzun{
  get { return uzun; }
 }

  bool Durum(int gelen)
 {
  return gelen >= kisa && gelen <=
         uzun;
 }
```

(b) Self Encapsulate Field

Şekil 3.6. Orijinal kod parçası ve self encapsulate field uygulanmış kod parçası

Alanlara dolaylı erişim, bir alanın erişim yöntemleri (getter ve setter) aracılığıyla gerçekleştirildiği zamandır. Bu yaklaşım, alanlara doğrudan erişimden çok daha esnektir. Ayrıca alt sınıflardaki "getter" ve "setter" yeniden tanımlanabilir. Alan değeri sadece

kurucuda belirtilir, böylece alanın tüm ömrü boyunca alan değiştirilemez. Fakat, alanlara doğrudan erişim kullanıldığında esneklik azalsa da kod daha basit ve daha bakımlı görünmektedir.

3.1.1.7. Replace magic number with symbolic constant

Sihirli sayı diye ifade edilen bir sayı, kaynak kod içerisinde karşılaşılan fakat açık bir anlam ifade etmeyen sayısal verilere denmektedir. Bir problemin çözümünde bu tür anti-desen çözümler kullanmak yazılım geliştirme, değiştirme ve bakım aşamalarında çok büyük sorunlara yol açabilmektedir. Örnek verecek olursak, bu sihirli sayıyı değiştirmemiz gerektiğinde "bul ve değiştir" mantığı işe yaramayabilir (Fowler vd., 1999). Çünkü aynı sayı farklı yerlerde farklı amaçlar için kullanılabilir. Bu yüzden bu sayıyı kullanan her kod satırını tek tek incelemek gerekir ki bu da maliyeti artıran bir unsurdur. Böyle durumlarda Replace Magic Number with Symbolic Constant yaklaşımını kullanmak gerekmektedir. Şekil 3.7(a) orijinal kod parçasını temsil ederken, Şekil 3.7(b) Replace Magic Number with Symbolic Constant uygulanmış kod parçasını ifade etmektedir.

```
double DaireAlan(double yaricap )
{
  return yaricap*3.14;
}
```

(a) Orijinal

```
const double PI_Sabit = 3.14;
double DaireAlan(double yaricap )
{
  return yaricap*PI_Sabit;
}
```

(b) Replace Magic Number with Symbolic Constant

Şekil 3.7. Orijinal kod parçası ve replace magic number with symbolic constant uygulanmış kod parçası

Bu teknik bir sabitin değerini değiştirmek, başka bir yerde farklı bir amaç için kullanılan aynı sayıyı yanlışlıkla değiştirme riski olmadan düzenlemeyi sağlar. Değiştirilecek sayıyı tüm kod tabanı boyunca aramaktan çok daha kolaydır. Ayrıca sayının bellekte birden fazla kopyası oluşmasının da önüne geçilmekte ve değer karmaşası sorunu da giderilmektedir. Böylece kodun okunabilirliği iyileştirilir ve bakım maliyeti azaltılmış olur.

3.1.1.8. Encapsulate field

Nesne yönelimli programlamanın temel taşlarından biri (Refactoring, 2021), nesne verilerini başka sınıf veya nesnelerden gizleme özelliği olarak bilinen Kapsülleme (Encapsulate)'dir. Private erişim belirleyici ile tanımlanan bir alan (field) başka sınıflardan gizlenmiş olur. Aynı zamanda bu alan başka sınıflarda kullanılamaz. Protected erişim belirleyici ile bulunduğu sınıf ve ondan türetilen diğer sınıflar içerisinden nesne verilerine erişim sağlanmaktadır. Kapsülleme sayesinde nesnelerin bilinçsiz kullanımının önüne geçilmiş olunur. Böyle durumlarda Encapsulate Field yeniden düzenleme tekniğini kullanmak soruna çözüm olmaktadır. Şekil 3.8(a) orijinal kod parçasını temsil ederken, Şekil 3.8(b) Encapsulate Field uygulanmış kod parçasını temsil etmektedir. Orijinal kod parçasında "kenar" adında "int" tipinde "public" olarak

```
class Kare
{
  public int kenar;
}
```

(a) Orijinal

```
class Kare
{
  private int kenar;
  public int Kenar
  {
    get {return kenar;}
    set {kenar = gelenDeger};
  }
}
```

(b) Encapsulate Field

Şekil 3.8. Orijinal kod parçası ve encapsulate field uygulanmış kod parçası

tanımlanan özellik Şekil 3.8(b)'de "private" erişim belirteci ile kapsüllenmiştir. Fakat bazı durumlarda private alanlara erişmek ve özellikleri kullanmak gerekebilir. Bu durumda Property kavramı devreye girmiştir. Property kavramı ile bir alanın değerini geri döndürmeye "get" ve değerini ayarlamaya "set" komutları ile imkan tanınır. Kapsülleme olmamış olsaydı tüm nesneler herkese açık olacak ve nesnelerin özellikleri kullanılabilir ve değiştirilebilir olacaktı. Bu durumda programın modülerliği ortadan kalkacak ve bakımı karmaşık olacaktır.

3.1.1.9. Consolidate duplicate conditional fragments

Tekrarlı kod, kaynak kodda birbiriyle aynı veya benzer kod parçaları olarak tanımlanan ve fonksiyonel olarak benzer işlevi yapan kod parçalarına denir ve aynı zamanda "kod klonu" olarak da adlandırılır (Hotta vd., 2012). Tekrarlı kodun varlığının yazılım geliştirme ve bakımı üzerinde olumsuz etkileri olduğu söylenir. Örnek olarak hata oluşumlarını artırırlar (Hotta vd., 2012). Ayrıca tekrarlı kod yazılımın bakım maliyetini artırır ve klonlanmış koddaki yanlış değişiklikler hatalı program davranışı sergilenmesine sebep olur (Sudhamani ve Rangarajan, 2015). Bu tür durumlarda hataların önüne geçmek ve bakım maliyeti gibi özellikleri iyileştirmek için Consolidate Duplicate Conditional Fragments yeniden düzenleme tekniği kullanmak gerekir. Koşul içeren kod blok parçalarının tümünde tekrarlı kod parçası yer alabilmektedir. Böyle durumlarda aynı kod parçasını koşul dışına çıkartarak kod tekrarının önüne geçilmiş olunacaktır. Şekil 3.9(a) orijinal kod parçasını temsil ederken, Şekil 3.9(b)'de Consolidate Duplicate Conditional Fragments isimli yeniden düzenleme tekniğinin örnek kod parçası üzerinde uygulanmış hali verilmektedir.

```
if (SiparisOnayı())
{
  toplam = fiyat * 0.95;
  Paketle();
}
else
{
  total = fiyat * 0.98;
  Paketle();
}
```

(a) Orijinal

```
if (SiparisOnayı())
{
   toplam = fiyat * 0.95;
}
else
{
   toplam = fiyat * 0.98;
}
 Paketle();
```

(b) Consolidate Duplicate Conditional Fragments

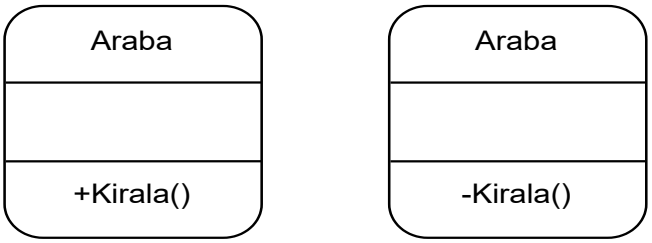
Şekil 3.9. Orijinal kod parçası ve consolidate duplicate conditional fragments uygulanmış kod parçası

Tekrarlı kod, koşullu dalların içindeki kodun evrilmesinin bir sonucu olarak koşullu olanın tüm dallarında bulunur. Bu tekniği uygulamadan önce tekrarlı kod bloğu

tespiti yapılır ve tekrarlı kod koşullu dalların başındaysa kodu koşuldan önce bir yere taşımak gerekir. Tekrarlı kod, dalların sonunda yürütülmekte ise koşullardan sonra yerleştirmek gerekir. Eğer tekrarlı kod dalların içine rastgele yerleştirilmişse, ilk önce kodu dalın başına veya sonraki kodun sonucunu değiştirip değiştirmemesine bağlı olarak sonuna taşımayı denemek gerekmektedir. Tekrarlı kodu yerleştirme işlemini tamamladıktan sonra tekrarlı kod bloğunu tekilleştirme işlemi ile tekniğin uygulanması tamamlanmaktadır (Hotta vd., 2012). Şekil 3.9(a)'da Paketle() metotu kod parçasında yer alan koşullu ifadelerin her iki dalında da yer almaktadır. Tekrarlı kod, dalların sonunda yürütülmekte ve koşullardan sonra yerleştirilmesi yapılarak teknik uygulanmaktadır. Bu metotu koşul dışına almak hem kod tekrarını önleyecek hem de koddaki kötü kokuyu giderecektir. Kodun tekrarını önlemenin yanında zamanda kodun sürdürülebilirliğini ve bakımını kolaylaştırmak için uygulanmaktadır.

3.1.1.10. Hide method

Yeniden düzenleme işlemi bazen yöntemin görünürlüğüyle ilgili durumu değiştirmemize sebebiyet verebilmektedir. Başka bir sınıfın bu yönteme ihtiyacı olduğu vakaları tespit etmek kolaydır ve yöntem daha görünür hale getirilebilir (Refactoring, 2021). Fakat yöntemin çok görünür olduğunu söylemek biraz daha zordur. Zengin bir arayüz oluştururken yöntemleri alıp gizlemek daha fazla davranış sağlayan ve oldukça yaygın bir durumdur (Fowler, 2018). Get ve set yöntemlerini gizleme yalnızca veri kapsüllemesinin ötesine çok az davranış katan bir sınıfla başlanıldığında ek davranışlar geliştirilmesine olanak sağlar. Yeni davranışlar sınıfa dahil edildiğinde, get ve set yöntemleri artık gerekli olmamaktadır ve gizlenebilir. Get ve set yöntemlerini özel yaptıktan sonra değişkenlere doğrudan erişim uygulanırsa yöntemleri kaldırmak gerekir. Hide Method uygulamak için öncelikle özel yapılabilecek metotları bulmak gerekir. Bu tür metotlar diğer sınıflar tarafından kullanılmaz veya yalnızca kendi sınıf hiyerarşisinde kullanılmaktadır (Refactoring, 2021). Bulunan her bir metodu mümkün olduğu kadar özel yapmak gerekmektedir. Şekil 3.10(a) orijinal metotun gösterimini temsil ederken, Şekil 3.10(b) Hide Method isimli yeniden düzenleme tekniğinin metotu özelleştirmesini temsil etmektedir. Şekil 3.10(a)'da yer alan kirala() metotu public bir yöntem iken 3.10(b)'de verilen hali özelleştirilmekte ve diğer sınıflar tarafından direk ulaşılamaz hale gelmektedir. Sınıfa yeni özellikler eklendikçe atıl duruma düşen public get ve



(a) Orijinal (b) Hide Method

Şekil 3.10. Orijinal kod dizaynı ve hide method uygulanmış kod dizaynı

set yöntemleri için bu metodlar uygulanabilmektedir. Hatta bu tür metotlar boşa çıkabilmekte ve bunun sonucunda metotların kullanımı sonlandırılabilmektedir. Böylece hafızadaki fazla yük (load) durumu ortadan kalkmış olur.

3.2. Yazılım Metrikleri

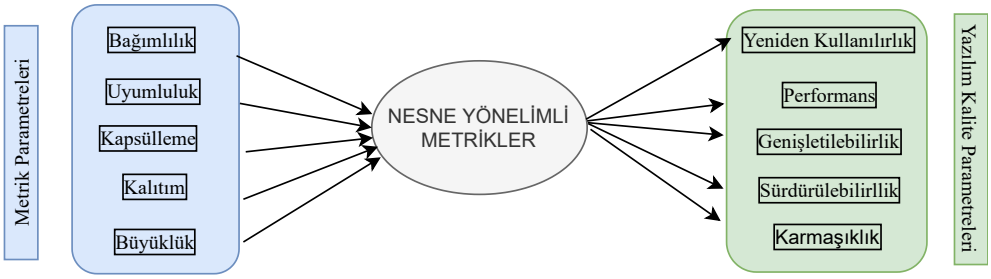
Yazılım metrikleri, yazılım geliştirme sürecini ve yazılım ürünlerinin kalitesini kontrol etmek için nicel bir yol sağlar (Li ve Henry, 1993) ve yazılımın belirli yönlerini ölçer. Çünkü yazılım metriklerinin sınıflandırılması yazılım kalitesi, yazılım geliştirme, yazılım standartları ve ölçümü konuları ile ayrılmaz bir bütün haline gelmiştir. Ayrıca, yazılım metriklerini kategoriye ayırmak zorlaşmıştır. Yazılım metrikleri genellikle iki kategoriye ayrılabilir: yazılım ürünü ölçümleri ve yazılım süreci ölçümleri (Li ve Henry, 1993). Yazılım ürünü ölçümleri, kaynak kodu veya tasarım belgeleri gibi yazılım ürünlerini ölçer. Yazılım süreci ölçümleri, tasarım ve kodlama aşamalarındaki geliştirme faaliyetlerine tahsis edilen iş gücü sayısı gibi yazılım geliştirme sürecini ve bakımını ölçer (Li ve Henry, 1993). Bu iki kategoriye ek olarak Kan tarafından geliştirilen yazılım proje ölçümleri başlığı diğer bir kategori olarak karşımıza çıkmaktadır (Kan, 2003). Bu kategori, yazılım projesinin özelliklerini ve yürütülmesini tanımlayan yazılım metrik sınıfıdır.

Günümüzde nesne yönelimli metrikler sektörde çok önemli bir rol oynamaktadır ve bir araştırma aracı olarak kullanılmaktadır (Padhy vd., 2018). Nesne yönelimli metriklerin kullanımı, kaynak kodda yazılım görselleştirmesiyle birlikte yeniden düzenlenmeye ihtiyaç duyan yerleri tespit etmek için çok uygundur (Kaur ve Singh, 2016). Geleneksel fonksiyonel ayrıştırma ve veri analizi tasarım yaklaşımı için metrikler tasarım yapısını

veya veri yapısını bağımsız olarak ölçerken; nesne yönelimli metrikler işlev ve verilerin birleştirilmesine bütünleşik bir nesne olarak odaklanmaktadır (Chidamber ve Kemerer, 1994). Bir metriğin faydasının yazılım kalitesinin nicel bir ölçüsü olarak değerlendirilmesi bir yazılım kalitesi niteliğinin ölçülmesine dayandırılmaktadır. Ancak seçilen metrikler çok çeşitli modellerde kullanışlıdır (Rosenberg ve Hyatt, 1997). Bu nedenle, nesne yönelimli metrik kriterleri aşağıdaki nitelikleri değerlendirmek için kullanılmaktadır:

- Anlaşılırlık
- Karmaşıklık
- Sürdürülebilirlik
- Testedilebilirlik
- Verimlilik
- Yeniden kullanılırlık

Nesne yönelimli metrikler, yazılım kalite nitelikleri ile ilişki halinde olmazsa yazılım kalitesine etkisinden söz etmek doğru olmaz. Bu ikili arasında etkileşimin olması yazılımın sürdürülebilirliğine katkı sağlamaktadır. Nesne yönelimli metriklerin ölçümü sayesinde metrik parametreleri ve yazılım kalite parametreleri arasındaki bağlaşım ilişkisini görmek mümkündür (Arora vd., 2011). Şekil 3.11 bu ilişkiyi temsil etmektedir.



Şekil 3.11. Yazılım metrikleri ve yazılım kalite parametreleri arasındaki ilişki (Arora vd., 2011)

Araştırmacılar, veri organizasyonunu veri tabanından incelemek için metrikleri iyi tanımlanmış bir pozitif yöntem olarak kullanırlar (Padhy vd., 2018). Çeşitli yazılım metrikleri önerilmiş ve çalışılmıştır: Halstead'in yazılım metrikleri (Halstead, 1977),

McCabe'nin çevrimsel karmaşıklık metriği (McCabe, 1976), Chidamber ve Kemerer'in CK metrikleri (Chidamber ve Kemerer, 1994), Metrics for Object Oriented Design (MOOD) metrikleri (Abreu ve Carapuça, 1994), Quality Model for Object Oriented Design (QMOOD) metrikleri (Bansiya ve Davis, 2002), Lorenz ve Kidd'in metrikleri (Lorenz ve Kidd, 1994). Genel olarak, dahili nitelikler yazılım metrikleri aracılığıyla yakalanır ve üst düzey özellikler bu metrikler için geçerli değerler cinsinden ifade edilir (Mahouachi vd., 2013). Kod metrikleri, yeniden düzenleme işleminin kod kalitesi üzerindeki etkisini değerlendirmek için dahili önlemler olarak seçilmiştir (Kumar, 2017).

3.2.1. Kod satır sayısı

Kod satır sayısı (LOC) metriği geleneksel metrik grubunda olup uzun süredir kullanılan metriklerden biridir. Yazılımın büyüklüğü hakkında ipuçları vermektedir. LOC bir prosedürü veya fonksiyonu ölçmek için kullanılır. Tüm prosedürlerin ve fonksiyonların toplam LOC'u bir programın boyutunu ölçmek için de kullanılır. Ayrıca sınıfın boyutunu ölçmek için iyi yöntemler arasındadır. Genel olarak, projenin kaynak dosyalarındaki kod boyutunu ölçmek için kod satır sayısı kullanılır (Alpernas vd., 2020).

Yaklaşık yarım asırdan beri yaygın olarak kullanılan LOC metriği kaynak kod satırı üzerinde çalışmaktadır (Jones, 1994; Fenton ve Neil, 2000), ancak farklı programlama dilleri söz konusu olduğunda görevi gerçekleştirecek kod satırı, dile göre değişiklik göstermektedir. Çünkü farklı dillerin farklı yapıları vardır. Bu yüzden farklı programlama dilleri ile yazılmış modülleri karşılaştırmak zordur. Fakat hesaplamasının kolay olması, kodun hata sayısı tespitinde ve modüllerin büyüklüğü hakkında bilgi vermesinin yanısıra kodda kullanılan yöntem, sınıf gibi özelliklerin eşik değerleri hakkında bilgi vermesiyle kodun sürdürülebilirliğine katkı sağlamaktadır.

3.2.2. Operand ve operatör sayısı

Bir bilgisayar programı, operatör veya operand olarak sınıflandırılabilecek bir dizgecik (token) koleksiyonunu içeren bir algoritmanın uygulamasıdır (Halstead, 1977). Başka bir deyişle, bir program bir dizi operatör ve onların ilgili bölümleri olarak

düşünülebilir. Halstead'in metrikleri, bu dizgeciklerin sayımlarının fonksiyonlarına odaklanmaktadır. Dizgecikleri sayarak ve hangilerinin operatör, hangilerinin operand olduğunu belirleyerek aşağıdaki temel ölçümler alınabilmektedir (Halstead, 1977):

- n1= Farklı operatörlerin sayısı
- n2= Farklı operandların sayısı
- N1= Toplam operatör sayısı
- N2= Toplam operand sayısı

Bunlara ek olarak Halstead şunları da tanımlamaktadır:

- n1*= Potansiyel operatör sayısı
- n2*= Potansiyel operand sayısı

Halstead, bir modül ve program için mümkün olan minimum operatör ve operand sayısı olarak n1* ve n2* olarak ifade etmektedir. Bu minimum sayı, gerekli işlemlerin mevcut olacağı programlama dilinin içinde yer alacaktır (Halstead, 1977). Örnek olarak C veya C# dilinde, main() fonksiyonunu herhangi bir programın içermesi gerekmektedir. Muhtemelen bir fonksiyon veya prosedür olarak n1* = 2, çünkü herhangi bir fonksiyon veya prosedür için en az 2 operatör olmak zorundadır:

- fonksiyonun adı için 1
- bir atama veya gruplama sembolü olarak işlev görmesi için 1

n2 * : Fonksiyona veya prosedüre geçirilmesi gereken, tekrarlama olmadan parametre sayısını ifade etmektedir (Menzies vd., 2002).

Halstead bütün bu metrik koleksiyonunu (n1, n2, N1, N2, n1*, n2*) "Software Science" olarak tanımlamakta ve bunları yazılım programının büyüklüğünü yansıtan uzunluk

(lenght), kelime hazinesi (vocabulary) ve hacim (volume) karakterlerini hesaplamada kullanmaktadır (Halstead, 1977).

$$N = N_1 + N_2 \tag{3.1}$$

$$n = n_1 + n_2 \tag{3.2}$$

$$V = N * log_2 n \tag{3.3}$$

Burada N programın uzunluğunu, n programın kelime hazinesini, V programın hacmini temsil etmektedir. Halstead bunlara ek olarak programın seviyesi, zorluğu gibi programın karakteristik özelliklerini de hesaplamak için metrik koleksiyonunu kullanmıştır. Halstead'ın metrikleri temelde 2 özelliğe dayanmaktadır: operatör sayısı ve operand sayısı. Fakat operatörler ve operandlar ayrımının nasıl yapılacağı konusunda tam bir görüş yoktur. Bu nedenle farklı sayma stratejileri aynı program veya algoritma için farklı sayıda operatör ve operand üretecektir. Bu iki özellik, programın kaynak kodunda veya eşdeğer algoritmada operatör ve operand sayısını sayarak kolayca matematiksel bir yapıya eşlenebilir (Al Qutaish ve Abran, 2005).

3.2.3. Fonksiyon sayısı

Fonksiyon sayısı (NOM) metriği sınıfta tanımlı fonksiyon sayısının toplamını ölçer (Demeyer vd., 2000; Lorenz ve Kidd, 1994). Kodun fonksiyon sayısı sayesinde kodun geliştirilmesine ve bakımına ne kadar zaman harcanacağı hakkında bilgi edinilebilir. Örnek olarak fonksiyon sayısı çok olan taban sınıflar çocuk düğümlerde daha çok iz bırakmaktadırlar (Ural vd., 2008). Çünkü tanımlanan tüm fonksiyonlar türetilen sınıflarda da mevcut olacaktır. Sınıf sayısı çok olan kodların tekrar kullanılabilirliği düşük olacaktır (Ural vd., 2008; Demeyer vd., 2000). Sınıftaki kodun fonksiyon sayısı sınıfın karmaşıklığının göstergesi haline gelmektedir. Fonksiyon sayısı metriği kodun hem sürdürülebilirliği hem de karmaşıklığı için önemli bir değere sahiptir.

3.2.4. Fonksiyon çağrılma sayısı

Fonksiyonları değer ile çağırma prensibinde, çağıran program ya da fonksiyondaki değişkenler ile çağrılan fonksiyonda bu değerlere karşılık gelen parametreler için bellekte farklı yerler ayrılmaktadır. Referans ile çağırma tekniğinde ise çağıran programdan argüman olarak bir değer yerine bu değere karşılık gelen bellekteki adres yollanır ve böylece veri transferi değerler yerine adresler ile gerçekleştirilir. Referans ile fonksiyon parametre çağrıları değer ile yapılan çağrılardan daha karmaşıktır. Herhangi bir tek fonksiyondan çağrılan herhangi bir işleve olan uzun mesafe çağrılan fonksiyon sayısı arttıkça artacaktır (Ryder ve Thompson, 2005). Bunun nedeni çağrılan fonksiyonların hepsinin bellekte aynı konumda olmamasıdır. Fonksiyon çağrılma sayısı metriği de bir fonksiyonun çağrılarının sayısıyla ilgili ölçümleri hesaplamak için kullanılmaktadır (Gray vd., 2011). Bir sınıfın fonksiyon çağrılma sayısı fazla olması sınıfın karmaşıklığının da yüksek olduğunu göstermektedir (Misra vd., 2011). Bunun sonucunda işlemci açısından işlemin yürütme bağlamını kaydetmek ve geri yüklemek, iş parçacığı planlamak ve konumun kaydedilmesi gibi işlemler yüzünden daha fazla kaynak tüketimi gerçekleşecektir. Fonksiyon çağrılma sayısı metrik ölçümü sayesinde yazılımın karmaşıklığı, bakımı ve sürdürülebilirliği hakkında bilgiler elde edilebilmektedir.

3.2.5. Bağlaşım sayısı

Sınıfın bağımlı olduğu sınıf sayısı nesne yönelimli programlamada nesne sınıfları arasındaki bağlaşım sayısı olarak ifade edilmektedir. Bağlaşım, iki nesnenin birbirine bağımlılığının bir ölçüsüdür (Dubey ve Rana, 2011). Bağlaşım sayısı metriği ise bir sınıf içinde tanımlı metotların ya da niteliklerin (attribute) diğer sınıfta kullanılmasının yanı sıra sınıflar arasında kalıtım ilişkisi yok ise iki sınıf arasında bağımlılık olduğunun tespitinde kullanılmaktadır (Ural vd., 2008). Kaliteli yazılımdan bahsettiğimiz zaman sınıflar arasındaki aşırı bağımlılığın olmaması gerekmektedir. Aşırı bağımlılık modüler yazılım tasarımına uygun değildir ve yazılımın tekrar kullanılabilirliğini azaltır. Bir sınıf ne kadar bağımsızsa başka uygulamalarda o kadar kolaylıkla yeniden kullanılabilir. Bağımlılıktaki artış, değişime duyarlılığı da artıracağından, yazılımın bakım maliyeti artar ve bakımı daha zor hale gelir. Bağımlılık aynı zamanda tasarımın farklı parçalarının

ne kadar karmaşık test edileceği hakkında fikir verir (Ural vd., 2008). Bu metrik sınıfın yeniden kullanılabilirliği, karmaşıklığı, test maliyeti, bakımı ve sürdürülebilirliği hakkında bilgi verir.

3.2.6. Kalıtım ağacının derinliği

Nesne yönelimli sistemlerin önemli özelliklerinden biri kalıtımdır. Kalıtım, bir sınıfın başka bir sınıfın özelliklerini edinme kabiliyetidir (Chhikara vd., 2011). Kalıtımın temelindeki amaç, kodu tekrar tekrar yazmak yerine aynı kodu kullanarak tekrarlardan kaçmaktır. Bir davranış bir süper sınıfta tanımlandıktan sonra, bu davranışı otomatik olarak tüm alt sınıflar miras alır. Bu sayede yöntem yazma gibi bir davranış tanımlanır ve tüm alt sınıflar tarafından kullanılır. Aynı şekilde bir alan tanımı süper sınıfta tanımlandıktan sonra aynı alan tanımı tüm alt sınıflar tarafından miras alınır. Sonuçta bir sınıf ve çocukları ortak özellikleri paylaşmış olurlar (Chhikara vd., 2011). Bu mekanizma sınıf hiyerarşisi tasarımını destekler (Chidamber ve Kemerer, 1991).

Kalıtım ağacı derinliği (DIT) metriği, doksanlı yılların başlarında Chidamber ve Kemerer tarafından önerilen ve nesne yönelimli metrikler listesinde (Chidamber ve Kemerer, 1994) yer alan ölçümlerden biridir. DIT, sınıfın içinde bulunduğu kalıtım hiyerarşisindeki pozisyon sayısını ölçmektedir (Chidamber ve Kemerer, 1991). Miras hiyerarşisi içindeki bir sınıfın derinliği üst sınıfların sayısıyla ölçülen sınıf düğümünden ağacın köküne kadar olan maksimum uzunluktur. Herhangi bir sınıftan türememiş sınıflar için metrik değeri 0 olarak ölçülür. Kalıtım ağacında bir sınıf ne kadar düşükse, kalıtım nedeniyle bu sınıf üst sınıf özelliklerine erişebilir (Li ve Henry, 1993). Alt sınıf üst sınıfta tanımlanan yöntemleri kullanmadan üst sınıftan devralınan özelliklere erişirse üst sınıfın kapsüllenmesi ihlal edilmiş olur (Li ve Henry, 1993).

DIT metrik ölçümü ne kadar büyük ise sınıfı korumanın o kadar zor olduğunu söylenebilir. Ayrıca bir sınıf hiyerarşide ne kadar derinse devralma olasılığı fazla olan yöntemlerin sayısı bir o kadar fazla olur (Chidamber ve Kemerer, 1994) ve bu da davranışını tahmin etmeyi daha karmaşık hale getirir (Makkar vd., 2012). Fakat miras kullanımının, gerekli yazılım bakım maliyetini ve test yükünü azalttığı bilinmektedir ((Chidamber ve Kemerer, 1994; Fenton, 1994). Aynı zamanda miras yoluyla

kodun tekrar kullanılması ile kod tekrarı oluşmaması sonucunda daha sürdürülebilir, anlaşılabilir ve güvenilir bir yazılım üretildiği bilinmektedir (Chidamber ve Kemerer, 1994).

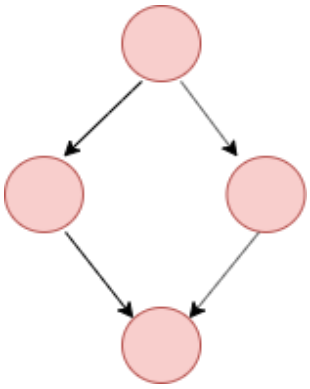
3.2.7. Çevrimsel karmaşıklık

Çevrimsel karmaşıklık (CC), program karmaşıklığını kontrol etmek ve yönetmek için kullanılabilecek bir graf tabanlı karmaşıklık ölçüsüdür (McCabe, 1976). CC, graf teorisi üzerine kurulmuştur. Bu metrik, bir programdaki doğrusal bağımsız yolların sayısına dayanır ve temel yol testi metodolojisindeki test durumlarının sayısını belirlemek için kullanılabilir (Watson vd., 1996). CC, bir programdaki işlevlere, modüllere, yöntemlere veya sınıflara göre hesaplanabilir. Temel olarak kontrol akış grafını kullanarak ilgili programın karmaşıklığını ölçmeye çalışır. Graf üzerinde yer alan düğümler, program kodunda bulunan if, switch, while, for, goto gibi mantıksal ifadelere göre oluşturulmaktadır ve bu düğümler ayrıtlar ile birbirine bağlanır. Örneğin bir metodun içerisinde yer alan if-else gibi karar yapılarının sayısı olarak ifade edilebilir. Çoklu modüllerden oluşan yazılımda problemlerin gözükme sıklığı daha fazladır. Bundan dolayı çoklu modüllerde genel olarak karmaşıklık hesabı Denklem 3.4'e (McCabe, 1976) göre hesaplanmaktadır:

$$V(G) = E - N + 2 * P \tag{3.4}$$

Denklem 3.4'de E grafın ayrıt sayısını (edge), N grafın düğüm sayısını (node), P ayrık bağlı bileşen (modül) sayısını (başlangıç 1 olarak kabul edilir), V çevrimsel sayıyı, G grafın fonksiyonu olan karmaşıklığı temsil etmektedir. Bu denklemde yazılım kodunun ne derece hata eğilimli olduğunu belirleyen karmaşıklık değeri hesaplanmaktadır. Hataya meyilli olan kodlar düşük güvenirliğe ve yüksek riske sahip olma eğilimindedirler.

CC, graf tabanlı bir kontrol akışı (control flow graph) kullanmaktadır. Bunu graf üzerinde bulunan ayrıtlar ve düğümlerin sayısı üzerinden gerçekleştirmektedir. Bu yüzden bir modülün karmaşıklık hesabını yapmak için öncelikle Şekil 3.12'deki gibi akış grafı çizilir. Program modülünün akış grafiğini oluşturmak için, modül *if*, *while*,



Şekil 3.12. If-then-else durumunun akış grafı

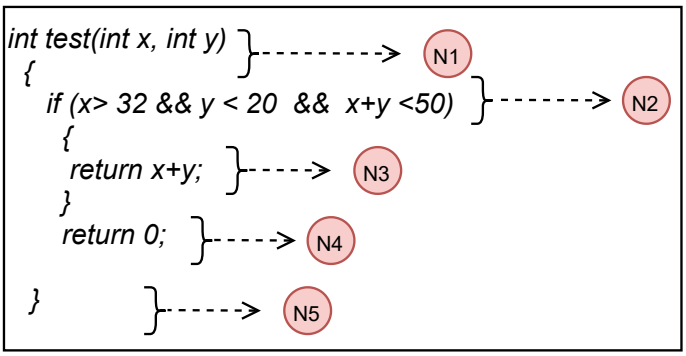
for, *switch*, *goto* gibi akışı etkileyen ifadelerle sınırlandırılmış alt bloklara bölünür ve bu bloklar grafın düğümlerini meydana getirir (Ikerionwu, 2010). Düğümler ayrıtlar ile birbirlerine bağlanır. Program kodundaki her bir dallanma ayrıtların yönünü göstermektedir. Çizilen graf üzerindeki düğüm ve ayrıt sayıları Denklem 3.4 kullanılarak kodun karmaşıklık hesabı yapılabilmektedir. Şekil 3.13'te fonksiyona parametre olarak gelen iki sayının verilen aralıkta olup olmadığını kontrol eden kod bloğu verilmektedir.

```
int test( int x, int y)
{
    if (x>32 && y<20 && x+y<50)
    {
        return x+y;
    }
    return 0;
}
```

Şekil 3.13. Kod bloğu

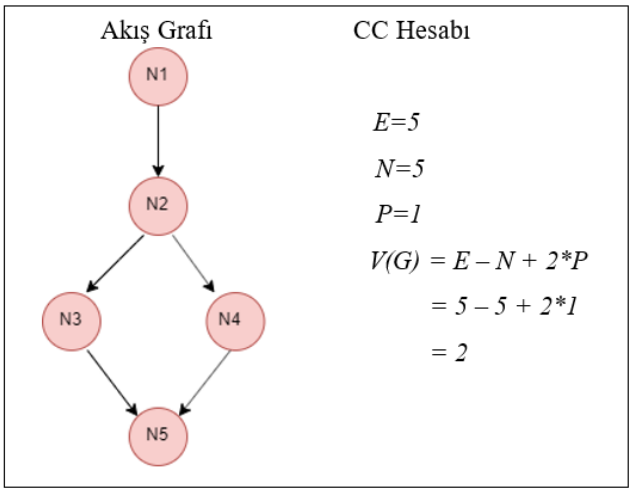
Verilen Java kod bloğunun Eclipse Metrics Plugin yazılım ölçüm aracı ile elde edilen değer 2 olarak karşımıza çıkarken GMetric aracıyla 4 olarak karşımıza çıkmaktadır. Yazılım ölçüm uygulamaları genellikle aynı kod için farklı karmaşıklık ölçüm değerleri çıkarmaktadır. Bunun nedeni, bir kontrol akış grafında koşuldaki birleşimleri ayrı dallar olarak sayarken, bir başka yazılım ölçüm aracında ise bu koşulun birleşimlerini tek bir dal olarak saymasıdır. Eclipse Metrics Plugin yazılım ölçüm aracı için Java kod

bloğunun düğümlerinin gösterimi Şekil 3.14'te verilmektedir. Akış grafı ve çevrimsel karmaşıklık hesabı ise Şekil 3.15'te gösterilmektedir.



Şekil 3.14. Kod bloğunun düğüm gösterimi

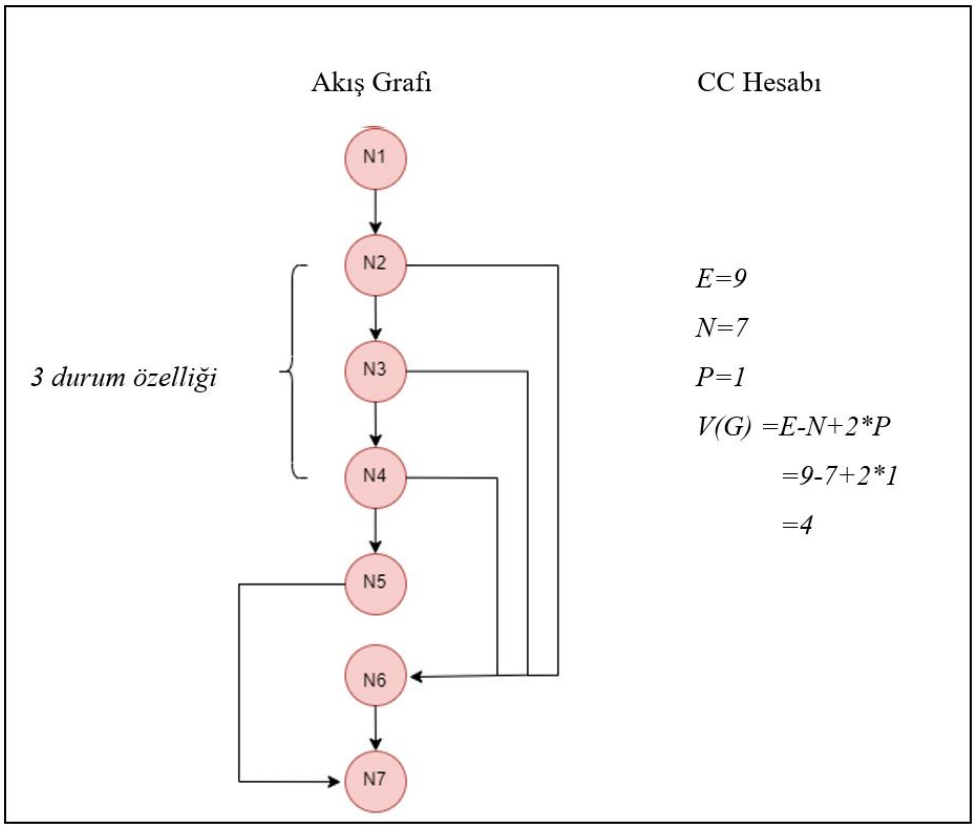
Kod bloğunda toplamda 5 adet ayrıt ve 5 adet düğüm bulunmaktadır. Tek modül üzerindeki kod bloğunun çevrimsel karmaşık değeri 2 çıkmaktadır.



Şekil 3.15. Eclipse metrics plugin çevrimsel karmaşık hesablama yöntemi

Daha gelişmiş bir ölçüm McCabe tarafından sunulmuş olan McCabe çevrimsel karmaşıklık (McCabe, 1976) ölçüsüdür. McCabe bir kontrol akış grafında koşuldaki birleşimleri ayrı dallar olarak saymayı tavsiye etmektedir. McCabe, karmaşıklığın maksimum modül boyutunun niceliksel bir gösterimini sağlamak için kullanılabileceğini ileri sürmektedir. Çok sayıda gerçek programlama projesinden veri toplayarak, çevrimsel karmaşıklığın = 10 değerinin bir üst sınır olarak göründüğünü keşfetmiştir. Modülün çevrimsel karmaşıklık boyutu bu sayıyı aştığında, bir modülü yeterince test

etmek son derece zorlaşmış olur (Ikerionwu, 2010).



Şekil 3.16. McCabe çevrimsel karmaşık hesaplama yöntemi

Verilen Java kod bloğu için McCabe'nin akış grafı ve çevrimsel karmaşıklık hesabı Şekil 3.16'da gösterilmektedir. McCabe'nin çevrimsel karmaşıklık hesabına göre 9 adet ayrıt ve 7 adet düğüm bulunmaktadır. Tek modül üzerindeki kod bloğunun çevrimsel karmaşıklık değeri 4 çıkmaktadır. Elde edilen sonuçlardan da anlaşılacağı gibi farklı yazılım ölçüm uygulamalarının aynı kod üzerindeki çevrimsel karmaşıklık değeri kullandıkları yöntemlerden dolayı farklı çıktığı görülmektedir.

3.2.8. Mantıksal kod satır sayısı

Kaynak deyimi çalışma zamanında bir eylem gerçekleştiren veya derleyicileri derleme zamanında yönlendiren bir kod bloğu olarak kabul edilir (Nguyen vd., 2007). Deyimler üç tipte sınıflandırılır: yürütülebilir, bildirim ve derleyici yönergesi. Yürütülebilir deyimler sonunda çalışma zamanı eylemlerine neden olmak için makine koduna çevrilirken bildirim ve derleyici yönergesi ifadeleri derleyicinin eylemlerini etkiler

(Nguyen vd., 2007). Örneğin CodeCount™ (CodeCount™, 2020) her derleyici yönergesini mantıksal bir kod satır sayısı (SLOC-L) olarak sayarken LocMetrics (LocMetrics, 2019), her derleyici yönergesini mantıksal bir SLOC olarak saymaz. Carnegie Mellon üniversitesinde yer alan Yazılım Mühendisliği Enstitüsü (Software Engineering Institute-SEI) çerçevesinde bildirimler, derleyici direktifleri, yorumlar ve boş satırlar yürütülebilir tipler haricindeki (nonexecutable) kaynak deyim tipi olarak değerlendirilmektedir (Park, 1992).

```
static void Main()
{
    // Bildirim deyimi.
    int counter;

    // Atama deyimi.
    counter = 1;

    int[] radii = { 15, 32, 108, 74, 9 }; // Bir dizi bildirme ve
                                             başlatma.
    const double pi = 3.14159; // Bir sabit bildirme ve başlatma.

    // Birden çok deyim içeren foreach deyim  bloğu.
    foreach (int radius in radii)
    {
        // Başlatıcı bildirim deyimi
        double circumference = pi * (2 * radius);
        // İfade deyimi (fonksiyon çağırma).
        System.Console.WriteLine("Radius of circle #{0} is {  1}.
                                  Circumference = { 2:N2}",
                                 counter, radius, circumference);

        // İfade deyimi (postfix increment).
        counter++;
    } // foreach deyim bloğunun sonu
} // Main fonksiyon gövdesinin sonu.
```

Şekil 3.17. Deyim içeren kod bloğu (Bill, 2015)

Fiziksel kod satırları net bir başlangıç ve bitiş noktasına sahip olması sayesinde kolayca sayılabilmektedir. Fiziksel kod satır sayısı (SLOC-L), program kaynak kodundaki boşluksuz ve yorumsuz satırların toplamını ifade etmektedir. Diğer taraftan, tüm kaynak kod satırlarını saymak yazılımın odak noktası olan mantıksal ifadeleri temsil etmemektedir. Bu yüzden, mantıksal SLOC saymak alternatif bir çözüm olarak karşımıza çıkmaktadır ve COCOMO II maliyet tahmin modeli için standart olarak mantıksal SLOC önerilir (Nguyen vd., 2007). Mantıksal SLOC sayma, sayılan deyimlerin fiziksel biçiminden bağımsız olarak gerçekleştirilir. Diğer bir ifadeyle,

bir satırda birden çok mantıksal ifadenin bulunabileceği veya bir mantıksal ifadenin birden çok satıra yayılabileceği anlamına gelmektedir. C ve benzeri programlama dilinde ifadeyi sonlandıran noktalı virgül karakterlerinin sayısıdır.

3.2.9. Fonksiyon başına düşen ifade sayısı

Kaynak kod düzeyinde "atomik" ve nispeten bağımsız bir birim gibi ele alınan deyim, bir programcının belirli bir zamanda gerçekleştirdiği en küçük iş artışı olarak kabul edilir. Deyimler program komutlarıdır. Herhangi bir programın gerçekleştirdiği eylemler deyimlerle temsil edilmektedir. C# kodu anahtar kelimeler, ifadeler ve operatörlerden oluşan deyimlerden oluşur (Bill, 2015). Sınıftaki kodun fonksiyon sayısı sınıfın karmaşıklığının göstergesi iken fonksiyon başına düşen ifade sayısı (Grechanik vd., 2010) fonksiyonun ne kadar büyük olduğunun göstergesidir. Elde edilen veriler yazılımın boyutu ve bakımı hakkında yorumlar yapılmasına yardımcı olabilmektedir.

3.2.10. Bakım yapılabilirlik indeksi

Bakım yapılabilirlik indeksi (MI), kaynak kodun ne kadar sürdürülebilir olduğunu (desteklenmesini ve değiştirilmesini) ölçen bir yazılım ölçüsüdür (Najm, 2014). Bakım yapılabilirlik indeksi formülü kod satır sayısı, çevrimsel karmaşıklık ve Halstead hacmi (HV) birleşiminden oluşmaktadır (Najm, 2014). Formüle geçmeden önce kaynak koddan aşağıdaki metrikleri ölçmemiz gerekir:

- HV = Halstead hacmi
- CC = Çevrimsel karmaşıklık
- LOC = Kod satır sayısı

Elde edilen ölçümlerden sonra MI değeri Denklem 3.5'te verilen formül aracılıyla hesaplanmaktadır (Coleman vd., 1994):

$$MI = 171 - 5.2 * ln(HV) - 0.23 * (CC) - 16.2 * ln(LOC) \tag{3.5}$$

Microsoft Visual Studio tarafından kullanılan türev (versiyon 2008'den beri) Denklem 3.6 ile hesaplanır (Kexugit, 2021):

$$MI = MAX(0, (171 - 5.2 * ln(HV) - 0.23 * (CC) - 16.2 * ln(LOC) * 100/171) \quad (3.6)$$

Negatif sayıların azalan kullanışlılığı ve metriği olabildiğince güvenilir tutma arzusunun bir sonucu olarak, 0 veya daha az indeksin tamamını 0 olarak ele almaya ve ardından 171 veya daha az aralığı 0 ile 100 arasında temel almaya karar verilmiştir (Kexugit, 2021). Bunun üzerine, eşik değerler belirlenmiş ve bu 0-100 aralığı 80-20 ayrılmış, böylece gürültü seviyesi düşük tutulmuş ve sadece şüpheli olan işaretli kod alınmıştır:

- 0-9 = Kırmızı
- 10-19 = Sarı
- 20-100 = Yeşil

Eğer indeks kırmızıyı gösterirse, kodla ilgili bir sorun olduğunu belirtmekte ve kodun sürdürülebilirlik açısından beklenen seviyede olmadığına işaret etmektedir.

Coleman'a göre, 85'in üzerindeki bir MI değeri yazılımın yüksek düzeyde sürdürülebilir olduğunu gösterir. 85 ile 65 arasındaki bir değer orta düzeyde bir sürdürülebilirliği gösterir ve 65'in altındaki bir değer sistemin bakımının zor olduğunu ve sürdürülebilirlik düzeyinin düşük olduğunu gösterir (Coleman vd., 1994). Kaydırılmış ölçek (0 ila 100) türevi kullanan Microsoft Visual Studio 2010 geliştirme ortamı da dahil olmak üzere çeşitli otomatik yazılım metrik araçlarında bu metrik kullanılır (Najm, 2014).

3.3. Metin Madenciliği

Veri madenciliği (Berry ve Linoff, 2004), eldeki verilerden değerli olanı ya da önceden bilinmeyen ve veriler içerisinde gizli olarak kalan aynı zamanda potansiyel olarak kullanışlı olan bilginin çıkarılması yaklaşımıdır (İlhan vd., 2008). Veri madenciliği ile yüzeysel benzerliği olan (Witten, 2004) metin madenciliği ise özel amaçlar için metinden bazı bilgiler çıkarmak adına, metnin analiz edilmesi işlemidir (Visa, 2001).

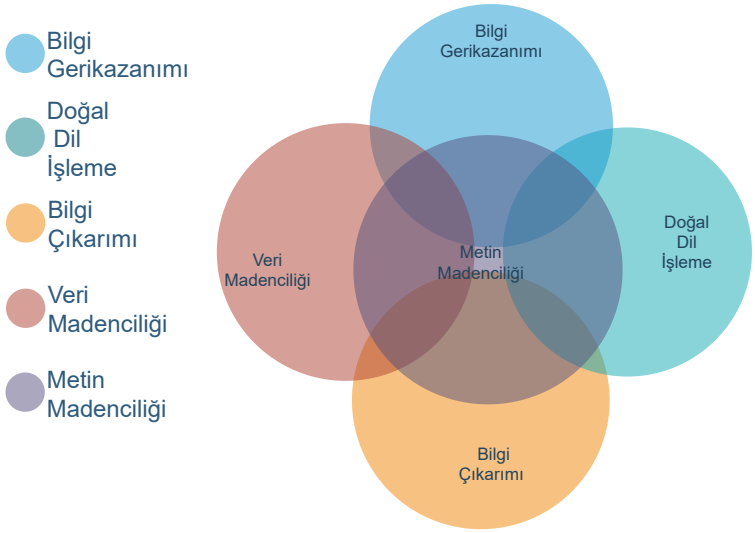
Örnek olarak metinlerden konu çıkarılması (concept/entity extraction), metinlerin sınıflandırılması, bölütlenmesi (clustering), duygusal analiz (sentimental analysis), metin özetleme (document summarization), sınıf taneciklerinin üretilmesi (production of granular taxonomy), varlık ilişki modellemesi (entity relationship modelling) gibi çalışmalar verilebilir (Seker, 2015b). Metin madenciliği, "metnin hesaplamalı analizi" için bir yöntemdir (King, 2015). Metin madenciliği araştırmacılar tarafından kelime frekanslarını saymak, kelime kullanım modellerini bulmak, n-gramları belirlemek (eş anlamlı kelimelerin), büyük ölçekli kalıpları ve eğilimleri incelemek ve konu başlıklarını keşfetmek için kullanır (Vijayarani vd., 2015). Metin madenciliği yoluyla metin dizileri (corpora) analizi metinler hakkında yeni bir fikir verebilir ve metin incelemesinden elde edilen veriler ek metin analizi için kullanılabilir.

3.3.1. Metin madenciliği alanları

Metin kalıplarını ve madencilik süreçlerini analiz etmek için uygulanan farklı metin madenciliği alanları ve teknikleri vardır: bilgi gerikazanımı (anahtar kelime arama / sorgulama ve indeksleme), doğal dil işleme (yazım düzeltme, temel hale döndürme (lemmatizasyon), gramer ayrıştırma ve kelime anlamı belirsizleştirme), bilgi çıkarma (ilişki çıkarma / link analizi) ve veri madenciliği (sınıflandırma, kümeleme, regrasyon) (Gupta vd., 2009; Dang ve Ahmad, 2014; Vijayarani vd., 2015). Şekil 3.18'de metin madenciliği genel alanları gösterilmektedir.

3.3.1.1. Bilgi gerikazanımı

Bilgi gerikazanımı, kullanıcı tarafından elde edilmek istenen genellikle bilgisayarlarda büyük dokümanlarda depolanan belirli bilgileri çıkarma yapılandırılmamış nitelikte metin materyalini bulma işlemidir (Manning vd., 2010). Belirli bir kelime veya cümle grubuna göre ilgili ve ilişkili kalıpları çıkarmayı sağlar. Diğer bir ifadeyle ilgilenilen metin koleksiyonu ile ilgili ön bilginin toplandığı aşamadır. Doküman alımının uzantısı olarak kabul edilir. Böylece doküman alımını, kullanıcı tarafından ortaya konan sorguya odaklanan bir metin özetleme aşaması veya tekniklerin kullanıldığı bir bilgi çıkarımı aşaması izleyebilir. Bilgi gerikazanım sistemleri, belirli bir sorunla ilgili



Şekil 3.18. Metin madenciliği alanları

doküman kümesini daraltmaya yardımcı olur (Kumar ve Bhatia, 2013). En bilinen bilgi gerikazanım sistemleri, Google ve benzeri arama motorlarıdır. İlgili belgeleri Web'deki bir cümleye göre çıkarmak için bilgi gerikazanım sistemini daha sık kullanırlar. Bu arama motorları, eğilimleri takip etmek ve daha önemli sonuçlar elde etmek için sorgu tabanlı algoritmalar kullanır (Talib vd., 2016). Bilgi gerikazanım sistemleri, analiz için belge sayısını azaltarak analizi önemli ölçüde hızlandırabilir (Sumathy ve Chidambaram, 2013).

3.3.1.2. Doğal dil işleme

Doğal dil işleme, insan dilinin incelendiği bir çalışma alanı olup yapay zeka alanındaki en eski ve en zorlu problemlerden biridir. Bu teknik sayesinde bilgisayarlar, insanların yaptığı gibi doğal dilleri anlayabilir. Metin madenciliğinin bütün aşamalarında kullanılmamakla birlikte metinden bazı anlamsal bilgilerin elde edilmesinde ve özellik çıkarımı sırasında sıklıkla başvurulan yöntemdir. Ayrıca sisteme bilgi çıkarma aşamasında girdi oluşturmaya yardımcı olmaktadır (Kumar ve Bhatia, 2013).

3.3.1.3. Bilgi çıkarımı

Bilgi çıkarımı, yapılandırılmış veya yarı yapılandırılmış makine tarafından okunabilen büyük miktarda metinden anlamlı bilgiler çıkaran bir tekniktir (Talib vd., 2016). Bilgi çıkarım sistemleri dokümandaki belirli nitelikleri ve varlıkları çıkarmak ve ilişkilerini

kurmak için kullanılır (Dang ve Ahmad, 2015). Çıkarılan külliyat (corpus) daha ileri işlemler için veritabanında saklanır. Külliyat kelimesi ile kastedilen, çok sayıdaki metnin düzenli ve yapısal olarak bir arada bulunması durumudur (Seker, 2008). Daha ilgili sonuçlara ulaşmak için bilgi çıkarma işlemini uygularken ilgili alanla alakalı ayrıntılı ve eksiksiz bilgi gerekir (Steinberger, 2012).

3.3.1.4. Veri madenciliği

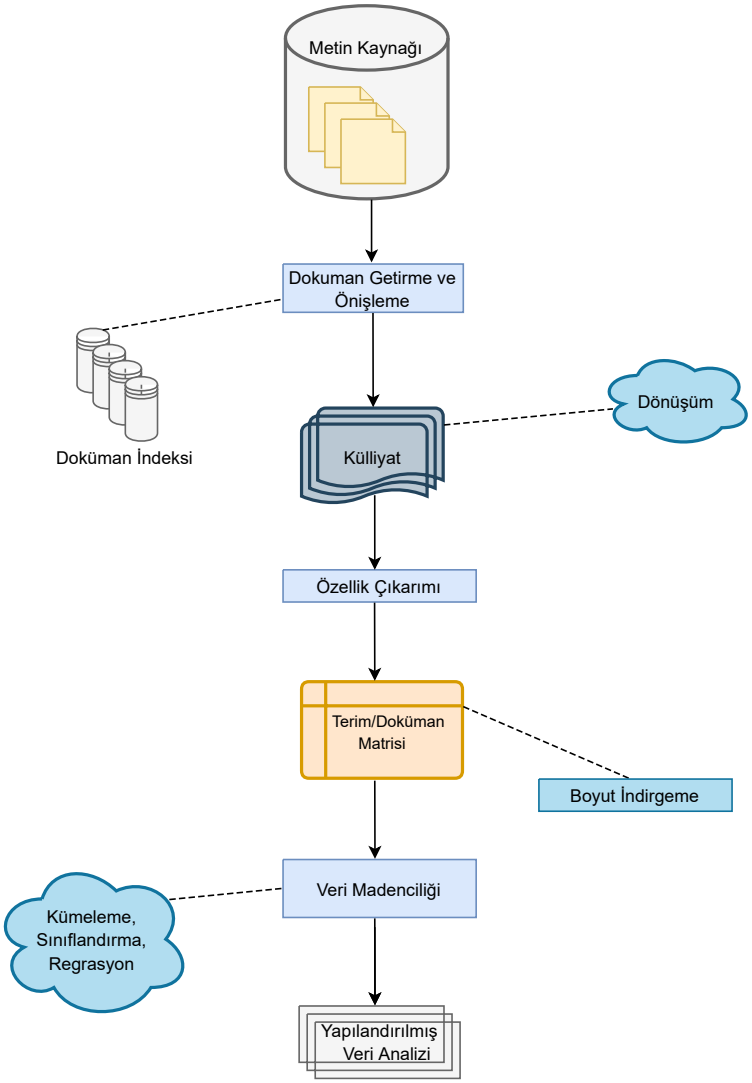
Veri madenciliği, verilerden otomatik olarak istatistiksel kuralları ve kalıpları keşfetmeye çalışır. Metin madenciliğindeki çeşitli araçları kullanarak metinleri anlamlı veriler haline getirme, metinlerden elde edilen verileri madenleme tekniğidir. Amaç, anlamı ortaya çıkaran ve araştırmacıların başka türlü keşfetmesi zor olabilecek yeni bilgileri keşfetmelerini sağlayan bilgi parçaları arasındaki ilişkileri bulmaktır ve bu bilgi parçalarını daha ileri analizler için anlaşılabilir bir yapıya dönüştürmektir (Sumathy ve Chidambaram, 2013). Öyleyse, metin madenciliği araştırmayı hızlandıran, yeni sorular sormamızı veya eskileri test etmemizi sağlayan bir araçtır.

3.3.2. Metin madenciliği adımları

Metin madenciliği metnin "sayısallaştırılması" süreci olarak özetlenebilir. Bu bilgiler ışığında metin madenciliğinin genel çalışma sürecinin mantıksal şeması Şekil 3.19'da verilmiştir.

Metin madenciliği, bilgiyi verimli bir şekilde madenlemek için gerçekleştirilecek bir dizi adımları içermektedir. Şekil 3.19'da detaylı olarak verildiği gibi metin madenciliği genellikle girdi metninin yapılandırılması (bazı türetilmiş dilsel özelliklerin eklenmesi, metin önişleme aşaması, ardından bir külliyata eklenmesiyle birlikte ayrıştırılması), yapılandırılmış veri içinde birtakım işlemlerin yapılması (özellik çıkarımı, boyut indirgeme, sınıflandırma, kümeleme vs.), analiz ve yorumlama adımlarını içermektedir.

İşlenecek verilerin belirli bir formatta olmaması metin madenciliği açısından verilerin analiz edilmesini zorlaştırmaktadır. Bu yüzden metin madenciliği alanında ön işleme aşaması, veri temizleme ve veriyi uygun formata getirme işlemi gerçekleştirilmektedir



Şekil 3.19. Metin madenciliği adımları

(Feldman vd., 2007). Bu aşamada kullanılan bazı teknikler Metin Temizleme (Cleanup), Metin Parçalama (Tokenization) ve Cümlenin Öğelerini Etiketleme (Part-of-speech Tagging) tekniklerdir.

Metin Temizleme; gereksiz veya istenmeyen bilgileri kaldırma anlamına gelir. Örnek olarak reklamları web sayfalarından kaldırma, ikili formattan dönüştürülen metni normalleştirme, tablolarla, şekillerle ve formüllerle uğraşmak gibi gereksiz veya istenmeyen bilgilerin kaldırılması anlamına gelir (Kumar ve Bhatia, 2013).

Metin Parçalama; belirli bir metinde kullanılan tüm kelimeleri elde etmek için bir parçalama (tokenization) işlemi gereklidir. Bir metin belgesi tüm noktalama işaretlerini kaldırarak sekmeleri, satır sonu karakterlerini ve diğer metin olmayan karakterleri

(non-text) boşlukla değiştirerek bir sözcük akışına bölünür (Hotho vd., 2005). Temiz ve uygun formata gelen metne külliyatı oluşturan dökumanlarda dizgeciklere (token) ayırma işlemi uygulanır. Daha sonra bütün dökumanlarda yer alan kelimelerin birleştirilmesi ile ilgili koleksiyonun "sözlüğü" nü (dictionary) elde edilir (Hotho vd., 2005).

Cümlenin Öğelerini Etiketleme; her dizgecik için kelime sınıfı ataması anlamına gelmektedir. Giriş olarak, dizgecikleştirilmiş belirli bir metin verilir. Etiketleyiciler, bilinmeyen kelimelerle ve belirsiz kelime etiketi eşlemeleriyle baş etmek zorundadır.

3.3.3. Metin madenciliği ve alanları kapsamında kullanılan kavramlar ve teknikler

Gelişen teknoloji ve kullanılan altyapılar sayesinde iletişim ve bilgi teknolojilerinin kullanımı da artmaktadır. Bu tür teknolojilerin devamlılığını sürdürebilmesi için sağlam altyapılar kullanılması kaçınılmazdır. Internet alyapısının gelişen teknolojiye ayak uydurması gerekmektedir. Bu altyapıların kullanımı da günden güne artmaktadır. Artan yapısal olmayan metin içerikli verilerden yapısal ve anlamlı veriler elde etmek için metin madenciliği kullanılmaktadır. Çünkü elde edilecek veriler üzerinde veri madenciliği teknikleri uygulayabilmek için öncelikle yapısal olmayan metinlerin işlenmesi gerekmektedir. Bu bağlamda tez kapsamındaki bu çalışmada kullanılan kavramlar ve yöntemler sırasıyla verilmektedir:

- Metin parçalama (Tokenization)
- Noktalama işareti silme (Punctuation)
- Durdurma kelimeleri (Stop Words)
- Kökenine döndürme (Stemming)
- Temel hale döndürme (Lemmatization)
- Cümle bölümlendirme (Sentence Segmentation)
- Düzenli ifadeler (Regular Expression)

- Dizin yöntemleri
    - ✓ Düzeltme (Trim)
    - ✓ Dizin bulma (Indexof)
    - ✓ Altdizin bulma (Substring)
    - ✓ Son dizin bulma (LastIndexOf)

Metin madenciliğinde genellikle girdi metninin yapılandırılması sırasında kullanılan metin içerikli dokümanların hazırlanması aşaması olan metin önişleme en kritik aşamalardandır. Bu aşamada metin içerikli dizgelerin dizgeciklere (token) parçalanması, gereksiz sık kullanılan kelimelerin (stop words) ayıklanması ve kelime köklerinin bulunması (stemming) en sık kullanılan ön işleme tekniklerindendir.

3.3.3.1. Metin parçalama

Nesne yönelimli programlama dilleri ile oluşturulan uygulamalarda genellikle uygulama içerisinde metinsel değerlerin tutulduğu veri tiplerinden biri karakter dizgeleridir. Dizgeler, programlama diline ait ve o dilde tanımlı olan sembollerin bir içerik akışında farklı sıra ve sayılarda sıralanması sonucu elde edilen metinlerdir. Metin parçalama ise metinsel bir içerik akışını kelimelere, terimlere, simgelere veya sembol adı verilen diğer bazı anlamlı öğelere ayırma işlemidir. Bu işlem sayesinde dizge içerisinden bir parça çıkarılarak verinin bir parçası elde edilir. Parçalara ayrılmış bu dizgelere ise alt dizge (substring) denilmektedir.

3.3.3.2. Noktalama işareti silme

Metinler gibi dizgeler içerisinde kullanılan noktalama işaretlerinin (! . , ; - ?) silinmesi string ifadeler üzerinde işlem yapacak modeller ile doğru sonuçlar elde edilmesine olanak sağlamaktadır. Bu tür nokalama işaretlerinin metin içerisinden kaldırılmasına olanak sağlayan teknik, noktalama işareti silme olarak adlandırılmaktadır.

3.3.3.3. Durdurma kelimeleri

Filtreleme yöntemleri ile dokümanlardaki sözcükler diğer bir ifadeyle sözlükteki kelimeler filtrelenebilir. Bu yöntemlerden en yaygın olarak kullanılan filtreleme yöntemi "durdurma kelimeleri filtreleme yöntemi"dir. Durdurma kelimesi filtreleme yöntemi, bağlaçlar, edatlar gibi içerik bilgisi olmayan veya içeriğe katkısı olmayan kelimeleri kaldırmaktır (Hotho vd., 2005). Frekansa dayalı girdi dokümanlarında sıralanacak kelimeler arasından "durdurma kelimeleri" filtreleme yöntemi ile sıralamadan çıkartılacak terimler tanımlanmaktadır. Bu yöntem sayesinde sözlükte (bütün dokümanlarda) ayırt edici etkisi olmayan ve istatistiksel olarak etkin bir faydası olmayan kelimeler çıkartılmaktadır (Hotho vd., 2005).

3.3.3.4. Kökenine döndürme

Kelimeleri basit, yalın haline getiren ya da çeviren metin madenciliği tekniğine "kökenine döndürme" denilmektedir. Türkçeden örnek vermek gerekirse çoğul eklerin kelimeden atılması veya çekim eki almış fiilin çekim ekinden ayrılarak fiilin kökünün elde edilmesidir.

3.3.3.5. Temel hale döndürme

Metin içerisinde geçen aynı kelimeden türemiş kelimeleri ayırt ederek tek bir kelimeye indirgenmesi yöntemidir. Temel hale döndürme işlemi kelimelerin cümle içerisindeki görevlerini hatta konumlarını tespit etmesi açısından hataya açık ve zor bir işlem olarak değerlendirilmektedir. Ayrıca bu işlem dilden dile faklılık göstermektedir (Tunalı, 2011).

3.3.3.6. Cümle bölümlendirme

Doğal dil işleme tekniklerinin çoğu cümle sınırları ile sınırlanan kelime belirteçlerinin dizilerine uygulanır ve bu nedenle metnin de cümlelere bölünmesini gerektirir. Metnin cümlelere bölünmesi çoğu durumda basit bir meseledir. Nokta, ünlem işareti veya soru işareti genellikle bir cümle sınırına işaret eder. Bununla birlikte, bir noktanın ondalık bir noktayı gösterdiği veya kısaltmanın bir parçası olduğu ve dolayısıyla bir

cümle kesintisine işaret etmediği durumlar vardır. Cümlelerin bölümlere ayrılması farklı dillerde beklenmeyen bazı zorluklar ortaya çıkarabilir. Bu yüzden sorunu çözmek için basit düzenli ifadelerden daha fazla kapsamlı sisteme kadar çeşitli yöntemler kullanılmaktadır (Mikheev, 2003). Diğer taraftan metnin cümlelere bölünmesi birçok metin işleme uygulaması geliştirmek için önemlidir. Bu işlemler için cümle bölümlendirme tekniği kullanılmaktadır. Teknik kullanılırken cümlenin sonunu işaret edebilecek noktalama işaretlerinin tespiti cümlenin doğru analizi için önemli yer tutmaktadır.

3.3.3.7. Düzenli İfadeler

Metin madenciliğinde metin içerisinde özel bilgiler yer alabilmektedir. Bu tür bilgilere örnek vermek gerekirse kişisel numaralar, e-posta adresleri, kimlik numaraları gibi bilgiler verilebilir. Bu tür durumlarda genellikle düzenli ifadeler veya içerik bağımsız gramerler tanımlanarak metin üzerinde çalıştırılır (Seker, 2015b). Düzenli ifadeler, karakter dizilerinin (strings) kümelerini tanımlamak için kullanılan bir gösterimdir. Belirli bir dizi düzenli ifade tarafından tanımlanan kümedeyken, düzenli ifadenin dizi ile eşleştiği bilinmektedir. Örnek olarak sayılarla eşleşen düzenli ifade gösterimine Denklem 3.7'de yer verilmektedir (Mikheev, 2003):

$$[0-9][0-9]?[0-9]?(?[0-9][0-9][0-9])*([.][0-9]+) \tag{3.7}$$

En basit düzenli ifade tek bir değişmez karakterdir. Özel meta karakterleri hariç *+?()|, karakterleri kendileri ile eşleşir. Düzenli ifadeyi bir meta karakteriyle eşleştirmek için ters eğik çizgiyle kaçış : \+ değişmez bir artı karakteriyle eşleşir.

Yeni bir düzenli ifade oluşturmak için iki normal ifade değiştirilebilir veya birleştirilebilir: m1 s ve m2 t ile eşleşirse, m1 | m2 s veya t ile eşleşir ve m1m2 st ile eşleşir.

Meta karakterleri *, + ve ? tekrarlayan işleçler: e1 * eşleşir. Her biri m1 ile eşleşen sıfır veya daha fazla dizilerin dizisi; m1 + bir veya daha fazla eşleşir; m1? sıfır veya bir ile eşleşir (Cox, 2007).

3.3.3.8. Dizin yöntemleri

Düzeltme (Trim) : Karakter dizisi (string) içerisinde geçerli tüm öndeki ve sondaki boşluk karakterlerini kaldırmaya yarayan yöntemdir. Bir boşluk olmayan karakterle karşılaştığında her baştaki ve sondaki kesme işlemini durdurur. Örneğin geçerli bir dize "abc xyz" ise Trim yöntemi "abc xyz" döndürür. Beyaz boşluk (whitespace) karakterlerini dizesindeki kelimeler arasından kaldırmak için "Düzenli İfadeler" yöntemi kullanılır (Microsoft, 2022).

Dizin Bulma (IndexOf) : Belirtilen unicode karakterin veya dizenin karakter dizisinde ilk geçtiği sıfır tabanlı dizini bildiren yöntemdir. Belirtilen karakter veya dize bulunmadığında yöntem -1 döndürür. Bu yöntem aşırı yükleme şeklinde kullanıma sahiptir (Microsoft, 2022).

Altdizin Bulma (Substring) : Bir karakter dizisinden belirtilen karakter konumunda başlayan ve dizinin sonunda biten bir alt dizin bulma yöntemidir. Başlangıç karakteri konumu bir sıfır tabanlıdır. Aşırı yükleme ile kullanımları mevcuttur. Örnek olarak Substring(Int32, Int32) yöntemi, belirtilen karakter konumunda başlar ve belirtilen uzunluğa kadar bir alt dizin alır. Bu yöntem geçerli örneğin değerini değiştirmez, bunun yerine başlangıcı yeni bir dize olan karakter dizisi döndürür (Microsoft, 2022).

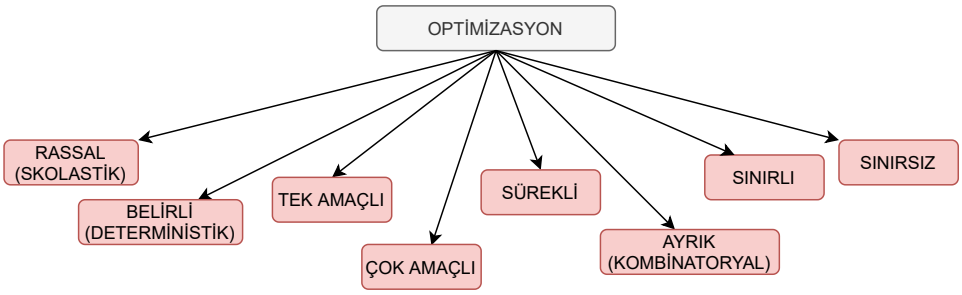
Son Dizin Bulma (LastIndexOf) : Belirtilen Unicode karakter veya geçerli karakter dizini içinde belirtilen bir dizenin son oluşum dizin konumunu tespit eden yöntemdir. Aşırı yükleme şeklinde kullanımları mevcuttur. Örnek olarak public int LastIndexOf (string value, int startIndex, StringComparison comparisonType) yönteminde arama, belirtilen karakter konumunda başlar ve dizenin başına doğru geriye gider. Belirtilen dize için arama yaparken gerçekleştirilecek karşılaştırma türünü bir parametre belirtir (Microsoft, 2022).

3.4. Kombinatoryal Optimizasyon

Optimizasyon havayolu planlamadan finasmana, internet yönlendirmeden mühendislik tasarımanına kadar her yerdedir ve bu nedenle geniş bir uygulama yelpazesi ile önemli

bir kavram haline gelmektedir (Koziel ve Yang, 2011). Hemen hemen tüm mühendislik ve endüstri uygulamalarında maliyet ve enerji tüketimini en aza indirgemek veya performansı, verimliliği ve karı en üst düzeye çıkarmak için sürekli bir şeyler optimize edilmeye çalışılmaktadır. Gerçek hayatta zaman, para ve kaynaklar daima sınırlı olduğu için optimizasyonun pratikte önemi daha fazladır. Herhangi bir türde mevcut kaynakların en uygun şekilde kullanılması bilimsel düşüncede bir kavram değişikliğini gerektirir. Bunun sebebi, gerçek dünyadaki uygulamaların çoğunun sistemin davranış şeklini etkileyecek çok daha karmaşık faktörlere ve parametrelere sahip olmasıdır (Koziel ve Yang, 2011).

Gelişen teknoloji ile az maliyet ve enerji tüketimi ile performans ve verimlilik artırılabilmekte fakat aynı zamanda yeni problemler ortaya çıkabilmektedir. Yeni problemlerin ortaya çıkması karar mekanizmasının işini zorlaştırmaktadır. Çünkü, bilinen çözüm teknikleri, problemlerin yeni halleri için yeterince çözüme ulaştıramamaktadır. Çözülmesi gereken problemler ya da karar verilmesi gereken yeni haller ne olursa olsun amaç daima en iyi sonucu elde etmek olmalıdır. Bu bilgiler ışığında optimizasyon, elde olan kaynakları en verimli şekilde kullanarak verilen amaç veya amaçlar için en iyi çözümü elde etmeyi gerçekleştirme sürecidir.



Şekil 3.20. Optimizasyon problem tipleri

Optimizasyon problemleri genel olarak ifade edildiklerinde kolay ancak çözülmesi zor olarak tanımlanan problemlerdir. Şekil 3.20, optimizasyon problem tiplerini göstermektedir. Tipleri probleme bağlı olarak değişen optimizasyon problemlerinin ortak yönü, objektif fonksiyon adı verilen minimize veya maksimize edilecek olan bir fonksiyon ile kısıtlamalar kümesi olarak adlandırılan sonlu sayıda eşitsizlik veya eşitlik sisteminden meydana gelmektedir. Optimizasyon problemleri birçok yolla formüle

edilebilmektedir. Genel olarak yaygın formülasyon, problemi doğrusal olmayan bir optimizasyon yöntemi olarak yazmaktır (Yang, 2013):

$$min \quad f_i(u), (i = 1,2,3,\ldots Z), \tag{3.8}$$

$$k.s. \quad h_j(u) = 0, (j = 1,2,3,\ldots K), \tag{3.9}$$

$$g_k(u) \leq 0, (k = 1,2,3,\ldots L) \tag{3.10}$$

Yukaridaki optimizasyon probleminde f_i, h_j ve g_k doğrusal olmayan fonksiyonlardır. Karar değişken vektörü $u = (u_1, u_2, u_3, \ldots . u_n)$ n boyutlu uzayda sürekli değişkenlerden veya tamsayılı değişkenlerden oluşabilmektedir. Amaç fonksiyonu ya da maliyet fonksiyonunu f_i temsil etmekte ve $Z > 1$ olduğu durumlarda, optimizasyon çok amaçlı ya da çok kriterli olmaktadır (Yang, 2013).

Karar değişken vektörü $u = (u_1, u_2, u_3, \ldots . u_n)$ ve $g_k(u) \leq 0$, $k = (1,2,3,\ldots .L)$ kısıt sistemini ele aldığımızda, $g_k(u) = 0$ denklemini sağlayan u'ların kümesi, tasarım uzayında bir çok boyutlu yüzey oluşturmaktadır. Bu kısıt yüzey, tasarım uzayını, $g_k(u) < 0$ ve $g_k(u) > 0$ olmak üzere iki bölgeye ayırır. $g_k(u) < 0$ olan bölgedeki noktalar "kabul edilebilir", $g_k(u) > 0$ olan bölgedekiler ise uygun olmayan şeklinde tespit yapılabilmektedir. Ayrıca $f_i(u)$ fonksiyonunun minimum noktası u^* olarak kabul edersek, aynı nokta fonksiyonunun negatifi $-f_i(u)$ ile ifade edilebilmekte ve problemin maksimum noktasıdır. f_i, h_j ve g_k'nın bazı doğrusal olduğu durumlarda, problem doğrusal olarak isimlendirilir. Bazı karar değişkenleri sadece ayrık değerler (integer gibi) aldığında, diğer değişkenler sürekli olsa dahi, büyük ölçekli optimizasyon problemleri için genelde zor olan karışık tipte olmasıdır (Yang, 2013).

Kombinatoryal optimizasyon, kombinatorik doğrusal (linear) programlama tekniklerini ve ayrık yapılar üzerindeki optimizasyon problemlerini çözmek için algoritmalar teorisini birleştiren bir uygulamalı matematik dalıdır (Papadimitriou ve Steiglitz, 1998). Kombinatoryal terimi, karar değişkenlerinin kesikli olduğunu ifade etmektedir. Diğer bir deyişle, problem çözümünün tamsayıların ya da diğer kesikli nesnelerin bir kümesi veya bir sırası olduğu anlamına gelir. Bu tür problemler için optimum çözümlerin bulunması kombinatoryal optimizasyon olarak sınıflandırılmaktadır. Kombinatoryal

optimizasyon problemi U, f, c üçlüsü şeklinde tarif edersek, U arama uzayını, f maksimize edilmesi veya minimuma indirilmesi gereken amaç fonksiyonunu ve c uygulanabilir çözümler elde etmek için yerine getirilmesi gereken bir dizi kısıtlamaları ifade etmektedir (Neumann ve Witt, 2010). Amaç, maksimizasyon veya minimizasyon problemi durumunda U^* ile bütün kısıtlamaları karşılayan en yüksek amaç değere sahip bir çözüm bulmaktır (Neumann ve Witt, 2010).

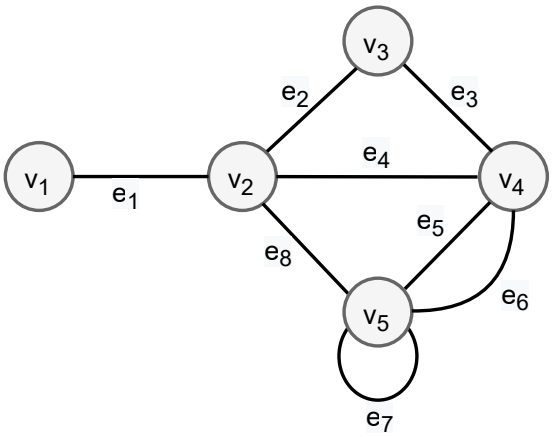
Gezgin satıcı problemi ve grafik renklendirme problemi kombinatoryal problemlerin örnekleri arasındadır. Bu problemler, belirli kısıtlamaları yerine getirerek bir kombinasyon, bir permutasyon veya bir alt küme gibi bir kombinatoryal nesne bulmayı hedefleyen problem türleridir. Hedeflenen bu kombinasyonel nesnenin aynı zamanda bir minimum veya maksimum maliyet gibi ek özelliklere sahip olması gerekebilmektedir. Genel olarak kombinatoryal problemler hem teorik hem de pratik olarak hesaplamada en zorlayıcı problemler arasındadır. Zorluğunu açıklayan birden fazla gerçeklik mevcuttur. Bunlardan biri, kombinatoryal nesnelerin sayısının normal olarak problemin boyutu ile hızlı bir şekilde büyümesi ve orta derecedeki durumlarda dahi tahmin edilemez boyutlara ulaşabilmesidir. Diğeri ise bu tür problemlerin büyük çoğunluğunu makul bir sürede çözmek için bilinen bir algoritmanın çoğu bilgisayar bilimcisi tarafından mevcut olmadığına inanılmasıdır. Fakat bazı kombinatoryal problemler etkili algoritmalar ile çözülebilir. Bunlardan biri en kısa yol bulma problemleridir (Levitin, 2011).

Kombinatoryal optimizasyon problemlerinden en tipik ve bilinen problemlerinden biri minimum yayılım ağaç problemidir (Graham ve Hell, 1985). Kombinatoryal optimizasyonun temel taşı olarak da görülmektedir. Problem hem pratik hem de teorik uygulamalarda önemlidir (Nešetřil vd., 2001). Minimum yayılım ağaç probleminin önemi ve popülaritesi çeşitli gerçeklerden kaynaklanmaktadır. Çözüm yöntemleri basit olsa da modern kombinatoriklerin fikirlerini üretmekte ve bilgisayar algoritmalarının tasarımında merkezi bir rol oynamaktadır (Graham ve Hell, 1985). Ayrıca büyük graflar için çözülmesini pratik hale getiren etkin bir çözüm sunmaktadır (Graham ve Hell, 1985).

3.4.1. Graf teorisi

Gerçek dünyadaki nesnelerin durumu, bir dizi noktadan oluşan ve bu noktaların belirli çiftlerini birleştiren çizgilerle birlikte bir diyagram aracılığıyla uygun bir şekilde tanımlanabilir. Bu tür durumların matematiksel bir soyutlaması, bir grafik kavramını ortaya çıkarır (Bondy vd., 1976). Graf teorisi, çizgileri inceleyen ve nesneler arasındaki ilişkileri modelleyen bir kuramdır. Temel anlamda bir problem kenarlar (edges) ve düğümler (vertices) ile modellenmektedir ve modellenen problemin bir graf şeklinde gösterim ilkesine dayanmaktadır (Seker, 2015a). Matematikten kimyaya bilgisayar bilimlerinden sosyal bilimlere kadar birçok farklı alanda kullanılmaktadır. Teorinin evrensel anlamda kullanılan sembolleri netlik kazanmamış olsa da bu alanda çalışan araştırmacıların (Bondy vd., 1976; Biggs vd., 1986; Gross ve Yellen, 2005; Chen, 2012) kullandıkları sembollerin çalışmacılar tarafından kabul gördüğünü söyleyebiliriz.

Tanım 3.1. G grafı, $G = (V,E)$ ile temsil edilir ve boş olmayan iki sonlu küme V ve E'den oluşan matematiksel yapıdır (Biggs vd., 1986). Grafta, V'nin elemanlarına düğümler (vertices) denir ve $V = (v_1, v_2, \ldots . v_n)$ düğümler kümesi ile gösterilir. E'nin elemanlarına kenarlar (edges) denir ve $E = (e_1, e_2, \ldots . e_m)$ kenarlar kümesi ile gösterilir (Gross ve Yellen, 2005).



Şekil 3.21. Graf çizgi çizim örneği

$G = (V,E)$ ile temsil edilen Şekil 3.21' deki graf 5 adet düğüm ve 8 adet kenardan oluşmaktadır. Grafın düğümleri ve kenarları: $V = (v_1, v_2, \ldots . v_5)$ ve $E = (e_1, e_2, \ldots . e_8)$. Örnek graftaki v_1 ve v_2 düğümleri arasındaki bağlantıyı sağlayan kenar e_1 veya (v_1, v_2) ile temsil edilir.

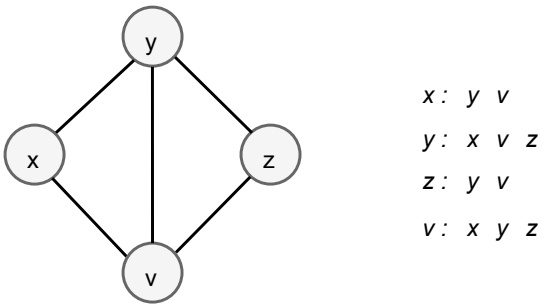
Tanım 3.2. Her kenar kendisine bağlı bir veya birden fazla düğüme sahiptir. Bunlara "başlangıç ve bitiş düğümleri" denir (Gross ve Yellen, 2005). Örnek graftaki v_1 ve v_2 düğümleri e_1 kenarının başlangıç veya bitiş düğümlerini temsil etmektedir.

Tanım 3.3. Bir v_x düğümü bir v_y düğümüne e_z kenar ile bağlı ise bu iki düğüm bitişik ve komşudur (West vd., 2001). Örnek graftaki v_2 ve v_3 düğümler hem komşu hem de bitişiktir.

Tanım 3.4. Bir kenarın başlangıç ve bitiş düğümü aynı düğüm olduğu durumlara "döngü" adı verilir (Bondy vd., 1976). Örnek graftaki e_7 kenarı döngüdür.

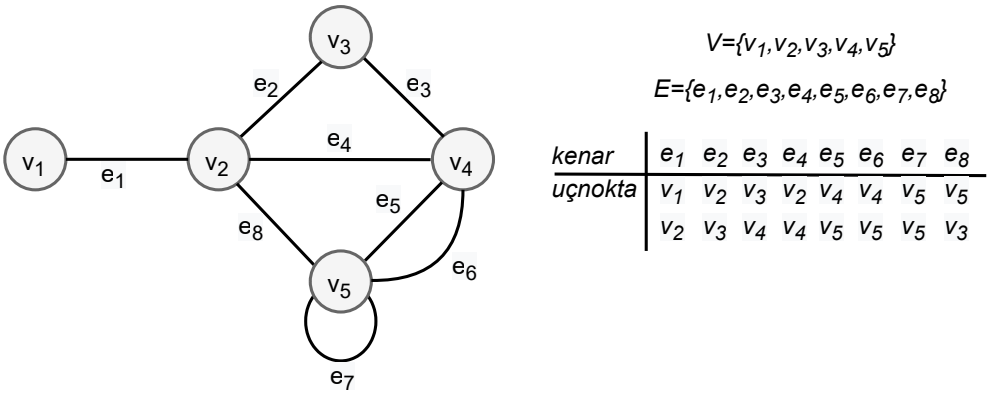
Tanım 3.5. Başlangıç ve bitiş düğümü aynı düğüm olan diğer bir ifadeyle aynı uç noktalara sahip iki veya daha fazla kenara "çoklu kenar" denir (Gross ve Yellen, 2005). Örnek grafta $\{e_6, e_7\}$ çoklu bir kenar setini oluşturmaktadır.

Tanım 3.6. Çoklu kenar veya döngü içermeyen graflara "basit graf" denir (Gross ve Yellen, 2003). Basit grafın biçimsel gösterimi her düğüm için komşu düğümlerin listesini içeren komşuluk listesi ile temsil edilir (Gross ve Yellen, 2005). Şekil 3.22 örnek basit graf ve biçimsel gösterimini vermektedir.



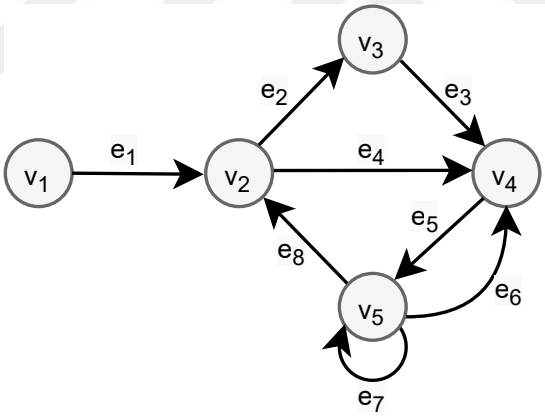
Şekil 3.22. Basit graf ve belirtim gösterimi

Tanım 3.7. Çoklu kenar veya döngü içeren graflara "genel graf" denir (Gross ve Yellen, 2003). Genel grafın biçimsel gösterimi kenar listesi, düğüm listesi ve uçnoktaları içeren bitişiklik tablosundan oluşmaktadır (Gross ve Yellen, 2005). Şekil 3.23 örnek genel graf ve biçimsel gösterimini vermektedir.



Şekil 3.23. Genel graf ve biçimsel gösterimi

Tanım 3.8. $G = (V,E)$ grafı $V = (v_1, v_2, \ldots . v_n)$ düğümler kümesi ile farklı düğümlerin sıralı ikili listelerinden e_z = { v_x v_y} oluşan $E = (e_1, e_2, \ldots . e_m)$ kenarlar kümesine yanısıra baş düğüm yönü : $e_z \rightarrow v_x$, kuyruk düğüm yönü: $e_z \rightarrow v_y$ gibi özelliklerine sahip ise bu tür graflara "yönlendirilmiş graf" denir (Gross ve Yellen, 2005). Şekil 3.24 örnek genel graf ve biçimsel gösterimini vermektedir.

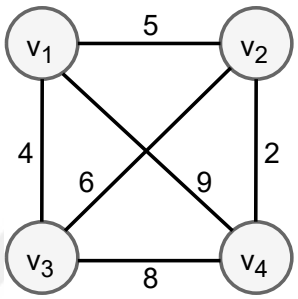


Şekil 3.24. Yönlendirilmiş graf

Tanım 3.9. Yönlendirilmiş ve yönlendirilmemiş kenarların birlikte kullanıldığı graflara "kısmi yönlendirilmiş graf" denir (Gross ve Yellen, 2003). Kısmi yönlendirilmiş grafın veya yönlendirilmiş grafın işaretlerinin kaldırılması ile temel graf elde edilir (Gross ve Yellen, 2003). Elde edilen graf aynı zamanda yönlendirilmemiş bir graftır.

Tanım 3.10. Çoklu kenar içeren fakat döngü içermeyen graflara "çoklu graf" (multigraf), kenarlar kümesi ile düğümler kümesinde hiç eleman yok ise bu tür graflara da boş graf denir (Gross ve Yellen, 2005).

Tanım 3.11. Bir grafta kenarların değerleri eşit değil ve her biri farklı bir değere sahipse bu tür graflara "ağırlıklı yada maliyetli graf" denir. Ağırlıklı bir graf $G = (V, w)$ ile temsil edilirse, burada V düğüm kümesine ve w her düğüm çiftine $(v_x v_y)$ gerçek bir negatif olmayan değer $w(v_x, v_y)$ veren bir ağırlık fonksiyonuna karşılık gelmektedir ($v_x \in V, v_y \in V$ ve $v_x \neq v_y$) (Umeyama, 1988). Şekil 3.25'te yönlendirilmemiş bir ağırlıklı graf verilmiştir.



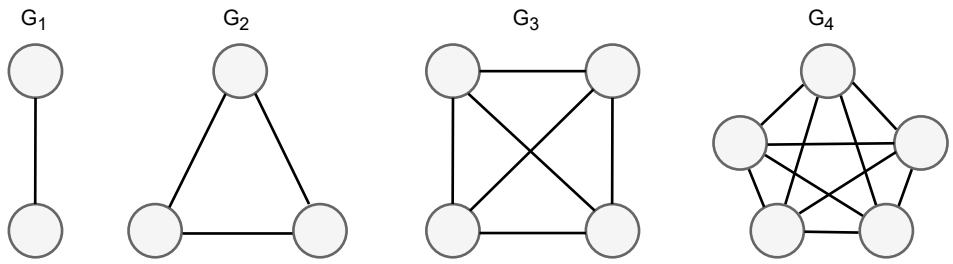
Şekil 3.25. Yönlendirilmemiş ağırlıklı graf

Tanım 3.12. $G = (V, E)$ bir graf $V = (v_0, v_1, \ldots v_n)$ düğümler kümesinden ve $E = (e_1, e_2, \ldots e_m)$ kenarlar kümesinden oluşmaktadır. Grafta v_{k-1} ile v_k gibi herhangi iki düğüm arasında bulunan ve $k = 1, 2, ..n$ için e_k'i temsil eden bir kenarın v_1 düğümünden v_n düğümüne varıncaya kadar izlediği $W = (v_0, e_1, v_1 .... v_{n-1}, e_n, v_n)$ düğüm ve kenar dizisine "yürüyüş" denir (Gross ve Yellen, 2003).

Tanım 3.13. Tekrar eden kenarı ve iç düğümü bulunmayan yürüyüş tipine "yol" (Gross ve Yellen, 2003), hiç kenarı olmayan ve sadece tek düğüm içeren yola ise "patika" denir (Gross ve Yellen, 2003).

Tanım 3.14. m düğümlü bir G grafında her bir çift farklı düğüm arasında en az bir yol varsa "bağlı graf", her düğümün bitişik düğümü ile arasında bir kenar varsa "tam graf" denir (Harary, 2015). Şekil 3.26'da tam graf örnekleri verilmektedir.

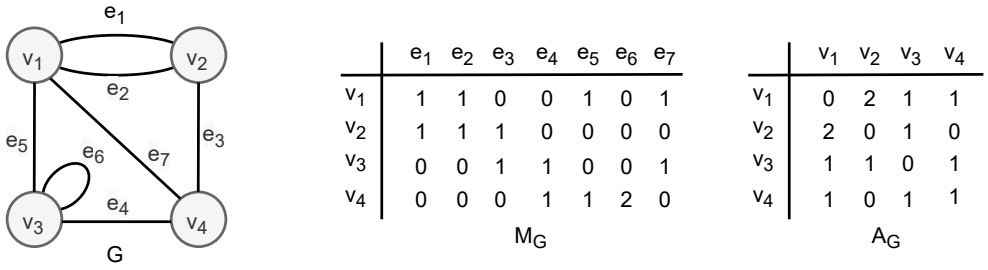
Tanım 3.15. Düğümleri $v_1, v_2, \ldots v_v$ şeklinde sıralanmış basit bir G grafının A_G ile temsil edilen ve v x v matrise karşılık gelen matrisine "komşuluk matrisi" denir ve şu özelliklere sahiptir (Gross ve Yellen, 2003):



Şekil 3.26. Tam graf örnekleri

$$A_G(a,b) = \begin{cases} 1 & \text{eğer } v_a \text{ ve } v_b \text{ komşu ise} \\ 0 & \text{değilse} \end{cases}$$

Tanım 3.16. Düğümleri $v_1, v_2, \ldots . v_v$ ve kenarları $e_1, e_2, \ldots . e_e$ olan bir G grafın v x e matrisine "bitişiklik matrisi" denir. G'nin bitişiklik matrisi $M_G = [m_{ij}]$ matrisidir, burada $[m_{ij}]$, v_i ve e_j'nin bitişiklik sayısıdır (0,1,2) (Bondy vd., 1976).



	e_1	e_2	e_3	e_4	e_5	e_6	e_7
v_1	1	1	0	0	1	0	1
v_2	1	1	1	0	0	0	0
v_3	0	0	1	1	0	0	1
v_4	0	0	0	1	1	2	0

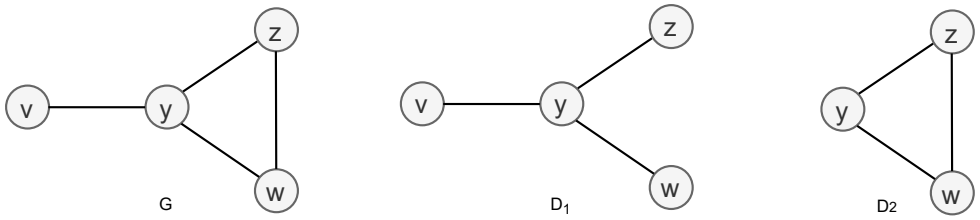
M_G

	v_1	v_2	v_3	v_4
v_1	0	2	1	1
v_2	2	0	1	0
v_3	1	1	0	1
v_4	1	0	1	1

A_G

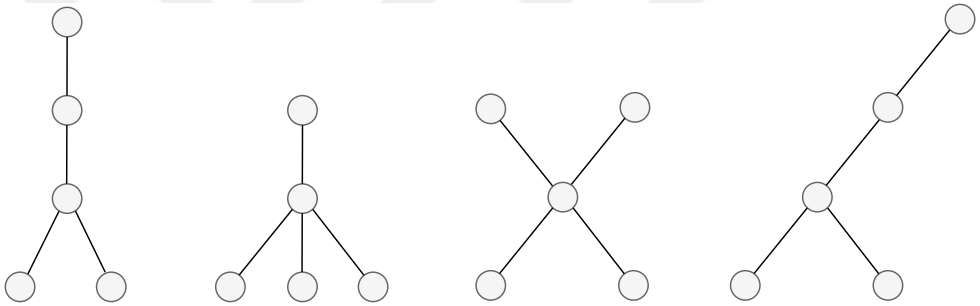
Şekil 3.27. Bir graf, grafın bitişiklik matrisi ve komşuluk matrisi (Bondy vd., 1976)

Tanım 3.17. $V_D \subset V_G$ ve $E_D \subset E_G$ olan bir D grafına "G grafının alt grafı" denir (Gross ve Yellen, 2003). Burada D grafı G grafının alt grafı ise G grafının kenar ve düğümleri D grafının kenar ve düğümlerini kapsamaktadır. Aynı zamanda G grafı D grafının süper grafı olur. Genelde G grafının bir alt grafiğine "izomorfik"; herhangi bir grafiğe de "G'nin bir alt grafiği" denilmektedir (Gross ve Yellen, 2003). Eğer $V_D = V_G$ ise D grafı G grafının yayılan (spanning) alt grafı olmaktadır. G grafında, G_P olarak gösterilen bir dizi $P = (p_1, p_2, \ldots . p_n)$ düğümler kümesine sahipse ve uç noktaları P 'de olan G'nin her kenarını içeriyorsa buna indüklenmiş alt graf denilmektedir. Şekil 3.28'de örnek bir G grafının D_1 yayılan alt grafı ve D_2 indüklenmiş alt grafı örneklerini verilmiştir.



Şekil 3.28. D_1 yayılan alt graf ve D_2 indüklenmiş alt graf
(Gross ve Yellen, 2003)

Tanım 3.18. Ağaç (tree) birbirine bağlı (connected) döngüsel olmayan (acyclic) bir graftır (Bondy vd., 1976). G grafının bir yayılan ağacı (spanning tree), bir ağaç olan G'nin yayılan alt grafıdır (Bondy vd., 1976). Yayılan ağaç bütün düğümleri kapsar ve döngü oluşturmaz. Bir ağaç olan G'nin kenar sayısı $e = v - 1$ şeklinde bulunur (Bondy vd., 1976). Şekil 3.29'da beş düğümlü bazı ağaç örnekleri gösterilmiştir.

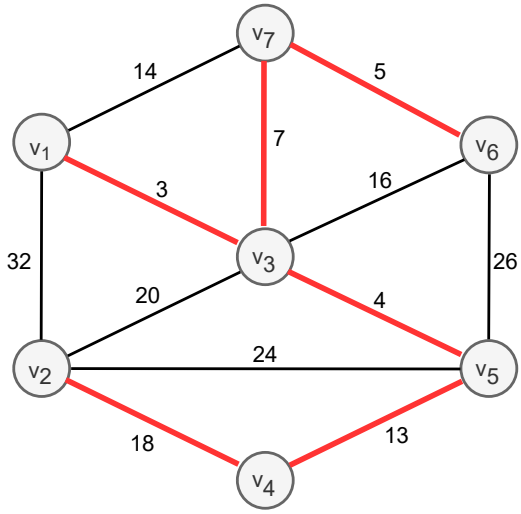


Şekil 3.29. Beş düğümlü ağaç örnekleri

3.4.2. Minimum yayılan ağaç ve algoritmaları

Minimum yayılan ağaç (MYA) (Boruvka, 1926; Kruskal, 1956; Prim, 1957), ağırlıklı bağlantılı bir grafiğin minimum ağırlıklı kapsayan ağacını bulmayı amaçlar (Graham ve Hell, 1985). Ağırlıklı bağlantılı graftaki tüm düğümleri döngüler oluşturmadan bağlayan ve tüm olası yayılma ağaçlarının minimum toplam ağırlığına sahip olan orijinal ağırlıklı grafın bir alt grafıdır (Tewarie vd., 2015). MYA aynı zamanda iyi bilinen bir optimizasyon problemidir (Chen ve Chang, 2001; Dey ve Pal, 2013; Dey vd., 2015) ve kombinatoryal optimizasyonun temel taşı olarak görülmektedir (Graham ve Hell, 1985; Nešetřil vd., 2001). Her iki araştırmacı Boruvka'ya (1926) minimum yayılan ağaç probleminin öncüsü olarak atıfta bulunmasına rağmen problemin kaynağı olarak Kruskal (1956) ve Prim'in (1957) çalışmasına ve ilk etkili çözümlerine başvurmak yazarlar arasında standart bir uygulama haline gelmiştir (Graham ve Hell,

1985). MYA uygulamasındaki belirsizlik, kenar ağırlıklarının tam olarak bulunmasını zorlaştırmaktadır. Fakat binlerce düğüme sahip büyük graflar için verimli bir çözüm olduğu kabul edilmektedir (Graham ve Hell, 1985). Şekil 3.30 verilen graf bağlantılı, yönlendirilmemiş ve ağırlıklı bir graftır.



Şekil 3.30. Ağırlıklı graf ve minimum yayılan ağacı

Ağırlıklı grafı $G = (V, E)$ temsil etmekte olup $V = (v_1, v_2, \ldots v_n)$ düğümler kümesi ve $E = (e_1, e_2, \ldots e_m)$ kenarlar kümesinden oluşmaktadır. Kenarların ağırlıkları $w(e_k)$ fonksiyonu ile gösterilirse e_k her düğüm çiftine $(v_x v_y)$ karsılık gelmektedir. Ağırlıklı graf G'nin minimun yayılım ağacı olan A'yı bulmak ve Denklem 3.11'de (Erickson, 2019) verilen fonksiyonu en aza indirmek için çeşitli algoritmalar kullanılır.

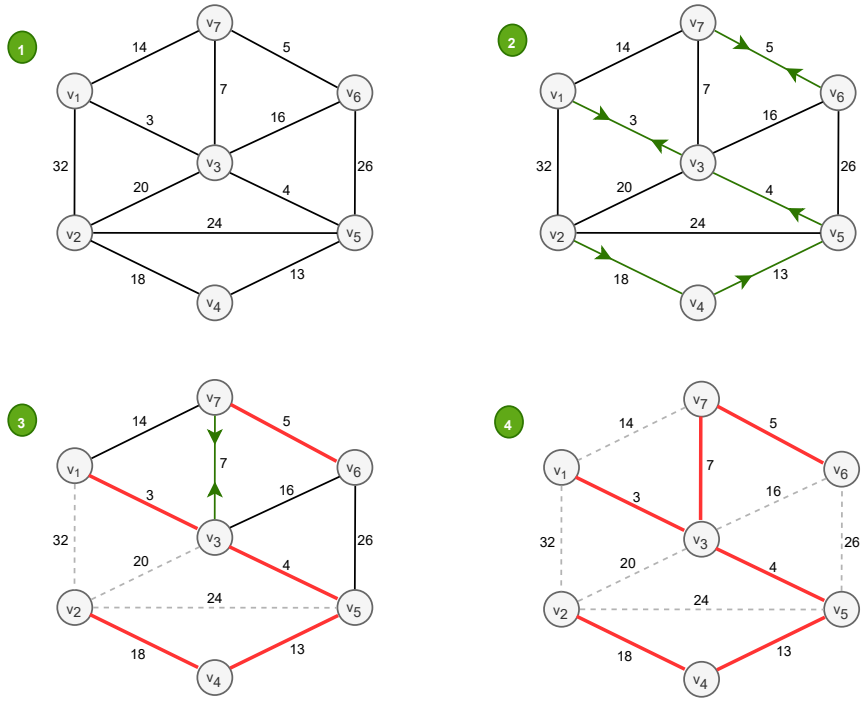
$$w(A) = \sum_{e_k \in A} w(e_k) \tag{3.11}$$

Minimum yayılma ağacı probleminin çok sayıda algoritmik çözümleri (Dijkstra, 1960; Loberman ve Weinberger, 1957; Gabow vd., 1986; Karger vd., 1995) mevcut olmasına rağmen üç algoritma, tarihi gelişimde problemin çözümü için merkezi rol oynamaktadır (Graham ve Hell, 1985): Boruvka (Boruvka, 1926); Kruskal, (Kruskal, 1956) ; Prim (Prim, 1957) .

3.4.2.1. Boruvka algoritması

Genel minimum yayılma ağacı algoritması, bir ara yayılan orman olarak adlandırılan G giriş grafiğinin döngüsel olmayan bir F alt grafiğini içerir. F, G'nin minimum kapsayan ağacının bir alt grafiğidir ve F'nin her bileşeni, köşelerinin minimum yayılan bir ağacıdır. Başlangıçta F, m adet tek düğümlü ağaçtan oluşur. Genel algoritma, ağaçları aralarına belirli kenarlar ekleyerek birleştirir. Algoritma durduğunda, F, minimum kapsayan ağaç olması gereken tek bir m adet düğümlü ağaçtan oluşur. Gelişen ormana hangi kenarların eklenmesinin belirlenmesi önem arz etmektedir; çünkü her kenar minimum yayılan ağaçta değildir (Erickson, 2019).

1926'da Boruvka (Boruvka, 1926) tarafından keşfedilen algoritma ise en eski ve en basit minimum yayılan ağaç algoritmasıdır. Boruvka'nın Batı Moravia'daki tüm şehirleri birbirine bağlayan asgari maliyetle elektrik şebekesini tasarlamak için oluşturduğu matematiksel formul minimum yayılan ağaç algoritmasının temelini oluşturmuştur (Durnová, 2006). Algoritma 1960'ların başında bilgisayar bilimcisi olan Sollin tarafından yeniden keşfedildi ve buna paralel olarak hesaplama literatüründe genellikle bu algoritmaya "Sollin'in algoritması" denilmektedir (Chung ve Condon, 1996; Erickson, 2019) . Boruvka algoritmanın temel adımları Şekil 3.31'de gösterilmektedir.



Şekil 3.31. Boruvka algoritmasının temel adımları

Verilen ağırlıklı grafta tek tek düğümler arasındaki mesafeler karşılıklı olarak farklıdır. Bu aynı zamanda ağırlıklı bağlantılı bir grafiğin minimum yayılma ağacının benzersizliği için gerekli ve yeterli bir koşuldur (Durnová, 2006). Algoritma genel minimum yayılma ağacı algoritmasında olduğu gibi bir ara yayılan ormanı olan F'ye G grafındaki her bir düğüm (yayılan) bir ağacı temsil edecek şekilde eklenerek başlatılır. Her ağaç üzerinde bağlantılı olan kenarlardan en az ağırlıklı kenarı seçer. Bu şekilde her biri aynı kenarı seçen iki ağaç küme olarak adlandırılacak olan bağlı ağaç kümesini oluşturur. Ağaç kümeleri döngü olmayacak şekilde oluşturulmalıdır. Her ağaç kendi kümesi içerisinde olmayan fakat kendisi ile bağlantılı olan kenarlardan en az ağırlıklı kenarı seçerek döngü oluşmayacak şekilde birleştirir. Eğer F ormanında ağaç kümesi sayısı tek ise işlem sonlandırılır. Değilse işlem ağaç kümesi tek olana kadar devam eder. Algoritma aşağıdaki adımlardan oluşan yinelemeler sonucunda yayılan bir ağaç oluşturur:

Adım 1 : Bağlantılı, ağırlıklı ve yönsüz graf giriş olarak verilir.

Adım 2 : Her düğüm, üzerinde en az ağırlıklı kenarı seçer. Her düğüm ağacı temsil etmekte ve bağlantılı olan kenarlardan en az ağırlıklı olan kenarı seçmelidir. Her biri aynı kenarı seçen iki ağaç küme olarak adlandırılacak olan bağlı ağaç kümesi oluşturulur.

Adım 3 : Her ağaç kendi kümesi içerisinde olmayan fakat kendisi ile bağlantılı olan kenarlardan en az ağırlıklı kenarı seçer ve döngü oluşturulmayacak şekilde birleştirir.

Adım 4 : Yayılan ormanda ağaç küme sayısı eğer tek ise minimum yayılan ağaç elde edilir ve işlem sonlandırılır. Değilse Adım 2' ye gidilir.

3.4.2.2. Prim algoritması

Algoritma, 1930'da matematikçi Vojtèch Jarnik (Jarník, 1930) tarafından icat edildi ve daha sonra ayrı ayrı 1957'de bilgisayar bilimcisi Robert C. Prim (Prim, 1957) tarafından ve 1959'da Dijkstra (Dijkstra vd., 1959) tarafından yeniden keşfedildi . Prim'in algoritması, bağlantılı bir ağırlıklı yönsüz grafik için minimum bir yayılma

ağacı bulan açgözlü bir algoritmadır (Srivastava ve Tyagi, 2013; Marpaung, 2020). Açgözlü yaklaşımda, her adımda problemin küçük çözümleri elde edilerek en uygun çözüme ulaşılmaya çalışılır. Algoritmaya uyarlandığında, ağaçtaki tüm kenarların toplam ağırlığının en aza indirildiği her düğümü içeren bir ağaç oluşturan kenarların bir alt kümesinin bulunduğu anlamına gelir (Srivastava ve Tyagi, 2013) ve bir dizi genişleyen alt ağaç boyunca minimum yayılan bir ağaç oluşturulur (Levitin, 2011).

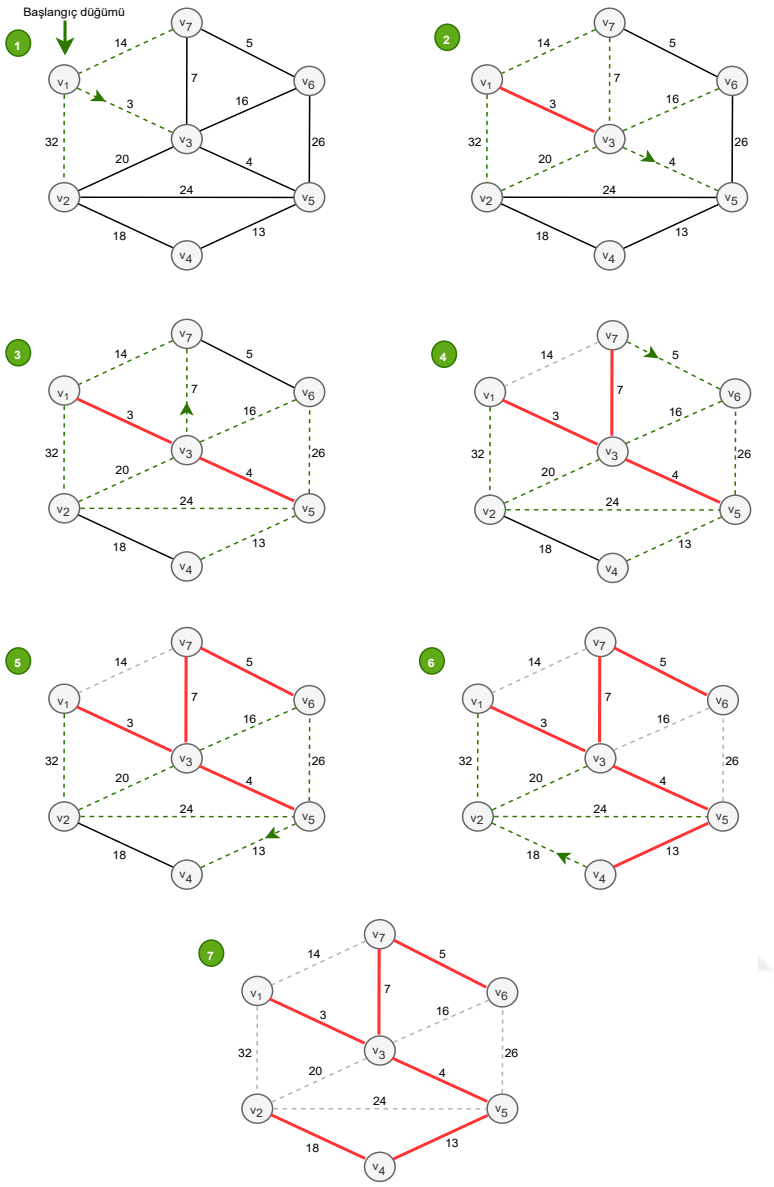
Prim algoritması, bir dizi genişleyen alt ağaç boyunca adım adım minimum yayılma ağacı oluşturur (Levitin, 2011). Böyle bir dizideki ilk alt ağaç, G grafiğinin düğümlerinin V kümesinden rastgele seçilen tek bir düğümden oluşur. Her yinelemede algoritma, açgözlü bir şekilde mevcut ağacı ağaçtaki bir düğüme en küçük ağırlıktaki bir kenarla bağlı olan ve ağaçta olmayan bir düğüme bağlayarak genişletir (Levitin, 2011). Bağlanan düğüm ağaca bitişik ve çevrimi olmayan kenara sahip olmalıdır (Srivastava ve Tyagi, 2013; Ramadhan vd., 2018; Marpaung, 2020). Algoritma bir ağacı yinelemelerinin her birinde tam olarak bir düğüm kadar genişlettiğinden bu tür yinelemelerin toplam sayısı $n-1$'dir; burada n, grafikteki düğüm sayısıdır. Algoritma tarafından oluşturulan ağaç, ağaç genişletmeleri için kullanılan kenarlar kümesi olarak elde edilir (Levitin, 2011). Prim algoritmasının temel adımları ile minimum yayılan ağaç yapısının elde edilmesi Şekil 3.32'de verilmekte olup Algoritma 1'de (Levitin, 2011) sözde kodu sunulmuştur. Algoritma adımlarındaki G, ağırlıklı grafiği temsil ederken, V

Algorithm 1: Prim algoritması

Giriş: $G = (V,E)$, Ağırlıklı bağlantılı yönlendirilmemiş bir grafik
Çıkış: E_t, Minimum yayılan G ağacını oluşturan kenarlar kümesi

1 $V_t \leftarrow v_0$ ▷ ağaç düğümleri kümesi herhangi bir düğüm ile başlatılabilir
2 $E_t \leftarrow \varnothing$
3 **for** $i \leftarrow 1$ *to* $|V|-1$ **do**
4 $v \in Vt$ ve $u \in V - Vt$ olmak koşuluyla bütün kenarlar (v,u) arasından minimum ağırlıklı kenarı $e^* = (v^*,u^*)$ bul
5 $V_t \leftarrow V_t \cup u^*$
6 $E_t \leftarrow E_t \cup e^*$
7 **return** E_t

ve E düğüm ve kenarları gösterir. İki düğüm arasında direk bir bağlantı yoksa sonsuz ∞ işareti ile gösterilir. Aksi durumda bağlantı ağırlığı verilir. Yöntemdeki u düğüm ağırlığına atıf yapar. Her düğüm için tüm bağlantılar arasından en küçük ağırlığa sahip



Şekil 3.32. Prim algoritmasının çalışma prensibi

kenar seçilerek ilerlenir. Seçilen tüm kenarlar birleştirildiğinde yeni bir grafik oluşur. e^* her hesaplamadaki en uygun kenar seçimidir. Bu seçimler birleştirilerek nihai grafik oluşturulur. Bu grafik E_t ile döndürülür. Prim algoritması aşağıdaki adımlardan oluşan yinelemeler sonucunda yayılan bir ağaç oluşturur:

Adım 1 : Bağlantılı, ağırlıklı ve yönlendirilmemiş graf giriş olarak verilir.

Adım 2 : Minimum yayılan ağaç (E_t) boş olarak atanır.

Adım 3 : Rastgele bir düğüm belirlenir ve minimum ağırlığa sahip ilgili kenar seçilir.

Adım 4 : Minimum ağırlığa sahip kenarları (v,u) ve E_t'ye komşu olanları seçin, ancak (v,u) E_t'de bir döngü oluşturmayacak şekilde (v,u) 'yi E_t 'ye ekleyin.

Adım 5 : Adım 4, $n-2$ kez tekrar edilir ve sonuç olarak minimum yayılan grafik E_t döner.

3.4.2.3. Kruskal algoritması

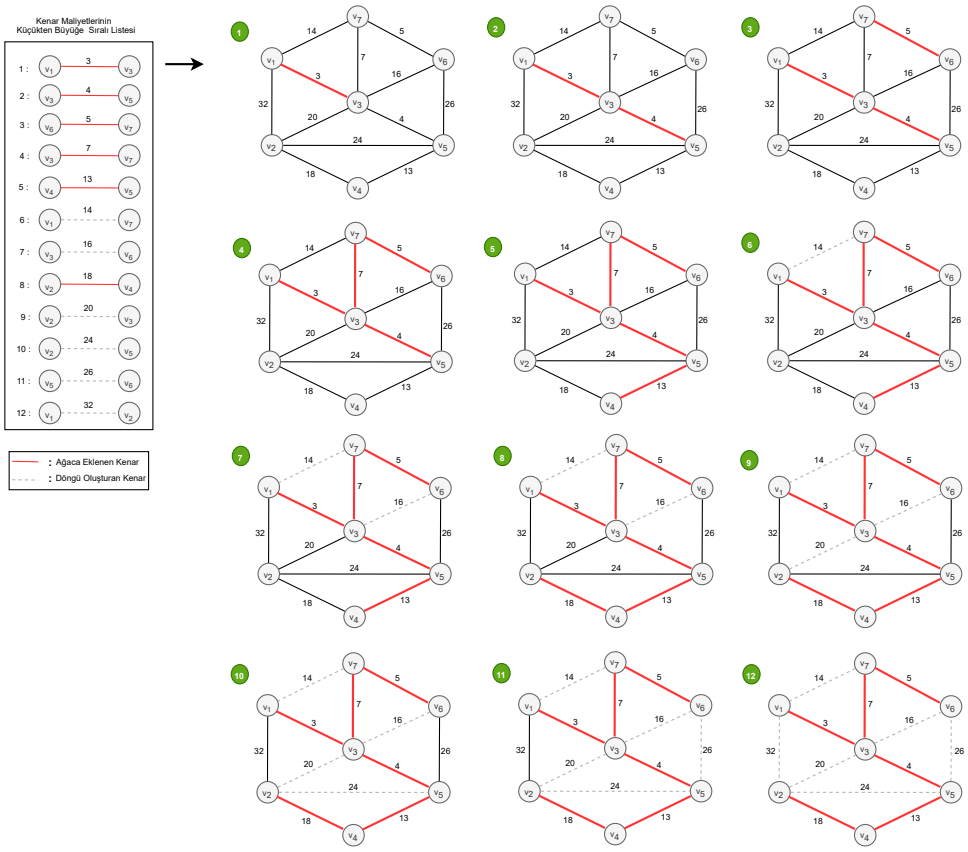
Algoritma Amerikan Matematik Derneği'nin bildirileri dergisinde 1956'da yayınlanmış ve Joseph Kruskal (Kruskal, 1956) tarafından yazılmıştır. Temel fikri, bir seferde $n-1$ kenar seçmek ve sonra açgözlülük tekniğini kullanmak olup seçilen sete katılacak kenarın döngü oluşturamayan minimum maliyetli kenar olmasıdır (Li vd., 2017). Çünkü seçilen kenar döngü oluşturursa bir yayılan ağaç oluşturulamaz. Bu algoritma, k adımlarına bölünmüştür; burada k, ağacın toplam kenar sayısıdır ve bir seferde bu k kenarlarının maliyet artırma sırasına göre yalnızca bir kenarı değerlendirilir. Bir kenar seçilir ve önceki seçilen kenarlara eklendiğinde döngü ortaya çıkarsa o kenar terk edilir. Aksi takdirde koleksiyon için seçim yapılır (Li vd., 2017). Kruskal algoritması her kenarı ağırlığına göre dikkate alır ve minimum yayılan ağaç bu koleksiyonda genişler.

Algorithm 2: Kruskal algoritması

Giriş: $G=(V,E)$, Ağırlıklı bağlantılı yönlendirilmemiş bir grafik
Çıkış: E_t, Minimum yayılan G ağacını oluşturan kenarlar kümesi

```
E sırala, kenar maliyetinin artan sırasına göre w(e_{i_1}) ≤ ··· ≤ w(e_{i_|E|})
E_t ← ∅; yolUzunluğu ← 0 ▷ ağacın kenarlar kümesini boş ata ve boyutunu sıfır ata
k ← 0                    ▷ işlenmiş kenar sayısını sıfır ata
while yolUzunluğu < |V| − 1 do
    k ← k + 1
    if E_t ∪ (e_{i_k}) Döngüsüz ise then
        E_t ← E_t(e_{i_k}) ; yolUzunluğu ← yolUzunluğu + 1
return E_t
```

$G=(V,E)$ n adet düğüme sahip bir bağlantılı grafı, $F=(V,\psi)$ kenarı olmayan bağlantısız bir grafı temsil etmektedir. G 'deki bütün kenarlar ağırlıklara göre küçükten büyüğe doğru sıralanır. G grafiğindeki her düğüm, bir ağacı veya bağlantılı bir bileşeni temsil etmektedir. En küçük ağırlıklı kenardan başlanır ve F'ye eklenir. Fakat eklenecek kenar döngü oluşturuyorsa kenardan vazgeçilir ve kalan kenarlardan minimum maliyetli kenar işleme alınır (Li vd., 2017). Bu durum tüm düğümler aynı ağaçta yer alana



Şekil 3.33. Kruskal algoritmasının çalışma prensibi

kadar devam eder ve düğümler adım adım birleştirilir (Kruskal, 1956). F için $n-1$ kenarları seçilmiştir, bu yüzden G 'nin $n-1$ kenarlarına sahip ağacı yayılan minimum maliyetlidir (Li vd., 2017). Kruskal algoritmasının temel adımları ile minimum yayılan ağaç yapısının elde edilmesi Şekil 3.33'te verilmekte olup Algoritma 2'de (Levitin, 2011) sözde kodu sunulmuştur. Kruskal algoritması aşağıdaki adımlardan oluşan yinelemeler sonucunda yayılan bir ağaç oluşturur:

Adım 1 : Bağlantılı, ağırlıklı ve yönlendirilmemiş graf giriş olarak verilir.

Adım 2 : Tüm kenarları E ağırlıklarına göre artan sırayla $w(e_{i_1}) \leq \cdots \leq w(e_{i_{|E|}}$ sıralanır.

Adım 3 : Minimum yayılan ağacın kenarlar kümesi (E_t) boş küme ve yol uzunluğunu sıfır olarak atanır.

Adım 4 : En küçük kenarı e_{i_k} seçilir.

Adım 5 : Şimdiye kadar oluşturulmuş yayılan ağaçla bir döngü oluşturup oluşturmadığı kontrol edilir. Döngü oluşturmuyor ise bu kenar da E_t 'ye dahil edilir.

Adım 6 : Adım 5, $(V-1)$ kez tekrar edilir ve sonuç olarak minimum yayılan graf E_t döner.

3.5. Enerjinin Genel Tanımı

Programların verimliliği genellikle büyüklük, hız ve enerji tüketimi yolu ile ölçülür (Bessa vd., 2019). Fakat geliştirilen yazılım programlarının karmaşıklığı, büyüklüğü ve donanımdaki ilerlemelerle ortaya konulan çok farklı yapılar, bilgisayarlarda performansın değerlendirmesini zorlaştırmanın yanısıra bilgisayarların verimli enerji tüketimini etkilemektedir. Enerji tasarrufu, çevresel etki ve artan maliyetler nedeniyle çok önemli bir konudur. Ayrıca kullanıcıların yüksek tüketim ve aşırı talepten kaynaklanan enerji ihtiyaçlarını karşılamak için yeşil özellikli yazılımlar geliştirilmeli ve böylece çevre üzerindeki olumsuz etkileri azaltılmalıdır.

Bilgisayar programları yürütüldüğünde enerji tüketirler. Enerji Uluslararası Birimler Sisteminde (SI) enerji birimi olan Joule (J) cinsinden ölçülür (Newell ve Tiesinga, 2019). Bir periyot boyunca harcanan elektrik gücü bu çalışma için enerjiyi açıklamaktadır. Denklem 3.12 (Newell ve Tiesinga, 2019), herhangi bir elektrikli cihaz tarafından tüketilen anlık gücü verir:

$$P = V * I \qquad (SIunits : Watt, Volt, Amper) \tag{3.12}$$

V, Volt cinsinden elektrik potansiyelini ve I, bir dirençten geçen elektrik akımını temsil eder. Enerji güç ve zamanın çarpımına eşittir. Formüller sırasıyla Denklem 3.13 ve Denklem 3.14'te verilmektedir:

$$E = P * T \qquad (SIunits : Joule, Watt, saniye) \tag{3.13}$$

$$E = V * I * T \qquad (SIunits : Joule, Volt, Amper, saniye) \tag{3.14}$$

Denklem 3.15 (Newell ve Tiesinga, 2019), elektrikli cihazlarda daha genel bir enerji tüketimi tanımıdır.

$$E = \int_{t2}^{t1} VcI(t)\,dt = Vc\int_{t2}^{t1} I(t)\,dt = Vc\int_{t2}^{t1} Vs(t)/Rs\,dt = Vc/Rs\int_{t2}^{t1} Vs(t)\,dt \quad (3.15)$$

Bu tanıma göre; T = t1-t2 tüketimi ölçmek için kullanılan integraldir ve Vc kararsız olmayan kaynak voltajını temsil eder. Rs bir direnç değeridir. I değerini elde etmek için, Ohm Yasası (I = Vs / Rs) kullanılır, Bu yasa, dirençteki V'leri ölçerek I değeri için Rs direncini kullanır. Bu işlemler, yürütüldüklerinde bilgisayar programları için enerji tüketimini veya güç tüketimini tahminlemeye yardımcı olurlar.

Joule cinsinden ölçülen enerji, bir zaman aralığında tüketilen toplam güçtür. Güç, yani enerjinin tüketilme hızı, statik ve dinamik gücün toplamıdır. Kaçak güç olarak da bilinen statik güç, devre aktivitesi olmadığında tüketilen güçtür. Dinamik güç, devredeki kapasitans yükünün şarj edilmesi ve deşarj edilmesinden kaynaklanan devre tarafından tüketilen güçtür (Hennessy ve Patterson, 2011). Yıllardır egemen güç kategorisi dinamik güç olmuştur ve Denklem 3.16 ile hesaplanır (Kaxiras ve Martonosi, 2008):

$$P_d = C * V^2 * A * f \quad (3.16)$$

Burada P_d dinamik güç, C kapasitans, V voltaj, A, aktif olan devrenin yüzdesini temsil eden aktivite faktörü ve f saat frekansıdır. Bir uygulamanın yürütülmesi sırasında bir sistemin güç tüketimi, bileşenlerin kendi güç tüketimine ve bileşenlerin nasıl kullanıldığına göre belirlenir (García-Martín vd., 2019).

4. GEREÇ ve YÖNTEM

4.1. Yeniden Düzenleme Tekniklerinin Seçimi

Yeniden düzenleme teknikleri genellikle kodlama türüne bağlı olarak proje kaynaklarına uygulanır. Bu sürecin temel amacı okunabilirlik, performans ve bakım dahil olmak üzere yazılımın bazı özelliklerini iyileştirmektir. Bazı entegre geliştirme ortamları, uygulayıcıların bu özellikleri kontrol etmesini sağlar. Dahası, geliştiricilerin isteklerine bağlı olarak bazı karmaşık araçlar tasarlanır. Verilen bilgiler ışığında enerji tüketimi açısından önemli görülen yedi yeniden düzenleme tekniği tez çalışması kapsamında seçilmiştir. Ayrıntılar Çizelge 4.1'de verilmiştir.

Çizelge 4.1. Seçilen yeniden düzenleme teknikleri listesi

Gösterim	**Açıklama**
R_1	Encapsulate Field
R_2	Simplify Nested Loop
R_3	Inline Temp
R_4	Introduce Explaining Variable
R_5	Replace Magic Number with Symbolic Constant
R_6	Consolidate Duplicate Conditional Fragments
R_7	Hide Method

4.2. Yazılım Metriklerinin Seçimi

Fowler birçok farklı türden yeniden düzenleme teknikleri sunarken (Fowler vd., 1999), bunların büyük sistemlere uygulanacağında ortaya çıkan temel sorunlardan biri hangi yeniden düzenleme tekniklerinin nereye uygulanacağı sorusu olmuştur (Simon vd., 2001). Yeniden düzenleme tekniklerinin insan sezgisine ve koku gibi öznel algılara dayandığına dair Fowler'ın ifadeleri bu soruyu daha da zorlaştırmaktaydı. Hatta yeniden düzenlemenin otomatik yapılamayacağı izlenimini vermekteydi. Fowler açıkça metriklere atıf yaparak insan sezgisini hiçbir metriğin bilgilendirmediğinden bahsetmekteydi (Fowler vd., 1999). Fakat geliştiriciler metriklerin yeniden düzenleme

uygulamasına yardımcı olabileceklerini çalışmalarında göstermektedirler (Simon vd., 2001, Mens and Tourwé, 2004, Bodhuin vd., 2007, Alshayeb, 2009, Cinnéide vd., 2012, Bavota vd., 2015, Hegedűs vd., 2018). Geleneksel ve nesne yönelimli metrikler, yazılımın sürdürülebilirliği (Coleman vd., 1994; Najm, 2014; Hegedűs vd., 2018) ve enerji verimliliği çalışmalarında (Cruz ve Abreu, 2017; Keong vd., 2015; Bangash vd., 2017; Ergasheva vd., 2020; Mancebo vd., 2021a) tercih edilmektedir. Tez çalışması kapsamında, geleneksel ve nesne yönelimli metrikler yeniden düzenleme işlemlerinin yazılım sürdürülebilirliği ve enerji verimliliği üzerindeki etkilerini değerlendirmek için dahili metrikler olarak seçilmiştir. Seçilen metriklerin ayrıntıları Çizelge 4.2'de verilmiştir.

Çizelge 4.2. Seçilen metriklerin listesi

Gösterim	**Açıklama**
M_1	Kod Satır Sayısı
M_2	Mantıksal Kod Satır Sayısı
M_3	Çevrimsel Karmaşıklık (McCabe)
M_4	Operand ve Operatör sayısı
M_5	Bağlaşım Sayısı
M_6	Bakım Yapılabilirlik İndeksi
M_7	Kalıtım Ağacının Derinliği
M_8	Fonksiyon Başına Düşen İfade sayısı
M_9	Fonksiyon Çağrılma Sayısı
M_{10}	Fonksiyon Sayısı

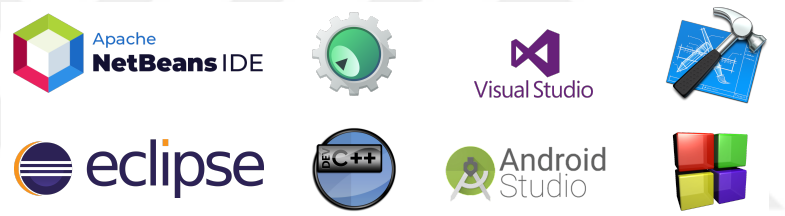
4.3. Yazılım Geliştirme Ortamının Seçimi

Tümleşik geliştirme ortamı kodlama, test etme, hata ayıklama gibi özellikleri ile bilgisayar programcılarına yazılım geliştirebilme imkanı veren ve yazılım geliştirme sürecini verimli bir şekilde kullanılmasını sağlamak için çeşitli araçları içerisinde barındıran yazılım paketleridir (Zeil, 2017). Tümleşik geliştirme ortamlarında bulunması gerekli temel başlıca özellikler aşağıda verilmektedir:

- Tümleşik bir derleyici, yorumlayıcı ve hata ayıklayıcı,

- Görsel kod yazım editörü,
- Kod dosyalarının hiyerarşik gösterimi
- Editör tarafından yakalanan, yorumlanan ve yürütülen hata mesajları
- Yazılımın derlenmesi, bağlanması, çalıştırılıp yürütülmesi
- Otomatik yeniden düzenleme desteği.

Tümleşik geliştirme ortamları yukarıda bahsedilen özelliklerin yanında birçok yönüyle (kod tamamlama araçları, grafik tasarımı vs.) yazılım geliştirme için kullanılabilen zengin bir programdır. En bilinen tümleşik geliştirme ortamları: NetBeans, Eclipse, KDevelop, Dev-C++, Microsoft Visual Studio, Android Studio, XCode, Code::Blocks (Zeil, 2017).



Şekil 4.1. Tümleşik geliştirme ortamı örnekleri

Microsoft Visual Studio (Microsoft, 2021), Microsoft yazılım şirketi tarafından geliştirilen bir tümleşik geliştirme ortamıdır. İçerisinde Visual Basic, Visual C#, C++ gibi farklı nesne yönelimli programlama dillerinin tümleşik geliştirme ortamlarını ve özelleştirilmiş kütüphaneleri barındırmaktadır. Gelişmiş kullanıcı arabirimi tasarımı üzerinden yazılım oluşturma, kodlama, test, hata ayıklama, kod kalitesini ve performansını çözümleme olanağı sunulmaktadır. Tez kapsamında geliştirilen algoritmanın nesne yönelimli programlama dilleri ile yazılmış kodlar ile uyumlu olması, yeniden düzenleme tekniklerinin etkileşimlerinin tespiti, geliştirilen otomasyon yazılımın kalitesinin korunabilmesi ve yeniden kullanılabilirliği artırmak için Microsoft Visual Studio tez çalışmasında kullanılmak üzere tercih edilmiştir. Ayrıca tez ile ilgili çalışmalar nesne yönelimli programlama dillerine ait bulguları kapsamaktadır. Bunun dışındaki programlama yapıları örneğin endüstriyel otomasyon yazılımları ve mantıksal programlama tezin kapsamı dışındadır.

4.4. Geliştirilen Algoritma ve Matematiksel Model

Tez çalışması kapsamında geliştirilen algoritmanın yazılım enerji tüketimini düşürebilmek için uygun yeniden düzenleme tekniklerinin sırasının oluşturulması öncelikli amaç olarak planlanmıştır. Geliştirilen algoritma özelinde problemin uygun bir başlangıç çözümüne uygulanır. Uygulama sonucunda başlangıç sonucundan daha iyi bir sonuç elde edilmektedir. Bu yeni sonuç başlangıç noktası kabul edilerek yeni bir çözüm elde edilmesine geçilir. Bu yüzden geliştirilecek algoritmanın uygulanabilir bir yöntem olabilmesi için yeniden düzenleme tekniklerinin iyi analiz edilip uygun bir amaç fonksiyonu haline getirilmesi gerekmektedir. Diğer taraftan, bir yeniden düzenleme sırasının üretildiği bir grafikte bir düğümün maliyetine karar vermek için matematiksel model geliştirilmesi planlanmıştır. Geliştirilen matematiksel model yazılım sürdürülebilirliği ile yazılım metrikleri zıtlık ilkesi (Candela vd., 2016; Tarwani ve Chug, 2016) doğrultusunda kompleks metrik hesabından oluşmaktadır. Bu bilgiler ışığında geliştirilen algoritma ve matematiksel model sırasıyla anlatılmaktadır.

4.4.1. Geliştirilen algoritma

Çalışma bağlamında, yeniden düzenleme işlemini kaynak kodlara uygulamak için kullanılacak en uygun sırayı bulan Prim tabanlı bir algoritma önerilmektedir. Önerilen algoritma, üç ana bölümden oluşmaktadır: 1) Projelere uygulanabilecek yeniden düzenleme tekniklerinin tespiti, 2) Prim tabanlı algoritmanın çalıştırıldığı grafiği oluşturan düğümler için maliyetlerin hesaplanması, 3) Yeniden düzenleme tekniklerinin sırasının üretilmesi. Önerilen algoritmanın adımları Algoritma 3'te verilmiştir. Algoritma 3, nesne yönelimli programlamadan oluşan proje sınıflarına ait *SC* kaynak kodların girdi olarak alınmasıyla başlar. Enerji tüketimi ile ilgili olarak en uygun yeniden düzenleme tekniklerinin sırasının çıktısını R_{order} temsil etmektedir. Sınıfların yeniden düzenleme teknikleriyle uyumluluğu 1. ile 12. satırlar arasındaki alt yordamlarda hesaplanmıştır. Projenin her bir sınıfı, n'nin sınıf sayısını temsil ettiği SC_i ile gösterilir. R_i SC_i için uygulanabilir teknik ise, L log dosyasına bir yeniden düzenleme tekniği R_i eklenir. Log, yeniden düzenleme tekniklerinin hareketlerin kayıt altına alınması için kullanılır. Satır 10-14'de verilen iç içe döngüler, her sınıfa ayrı ayrı yeniden düzenleme tekniklerini uygular ve yeniden düzenlenen sınıfları kaydeder. Daha

sonra yeniden düzenlenen sınıflar için düğümlerin maliyetleri, çalışma bağlamında tasarlanan matematiksel bir model aracılığıyla hesaplanır. R, sınıflara uygulanan bir yeniden düzenleme tekniğini belirtirken, SR_{ik} projenin yeniden düzenlenmiş bir sınıfıdır. Düğümlerin maliyetleri Denklem 4.1 ile hesaplanır. Düğümlerin grafiği, her bir düğümün bir R'yi temsil ettiği ve E'nin iki komşu düğümün ortalama maliyetini temsil ettiği gözönüne alınarak oluşturulmaktadır. R_s, grafikte bir köşe seti oluşturan yeniden düzenleme tekniklerinin listesini gösterir. Rastgele başlatma r_0 aracılığıyla gerçekleştirilir. R_s ve r_0 Satır 15'te temsil edilir. o, yeniden düzenleme teknikleri listesine eklenecek yeni adaydır. E_{seq}, sıralı kenar kümesini temsil eder. Yinelemeli hesaplama 18-22. satırlar ile sürdürülür. İşlem, tüm kenarlar ve düğümler ziyaret edilene kadar devam eder. E_{seq}, yeniden düzenleme teknikleriyle iki kat artırılırsa, benzersiz bir E_{seq} elde etmek için budanır. Son olarak, yeniden düzenleme tekniklerinin son sırasını içeren R_{order}, Satır 24'te döndürülür.

Algorithm 3: Geliştirilen önceliklendirme algoritması

Giriş: SC , Kaynak kod

Çıkış: R_{order} , Optimal sıra

$i \leftarrow say(SC)$

$j \leftarrow 7$

while $i > 0$ **do**

    while $j > 0$ **do**

        $L_i \leftarrow uygula(R_j)$

        $j \leftarrow j - 1$

    $i \leftarrow i - 1$

$k \leftarrow say(L)$

$i \leftarrow say(SC)$

while $i > 0$ **do**

    while $k > 0$ **do**

        $SC_i \leftarrow uygula(R_k)$

        $SR_{ik} \leftarrow maliyetHesapla(SC_i)$

    $i \leftarrow i - 1$

$R_s \leftarrow r_0$

$E_{seq} \leftarrow \varnothing$

$k \leftarrow say(L)$

while $k > 0$ **do**

    $e^* = (r^*, o^*)$ ▷ $MinimumMaliyetliKenarBul(r, o)$

    $k \leftarrow k - 1$

    $R_s \leftarrow R_s U o^*$

    $E_{seq} \leftarrow E_{seq} U e^*$

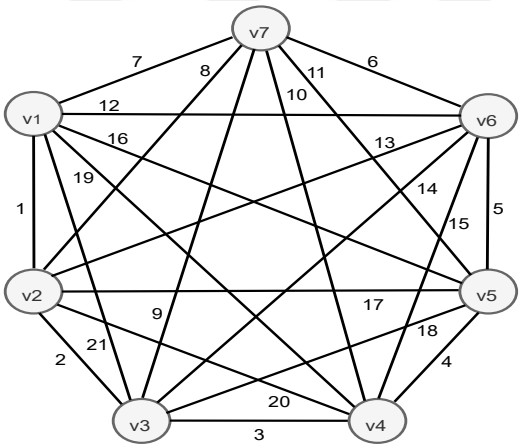
$R_{order} \leftarrow tekrarlarıBuda(r, E_{seq})$

return R_{order}

4.4.2. Geliştirilen matematiksel model

Tanım 4.1. (Yeniden düzenlemenin graf gösterimi). Bir yeniden düzenleme listesi $G = (V,E)$ ile temsil edilebilir. G grafı, boş olmayan iki sonlu küme V ve E 'den oluşan bir matematiksel yapıdır (Biggs vd., 1986). Grafta, V'nin elemanlarına düğümler (vertices) denir ve $V = (v_1, v_2, \ldots . v_n)$ düğümler kümesi ile gösterilir. E'nin elemanlarına kenarlar (edges) denir ve $E = (e_1, e_2, \ldots . e_m)$ kenarlar kümesi ile gösterilir (Gross ve Yellen, 2005). Burada V, yeniden düzenleme tekniklerini içeren bir düğümler kümesi; E, düğümler arasındaki bağlantıları içeren bir kenar kümesidir. n kenar sayısı olsun; $n * (n-1)/2$ kenar sayısına sahip tamamen bağlantılı bir graf oluşturmak hedeflenmektedir. Her yeniden düzenleme tekniğinin, büyüklüğü kaynak kod analizine bağlı olan bir w ağırlığı vardır. w, bir düğüm çiftinin (r,o) iki maliyetinin ortalaması alınarak hesaplanır.

Yukarıdaki tanım, Şekil 4.2'de gösterildiği gibi tam ağırlıklı bir graf oluşturur.



Şekil 4.2. Tam ve ağırlıklı graf örneği

Tanım 4.2. (Yolun oluşturulması). Düğümlerden biri seçilerek V'nin rastgele başlatılması tamamlanır. Kenar listesi e^*, düğümlerin adım adım kapsamlı bir değerlendirmelerinden alınan kenarlarla doldurulur. Sonuç olarak, bir son kenar listesi e^* elde edilir.

Öneri 4.3. (Yolun yürütülmesi). Bir G verildiğinde, düğümlerin her maliyeti bir karmaşıklık formülü ile temsil edilebilir. Düğümlerin maliyetinin belirlenmesinin, G üzerinde güvenilir bir yol ortaya çıkarmak için çok önemli olduğuna dikkat

edilmelidir. G'nin tüm düğümlerini kapsayan yol inşa edilir, böylece maliyetlerin her bir değerlendirmesinde bunlardan bazıları budanır. Nihai kenar listesi e^*, kenar sayısından daha kısa olmalıdır ((e^*)<(n*(n-1)/2). Rastgele e^*'nin, sıralı bir e^*'ye kıyasla daha yüksek enerji tüketimine yol açtığı çalışmada öne sürülür.

Yazılım metrikleri yazılım geliştirme sürecini ve yazılım ürünlerinin kalitesini kontrol etmek için nicel bir yol sağlar (Li ve Henry, 1993) ve yazılımın karmaşıklık, büyüklük, kalıtım gibi belirli yönlerini ölçer. Genel olarak, yazılımın dahili nitelikleri yazılım metrikleri aracılığıyla yakalanır ve üst düzey özellikleri bu metrikler için geçerli değerler cinsinden ifade edilir (Mahouachi vd., 2013). Yazılım metrik çeşitlerinden biri olan nesne yönelimli metriklerin kullanımı kaynak kodunda özellikle yazılım görselleştirmesiyle birlikte yeniden yapılanmaya ihtiyaç duyan yerleri tespit etmek için çok uygundur (Kaur ve Singh, 2016). Bu nedenle, kod karmaşıklığı ve boyutu hesaplayabilen metrikler ana kriter olarak seçilir. Bu çalışmada geleneksel ve nesne yönelimli metrikler, yeniden düzenleme işleminin kod kalitesi üzerindeki etkisini değerlendirmek için dahili metrikler olarak seçilmiştir.

Diğer taraftan yazılım sürdürülebilirliği ile yazılım metrikleri arasında genelde ters ilişki olduğu bilinmektedir (Tarwani ve Chug, 2016). Nesne yönelimli programlamada ise yapışıklık (cohesion) ve bağlaşım (coupling) metrikleri yazılımın sürdürülebilirliği açışından iki zıt hedeftir. Yüksek kaliteli moduler yazılımlar elde etmek için yapışıklık maksimizasyonu ve bağlaşım minimizasyonu önemlidir; fakat bu metriklerin yazılım kalitesini değerlendirmede tek başlarına yeterli olmayıp yazılımın boyutu, karmaşıklığı gibi faktörler ile birlikte ele alınması gerekmektedir (Candela vd., 2016). Denklem 4.1'de verilen hesaplama, bir yeniden düzenleme sırasının üretildiği bir grafikte bir düğümün maliyetine karar vermek için tasarlanmıştır.

$$W_c = \frac{(M_a + M_b + M_c + (M_d * M_e))}{M_g} \tag{4.1}$$

W_c, bir düğümün maliyetini belirtir. M_a, McCabe'nin çevrimsel karmaşıklık değeridir ve bir düğümün maliyeti ile doğru orantılıdır. M_b, kod satır sayılarını temsil eder. M_c, kalıtım ağacının derinliği ve M_d fonksiyon sayısıdır. M_e fonksiyon çağrılma sayısı, düğüm maliyetini maksimum seviyede tutmak amacıyla M_d ile çarpılırak payın ikinci

bölümünü oluşturmaktadır. Bakım yapılabilirlik indeksi olan M_g ise düğüm maliyetini azaltmak için kullanılır ve paydayı oluşturur.

4.5. Deneysel Veri Seti

Sunulan algoritma ve yöntem için geliştirilen aracımızın nesne yönelimli kaynak kodlara ihtiyacının olmasının yanında yasal kısıtlamalar göz önüne alınarak seçilen projeler açık kaynak kodlu projeler olacak şekilde sınırlandırılmıştır. Böylece istenen değişikliklere izin veren ve nesne ilişkilerini içeren kodlar, analiz edilmek için veri setine dahil edilmiştir. Taşınabilir cihazlardaki oyunlar enerji yoğun uygulamalar olduğu için (Zotos vd., 2005) birkaç yıldır var olan ve aktif bir kullanıcı topluluğuna sahip oyun tabanlı yerleşik uygulamalar veri seti seçiminde önemli bir kriter olarak belirlenmiştir. Genel olarak, veri aktarımı enerji tüketiminin ana kısmını oluşturur (Sun vd., 2017; Lyu vd., 2019). Bu bilgi doğrultusunda yapılandırılmış sorgu dili içeren projelerin seçimi diğer önemli bir kriter olarak belirlenmiştir. Bir diğer kriter ise, tekniklerin bir kaynak kod içinde birkaç farklı bölüme uygulanması halinde enerji tüketimini azaltmak mümkün olduğundan uygulamaların en az üç yeniden düzenleme tekniği ile uyumlu olması durumudur (Kim vd., 2018). Çok büyük projelerde (web tabanlı, bulut tabanlı) geliştirilen aracın yazılım güvenilirliği (Triwijoyo vd., 2017) düşük olduğundan öncelikli bir yeniden düzenleme listesi vermenin pratik olmamasıyla sonuçlanmaktadır. Bu yüzden deneysel veri setleri proje boyutu açısından sınırlandırılmış ve düz kodlar analiz edilmeye çalışılmıştır. Tez, nesne yönelimli programlama için Java ve C# uygulamalarına odaklanmakta ve bu dillerin bulgularını kapsamaktadır. Bu iki dil yazılımın sürdürülebilirliği için analiz edilen araçlarla uyumlu olduklarından (Ardito vd., 2020) ve enerji verimliliği çalışmalarında tercih edildiğinden (Mancebo vd., 2021b; Pereira vd., 2021) sıklıkla kullanılırlar. Ayrıca benzer sözdizimine sahiptirler ve bu iki dilde yazılan ortak amaçlara sahip uygulamalar, sınıf kapsülleme, polimorfizm ve yeniden kullanılabilirlik açısından benzer özellikler gösterir (Ogala ve Ojie, 2020). Fakat enerji verimliliği açısından uygulamalarda benzerlik gösterip göstermediklerinin tespiti için kapsamlı analizler gerekmektedir.

Seçilen projeler sekiz adet açık kaynaklı C# ve Java programlama dili ile oluşturulmuş yazılım projeleridir. Seçilen tüm çalışmalar çeşitli perspektiflerden analiz edilmiş ve

çalışmalar arasında herhangi bir ilişki tespit edilmemiştir. Projelerin içsel davranışları minimum üç yeniden düzenleme tekniği içerecek şekilde düzenlenmiş ve çalışma kapsamında düzenlenen bu kodlar orijinal kod olarak ele alınmıştır. Sınıf başına hesaplama işlemlerinin yapıldığı projelerin detayları hakkında bilgiler Çizelge 4.3'te verilmektedir.

Çizelge 4.3. Deneysel projelerin özeti

Ad	**Dosya Sayısı**	**LOC**	**SLOC-P**	**SLOC-L**	**CC**
Bordro Yönetim Sistemi Projesi [a]	17	9176	7833	5105	789
Otel Yönetim Sistemi Projesi[a]	20	6210	5090	3471	472
Basit Hesap Makinesi Projesi[b]	6	674	490	351	15
Hastane Yönetim Sistemi Projesi[c]	16	2147	1611	1276	55
Çalışan Yönetim Sistemi Projesi[c]	24	2892	2205	1743	50
Oyun 2048 Projesi[c]	6	1466	1145	808	141
Mobil-Hesap Makinesi Projesi[d]	10	637	419	317	46
Mobil-Oyun 2048 Projesi[e]	17	1204	938	698	132

LOC: Kod satır sayısı, SLOC-P: Fiziksel kod satır sayısı, SLOC-L: Mantıksal kod satır sayısı, CC: Çevrimsel karmaşıklık (McCabe)

[a] *Web Sayfası : https://www.kashipara.com/project/projectcsharp.php*

[b] *Web Sayfası : https://github.com/Charpur98/Simple-Calculator*

[c] *Web Sayfası : https://code-projects.org/c/languages/project/c-sharp-projects/*

[d] *Web Sayfası :https://github.com/RushanB/Calculator*

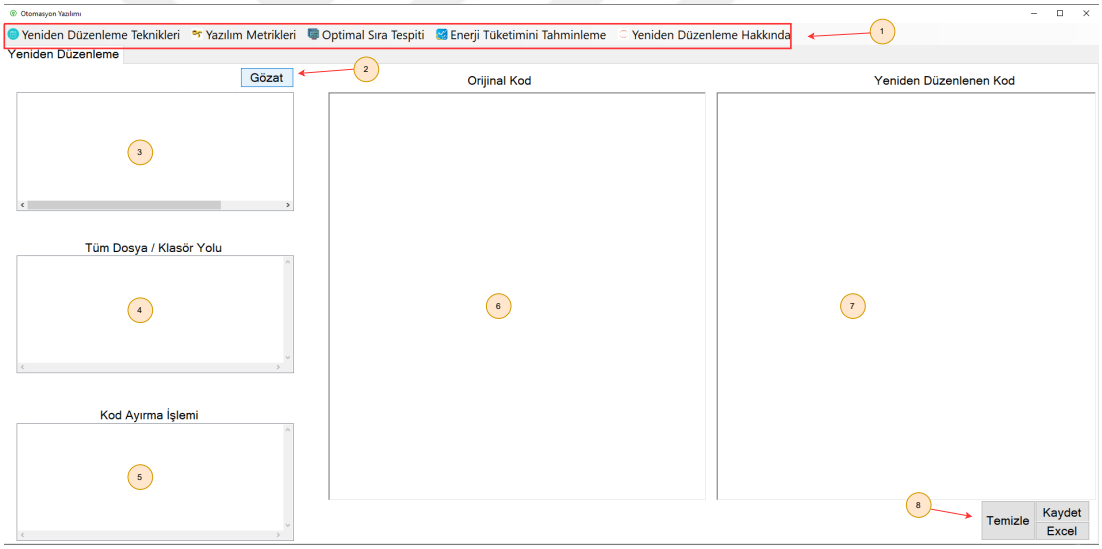
[e] *Web Sayfası :https://github.com/roxrook/2048-android*

Bordro Yönetim Sistemi, Otel Yönetim Sistemi, Hastane Yönetim Sistemi, Çalışan Yönetim Sistemi, Basit Hesap Makinesi ve Oyun 2048 projeleri C# programlama dili ile oluşturulmuş masaüstü uygulamalarıdır. Mobil-Hesap Makinesi ve Mobil-Oyun 2048 projeleri Java programlama dili ile oluşturulmuş Android uygulamalarıdır.

4.6. Geliştirilen Otomasyon Yazılım Aracının Genel Özellikleri

Tez kapsamında geliştirilen algoritmanın uygulanması için Windows tabanlı grafiksel kullanıcı arayüze sahip otomasyon yazılım aracı geliştirildi. Bu aracın temel özellikleri, nesne yönelimli bir programlama dili olan C# ile geliştirilen bir uygulamanın enerji tüketimini iyileştirmek için yeniden düzenleme dizileri elde etmenin yanında yazılım geliştiricilere yazılım kalitesi ve enerji tüketimi açısından destek vermektir. Geliştirilen araç, Encapsulate Field, Simplify Nested Loop, Inline Temp, Introduce Explaining Variable, Replace Magic Number with Symbolic Constant, Consolidate

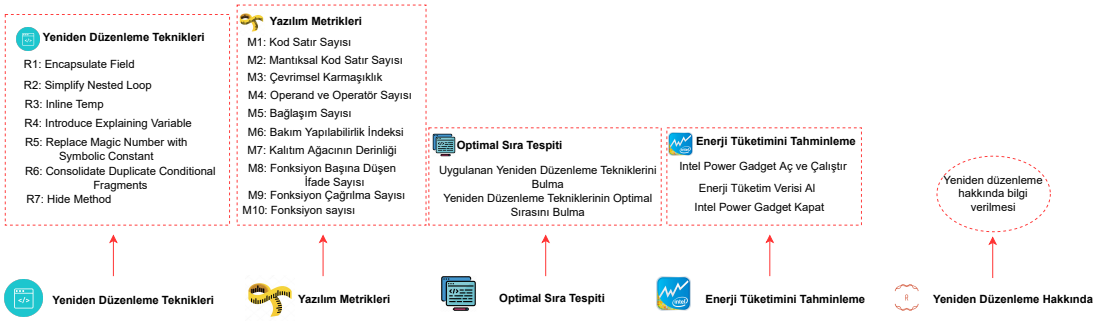
Duplicate Conditional Fragments ve Hide Method yeniden düzenleme tekniklerini enerji tüketimini düşürmeye yönelik olumsuz etkileri olacağı düşünülen kodlara uygulayabilmekte, nesne yönelimli metrikler ve geleneksel metrikler yardımıyla kod ölçümlerini yapabilmekte, enerji tüketimini optimum seviyeye indirmeye yarayacak yeniden düzenleme tekniklerin sırasını üretebilmekte ve masaüstü kullanıcıları için enerji tüketim ölçümü gerçekleştirip görsel olarak kullanıcılara sunabilmektedir. Çalışma kapsamının geri kalanında geliştirilen araç olarak anılacaktır. Önerilen algoritmanın nesne yönelimli programlama dillerinde yazılan kodlarla uyumlu olmasını sağlamak, yeniden düzenleme tekniklerinin etkileşimlerini tespit etmek, geliştirilen yazılımın kalitesini korumak ve artırmak amacı ile geliştirilen araç için Microsoft Visual Studio geliştirme ortamı tercih edilir. Yazılım, nesne yönelimli bir programlama dili olan C# ile kodlanmıştır. Kullanıcı arayüz tasarımı Şekil 4.3'te verilmiştir.



Şekil 4.3. Kullanıcı arayüz tasarımı

Kullanıcı açılış ekran arayüzü 8 bileşenden oluşmaktadır. Şekil 4.3'te 1 numaralı alan içerisinde alt menüler içeren menu item kontrolleri mevcuttur. 2 numaralı alan Gözat butonunu, 3, 4, 5 numaralı alanlar listview kontrollerini, 6 ve 7 numaralı alanlar richTextBox kontrollerini ve 8 numaralı alanlar ise Kaydet, Temizle ve Excel buton kontrollerini işaret etmektedir. Arayüz bileşenlerinin fonksiyonları sırasıyla anlatılmaktadır.

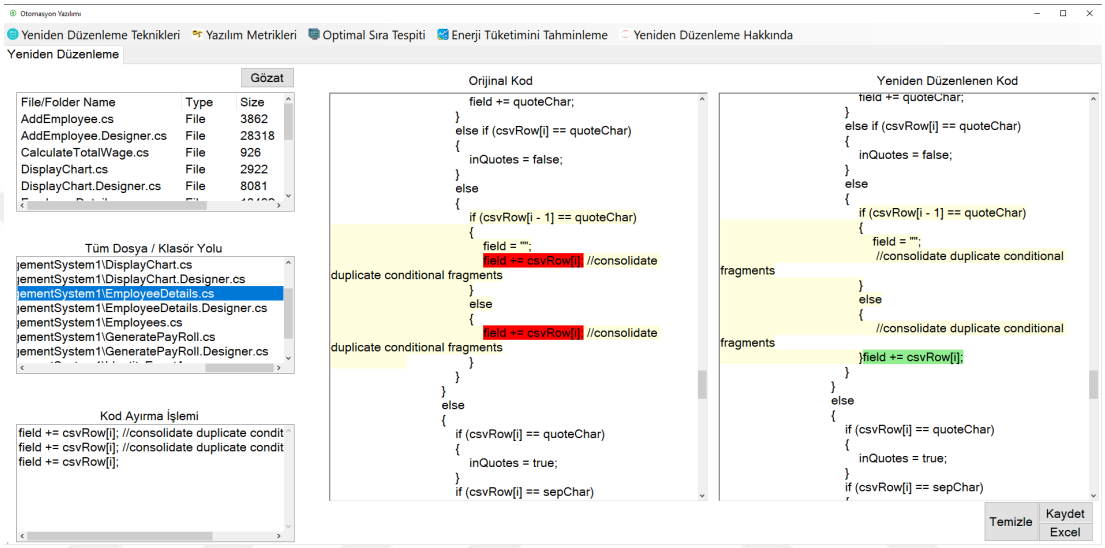
Numara 1 olarak adlandırılan alan 5 ana bileşen içermektedir: Yeniden Düzenleme Teknikleri, Yazılım Metrikleri, Optimal Sıra Tespiti, Enerji Tüketimini Tahminleme and Yeniden Düzenleme Hakkında. Bu bileşenlerin alt menüleri Şekil 4.4'te verilmektedir.



Şekil 4.4. Alt menülerin genel görünümü

Yeniden Düzenleme Teknikleri isimli menuItem nesnesinin aktif olabilmesi için Gözat (2 numaralı) buton kontrolünün C# veya Java programlama dilleri ile oluşturulmuş ".cs" veya ".java" uzantılı dosyalardan veri alması gerekmektedir. Dosyalardan elde edilen veriler kaynak kodları temsil eder. Seçilen kaynak kodların listview kontrollerinde, boyutları, özellikleri (3 numaralı) ve yol uzantıları (4 numaralı) listelenirken veri görselleştirme tablosuna da aktarımı yapılır. Kullanıcı, 4 numaralı Tüm Dosya/Klasör Yolu adlı listView denetiminden düzenlemek istediği orijinal kodun uzantısını seçmelidir. Nesne yönelimli bir programlama dili olan C# veya Java ile oluşturulan kaynak kodlar orijinal Kod (numara 6) richTextBox kontrolüne yüklenir. RichTextBox denetimi, bir denetim kümesi içindeki metnin bir bölümünü biçimlendirmek için kullanabileceğiniz özelliklere sahip olmasının yanısıra dosyaları kaydetmek ve açmak için işlevler sağlayan yöntemleri de içerir. Daha sonra Şekil 4.4'te verilen Yeniden Düzenleme Teknikleri adlı alt menuItem kontrol menüsünden kaynak koda uygulanacak yeniden düzenleme tekniği seçilip uygulandıktan sonra elde edilen orijinal ve yeniden düzenlenen kodlar Şekil 4.5'te gösterildiği gibi 6 ve 7 numaralı richTextBox kontrollerinde kullanıcıya sunulmaktadır. Yeniden düzenleme tekniklerinin uygulanması sırasında Microsoft Kod Analiz kütüphanesinin yanısıra metin parçalama, noktalama işareti silme, durdurma kelimeleri, temel hale döndürme, cümle bölümlendirme, kökenine döndürme, dizin (trim, indexof, split, substring) teknikleri ve düzenli ifadelerden yararlanılmıştır. Ayrıca yer değiştirme (replace) ve dosya yazma yöntemleri ile bulunan kod bloğu yeni özelliklere sahip kod bloğu ile değiştirilmektedir. Sınıflara uygulanan yeniden

düzenleme tekniklerinin log kayıtlarında tutulması için Kaydet butonunun click eventinin aktif olması gerekmektedir. Kaydedilen ve yeniden düzenlenen kodların değişen metrik değerlerinin bireysel hesaplaması Şekil 4.4'te verilen metrik menüsü yardımıyla yapılmaktadır. Excel butonu yardımıyla excel dosyasına ölçüm sonuçları kaydedilmektedir. Elde edilen ölçüm sonuçları graf tabanlı arama yaklaşımındaki her bir düğümü temsil eden yeniden düzenlenen kodlar için elde edilecek düğüm maliyetinde kullanılmaktadır. Düğüm maliyet hesaplaması geliştirilen matematiksel modeldeki



Şekil 4.5. Orijinal ve yeniden düzenlenmiş olanları kapsayan kod parçası örnekleri

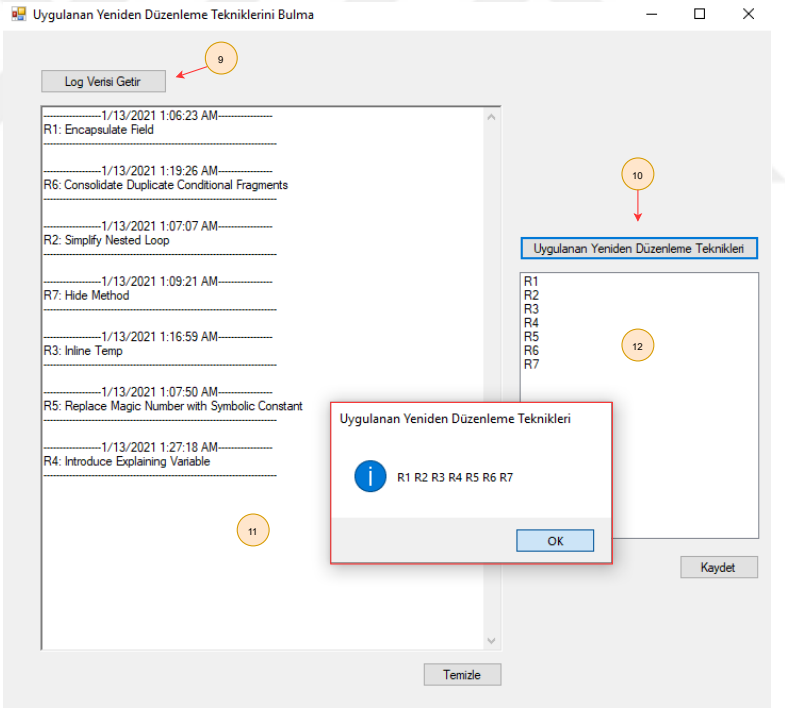
kompleks metrik hesaplaması ile elde edilmektedir. Ayrıca, elde edilen nihai kompleks metrik ölçüm değerleri ve uygulanan yeniden düzenleme listesi kullanılarak her sınıfa ait komşuluk matrisi oluşturulmaktadır. Geliştirilen araç sayesinde R GUI script (R, 2021) olarak bu matrisler üretilmekte ve en optimal sırayı bulmak için Prim tabanlı algoritma ile birlikte kullanılmaktadır. Oluşturulan bir sınıfa ait komşuluk matris örneği Çizelge 4.4'te verilmektedir.

Optimal Sıra Tespit menü öğesi iki alt ana menü öğesi içermektedir: Uygulanan Yeniden Düzenleme Teknikleri Bulma, Yeniden Düzenleme Tekniklerinin Optimal Sırasını Bulma. Log kullanarak uygulanan kod yeniden düzenleme tekniğini bulma menu bileşeninin click eventi aktif hale geldiğinde Şekil 4.6'da verilen form tasarımı kullanıcıya sunulmaktadır. Bu alt menü her sınıfa ait log dosyalarını kullanarak o sınıflara uygulanan yeniden düzenleme teknikleri listesini üretmekte ve kullanıcılara yansıtmaktadır. 9 numaralı open log data butonu yardımıyla sınıfa uygulanan yeniden

Çizelge 4.4. Komşuluk matris gösterimi

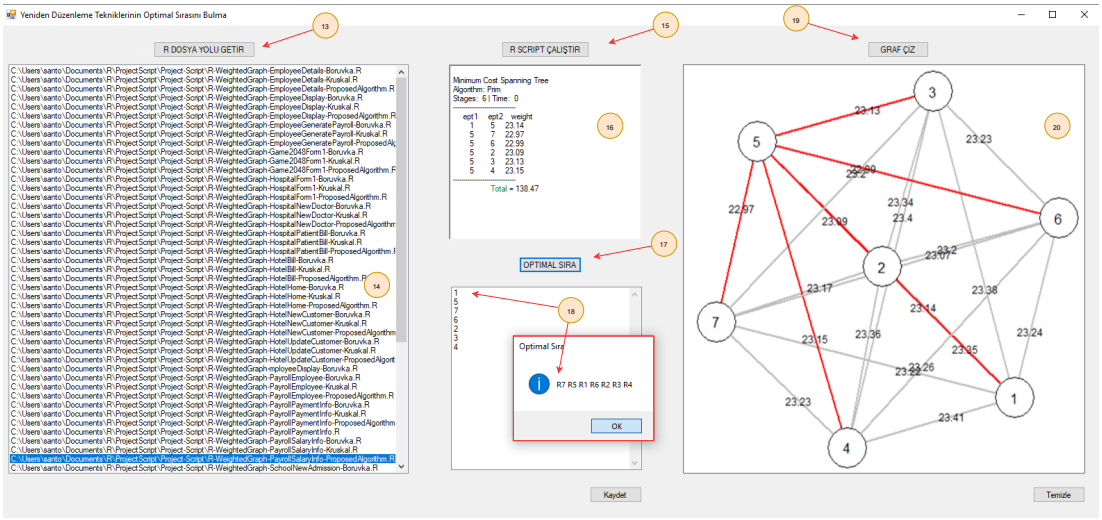
-	R7	R2	R3	R4	R5	R6	R1
R7	-	23.35	23.38	23.41	23.14	23.24	23.22
R2	23.35	-	23.34	23.36	23.09	23.20	23.17
R3	23.38	23.34	-	23.40	23.13	23.23	23.20
R4	23.41	23.36	23.40	-	23.15	23.26	23.23
R5	23.14	23.09	23.13	23.15	-	22.99	22.97
R6	23.24	23.20	23.23	23.26	22.99	-	23.07
R1	23.22	23.17	23.20	23.23	22.97	23.07	-

düzenleme teknikleri listesi o sınıfın log kayıtlarından 11 numaralı richtextbox kontrolüne aktarılmaktadır. 10 numaralı Uygulanan Yeniden Düzenleme Teknikleri buton nesnesi yardımıyla uygulanan yeniden düzenleme teknikleri 12 numaralı listBox kontrolüne aktarılmakta ve aynı zamanda mesaj olarak kullanıcı bilgilendirilmektedir. Uygulanan yeniden düzenleme teknikleri kullanıcıya aktarılan formda temsili kısa isimlendirme şeklinde gösterilmektedir. Graf düğüm ve kenar bilgilerinden elde



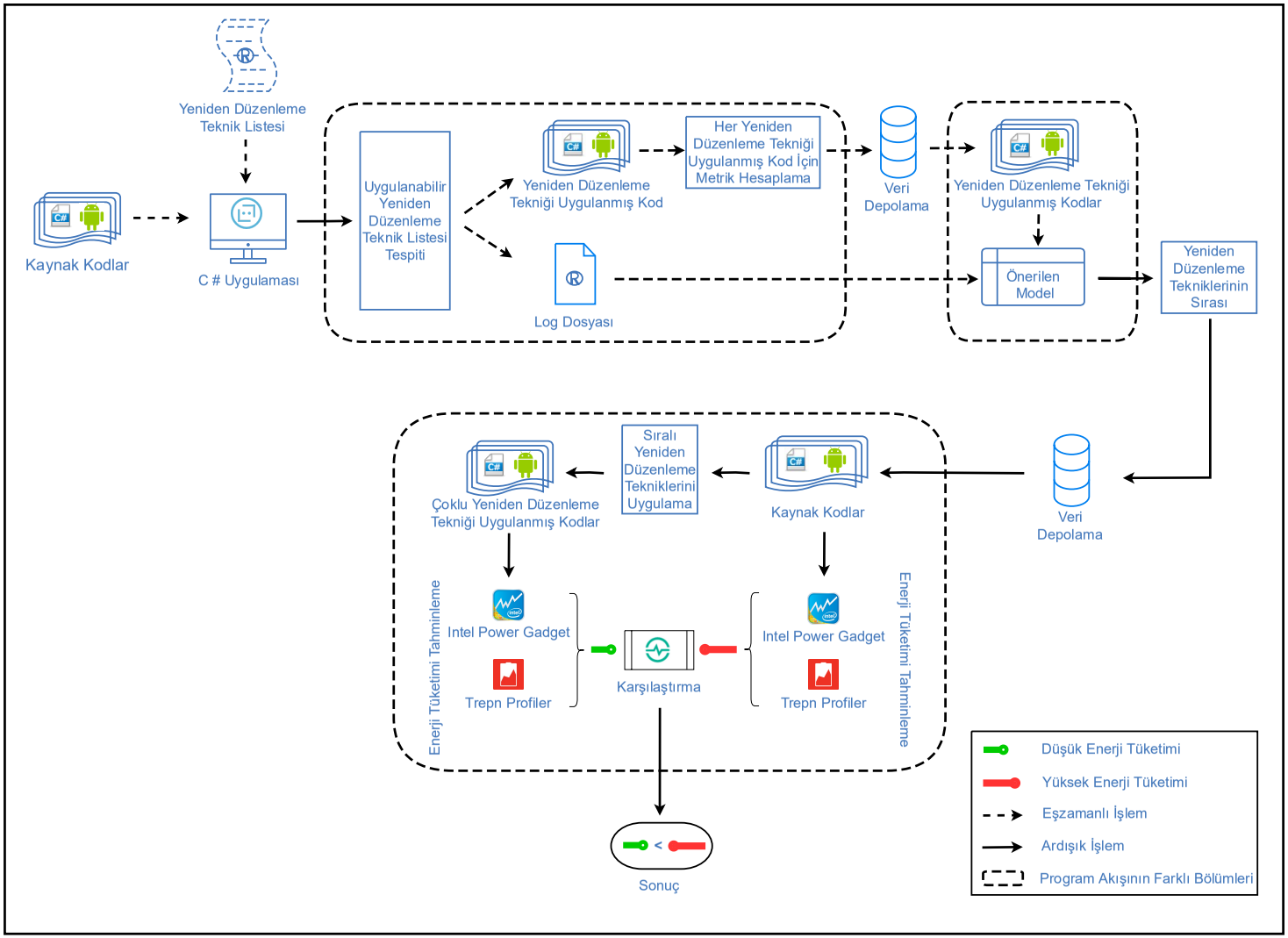
Şekil 4.6. Uygulanan yeniden düzenleme tekniklerinin tespiti

edilen komşuluk matris değerleri kaydedilen R GUI script dosyalarında tutulmaktadır. R GUI kullanılarak uygun yeniden düzenleme teknikleri sırasını elde etmek için geliştirilen yazılımın ekran görüntüsü Şekil 4.7'de verilmektedir. İlk olarak R GUI Script dosyalarının yolları, kullanıcı arayüzünde 13 numaralı olarak verilen Open R Path



Şekil 4.7. Yeniden düzenleme tekniklerinin optimal sırası

isimli buton yardımıyla 14 numaralı listView kontrolüne yüklenmektedir. Kullanıcının seçtiği yoldaki script dosyası 15 numaralı R Script Çalıştır isimli buton nesne yardımı ile Prim tabalı algoritmada yürütülmekte ve elde edilen sonuçlar 16 numaralı richTextBox kontrolü içerisine yüklenmektedir. Ardından, 17 numaralı Optimal Sıra isimli buton kontrolü yardımıyla en uygun yeniden düzenleme sırası elde edilmektedir. Elde edilen sıra 18 numaralı listbox kontrolüne aktarılmakta aynı zamanda mesaj olarak kullanıcıyı bilgilendirmektedir. Bu sıranın elde edilmesinde kullanılan grafın grafiksel gösterimi ise 19 numaralı Graf Çiz isimli buton kontrolü vasıtasıyla 20 numaralı Graf isimli pictureBox kontrolüne yüklenir. Geliştirilen otomasyon yazılım aracının replikasyon paketi https://github.com/isanlialp/ReplicationPackage linkinde mevcuttur.



Şekil 4.8. Çalışmanın kavramsal çerçevesi.

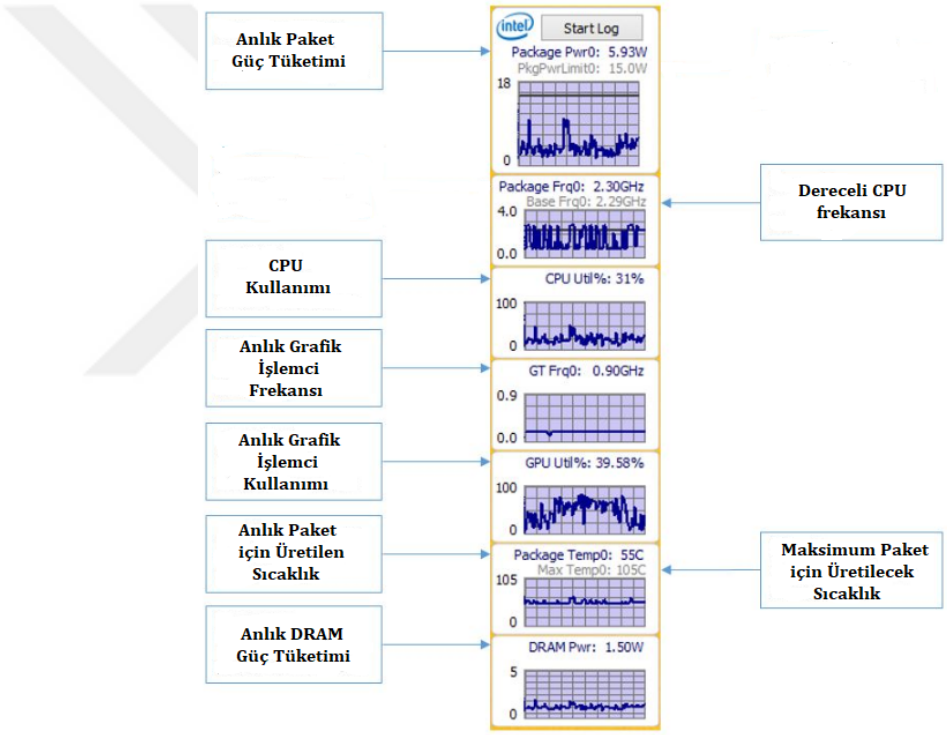
4.7. Enerji Tüketiminin İzlenmesi

Bilgisayar programları, kullanıldığında çok sayıda kaynağa ihtiyaç duyar. Sabit disk, grafik kartı, depolama aygıtı, bellek ve CPU ana kaynaklar olarak kabul edilebilir. Bilgisayar programlarının her bir işleminin yürütülmesi sırasında, işlem başına disk okuma / yazma ya da CPU hesaplaması gibi bazı işlemler enerji tüketir. Bir kullanıcı işlemcinin enerji kullanımını tahmin etmek isterse, enerji kullanımını izlemek için özel araçlar vardır:

- Jouletrack
- XEEMU
- Jalen
- Jolinar
- pTop
- EnergyChecker
- PowerApi
- PHOENIX
- Petra
- Powerscope
- GreenAdvisor
- Monsoon
- Intel RAPL
- Joulemeter
- Trepn Profiler
- Intel Power Gadget

Bazı özel araçlar, kullanıcının herhangi bir donanım enstrümantasyonu gerektirmeden enerji tüketimini tahmin etmesini sağlar. Geliştiricilerin belirli işlemleri

gerçekleştirirken uygulamaların enerji tüketimi davranışından haberdar olmaları çok önemlidir. Bu tür özel araçlar olmadan, geliştiricilerin program kodunu enerji verimliliği için optimize etmeleri zordur (Hoque vd., 2015). Çalışma kapsamında, C# ve Java programlama dilinde yazılmış kaynak kodlar için enerji tüketimi tahmin edilmektedir. Diğer nesne yönelimli programlama dilleri bu çalışmanın kapsamında değildir. Sekiz açık kaynak kodlu proje ölçüme tabi tutulmuştur. Enerji tüketimini tahmin etmek için, nesneye yönelik kod tabanlı tahmini destekleyen Intel Power Gadget (Intel, 2020) ve Trepn Profiler (Qualcomm, 2018) yazılım güç tahminleme araçları kullanılmaktadır. Şekil 4.9'da Intel Power Gadget aracını oluşturan bileşenler ve özellikleri verilmektedir. Intel Power Gadget aracı, tahminleme sonucunda bir log dosyası üretir. Araç paket güç

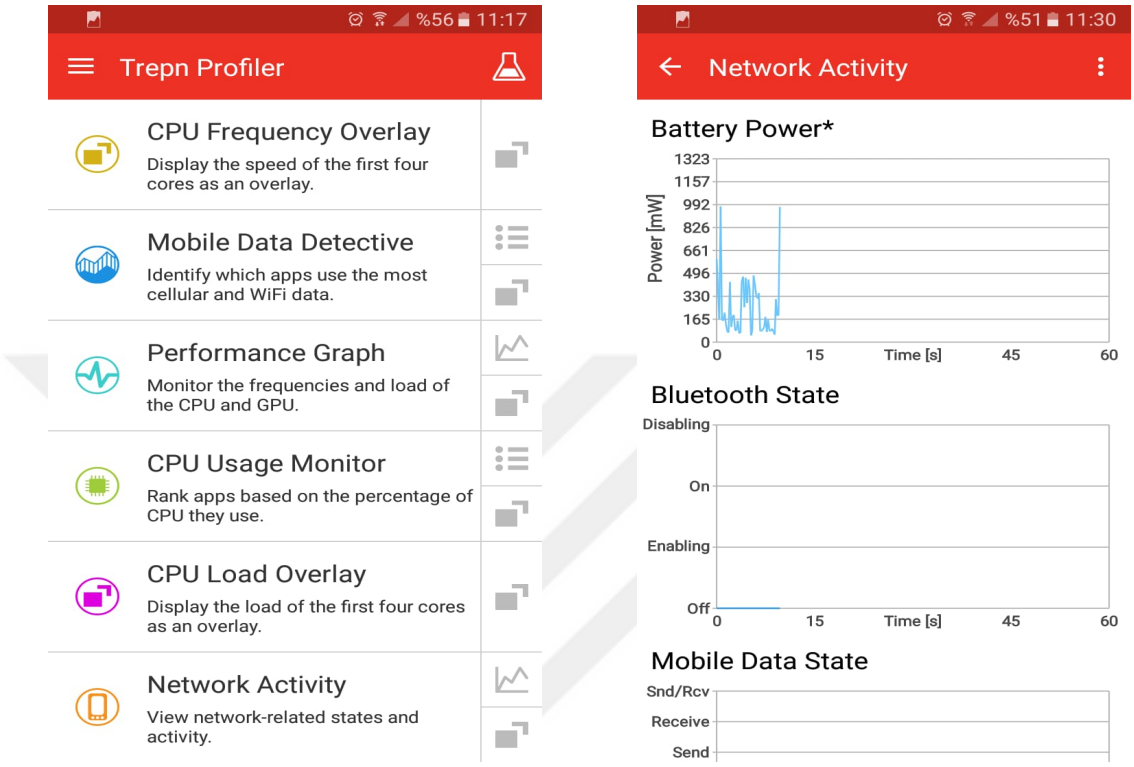


Şekil 4.9. Intel Power Gadget bileşenleri

limiti, geçen süre (elapsed time), işlemci frekansı, grafik teknolojileri (GT) frekansı, işlemci sıcaklığı, ortalama ve işlemcinin kümülatif gücünü içerir. GT, Intel tarafından merkezi işlem birimiyle aynı pakette veya kalıpta üretilen bir dizi tümleşik grafik işlemcisinin toplu adıdır. İşlemci enerji tüketimini tahmin etmek için, araç Denklem 4.2 (Intel, 2020)'de sunulan formülü kullanılır:

$$T_{et} = IA + GT\,(varsa) + D_{et} \tag{4.2}$$

T_{et} işlemcinin toplam enerji tüketimi, IA enerji işlemci çekirdekleri enerjisi olarak ölçülmektedir. GT enerji ise grafik işlemci birimi enerjisi olarak ölçülmektedir fakat bu değer masaüstü bilgisayarlarda ve sunucular için bazı işlemcilerde yoktur. D_{et} bilgisayarın diğer parçalarının enerji tüketimini temsil eder ve araç bunu ölçmez. Şekil



(a) (b)

Şekil 4.10. Trepn Profiler bileşenleri

4.10'da Trepn Profiler aracını oluşturan bileşenler ve özellikleri verilmektedir. Trepn Profiler, Qualcomm topluluğu tarafından geliştirilmiştir ve Snapdragon yonga seti tabanlı Android cihazlara sahip cihazlarda çalışır (Qualcomm, 2018). Uygulama CPU, GPU, Wi-Fi, uyandırma kilitleri, bellek, dijital güvenlikli (SD) kart güç tüketimini ve tüm cihazın çalışma zamanı enerji tüketimini ölçebilir (Hoque vd., 2015). Güç okumalarını, algılama dirençlerinden okumaları toplayan ve bireysel donanım bileşenleri için akıma dönüştüren güç yönetimi entegre devresinden (PMIC) ve bataryadan güç dağıtımını kontrol eden entegre güç yönetimi IC'ye sahip özel pil yakıt göstergesi çipi yazılımından alır (Qualcomm, 2018). Trepn her 100 ms'den sonra bilgileri örnekler ve farklı bilgi görselleştirme modları ile sunar. Trepn Profiler ayrıca ön planda farklı grafik ve çizelgelerin bir görünümünü sağlar, böylece uygulama geliştiriciler uygulamaların performansını çalışma zamanında CPU kullanımı ve enerji

tüketimi ile ilişkilendirebilir. Çevrimdışı analiz için gerçek zamanlı ham veri aktarımına izin verir (Hoque vd., 2015). Profil oluşturma verilerini daha sonra analiz etmek üzere kaydetmek için gelişmiş mod seçilir. Profil oluşturma aralığı 100 milisaniye olarak ayarlanır ve güç istatistikleri için veri noktaları sekmesinden pil gücü seçilir. Uygulamanın profil oluşturma işlemi daha sonra başlatılır. Belirli bir süre sonra profil oluşturma oturumu sonlandırılır ve sonuçlar bir csv dosyası olarak kaydedilir. Güç istatistikleri mikrowatt cinsinden ölçülür ve ölçüm için profil oluşturma süresi milisaniye cinsindendir. Enerji birimi, Trepn Profiler'dan elde edilen değerler Denklem 4.3 (Chan-Jong-Chu vd., 2020) hesaplanarak dönüştürülür:

$$Enerji(J) = (\frac{P}{10^6})W \times (\frac{T}{10^3})s \tag{4.3}$$

Denklem 4.3'te J joule, P güç, W watt, s saniye ve T profil süresi olarak temsil edilmektedir. W gücü ölçmek için kullanılırken, J işi gerçekleştirmek için gereken kümülatif watt-saniyeyi ölçmek için kullanılır. Testlerin tümü aynı uzunlukta olduğunda, ortalama watt (ortalama güç) yaygın olarak (Hindle, 2015; Aggarwal vd., 2014) kullanılır. Çalışmada ortalama watt kullanılmıştır. Bir test sırasında joule cinsinden tüketilen enerji, ortalama wattların saniyelerle çarpılmasıyla hesaplanır.

Çalışma kapsamında projelerin enerji tüketim sonuçlarının elde edilmesi için hem dizüstü bilgisayar hem de cep telefonu kullanılmaktadır. Intel Power Gadget, dizüstü bilgisayarın değerlerini kaydetmek için kullanılır. Trepn Profiler, Android tabanlı Java projelerinin cep telefonunda enerji ayak izini kaydetmek için seçilmiştir. Kullanılan dizüstü bilgisayarın özellikleri: Intel Core i7-8750H, temel frekans işlemci 2.20 Ghz, 6 çekirdek, önbellek 9MB, 8 GB bellek ve 256 GB disk. Kullanılan cep telefonu özellikleri: 8 çekirdekli 2.0 GHz Cortex-A53 işlemcili, 4 GB ram ve 32 GB disk içeren Samsung Galaxy C7. Dizüstü bilgisayar için ölçüm hiçbir uygulama yürütülmediğinde (boş durumunda) ve uçak modunda yapılır. Wi-Fi ve Bluetooth gibi harici iletişim sağlayan bazı cihazlar çevrimdışı olarak ayarlanmış şekilde bataryadan güç tüketimi esnasında ölçümler gerçekleştirilir. Dizüstü bilgisayarda deneyler, Windows 10 Pro ile çevrimdışı olarak gerçekleştirilir. Tüm deneyler çevrimdışı yapıldığından ağ gücü hesaba katılmaz. Ayrıca tüm deneyler aynı zaman aralığında (5 dakika) (Palomba vd., 2019) programın en yoğun kısmını test etmek için gereken süre dikkate alınarak

gerçekleştirilir ve kontrolsüz davranışın etkisini azaltmak için on kez tekrarlanır. Ayrıca, eş zamanlı yürütmede deneyi gerçekleştirmek için gereken minimum sayıda uygulamaya (işletim sistemi çekirdek hizmetleri, enerji ölçüm aracı, ölçülen uygulama vb.) izin verilir. Güvenilir bir değerlendirme yapabilmek için aynı konfigürasyon cep telefonunda da uygulanır. Cep telefonu için deney, tüm profil oluşturma süresi boyunca uyanık kilit ile mümkün olan maksimum periyot ile gerçekleştirilir. Mobil işlemcinizin uyanık kalması doğru ortalama güç okumaları sağlamaktadır. Uygulamanın güç profilleri csv dosyaları olarak kaydedilir ve bunlar daha sonra tüketilen enerjiyi hesaplamak için çevrimdışı olarak işlenir. Cep telefonu ile ilgili çalışmalar manuel olarak yapılmaktadır. Ayrıca, programları yürütmek için bazı girdi verilerinin gerekli olması halinde, orijinal kod ve yeniden düzenlenmiş kod için aynı girdi verileri kullanılmıştır.

5. ARAŞTIRMA BULGULARI

Projelere ait toplam 116 adet kaynak dosya incelenmiştir. Enerji tüketimi ve kodun sürdürülebilirliği açısından önemli görülen ve bu doğrultuda kodlaması yapılan yeniden düzenleme teknikleri incelenen yazılım projelerindeki kodlara uygulanmaktadır. Geliştirilen otomasyon yazılım sayesinde elde edilen proje kodlarına yeniden düzenleme teknikleri bireysel olarak uygulanmaktadır. Sonrasında metrik ölçüm değerleri elde edilmekte ve bu değerler geliştirilen yöntem için giriş değerlerini oluşturmaktadır. Daha sonra hesaplamalardan elde edilen çıkış değerleri Prim tabanlı önceliklendirme algoritmasına giriş değerleri olarak verilmekte ve sonucunda üç veya daha fazla yeniden düzenleme tekniklerinin sırası elde edilmektedir.

Orijinal ve değiştirilen kaynak kodları için sekiz projenin kümülatif enerji tüketimi değişikliklerinin toplu halde kutu grafik gösterimleri sırasıyla Şekil 5.9 ve 5.10'da sunulmaktadır. Sonuçların kümülatif olarak ve J cinsinden verildiğini belirtmek gerekir. Boruvka ve Kruskal algoritmaları yürütülürken deney, Denklem 4.1'de verilen düğüm ağırlığı formülünden yararlanılacak şekilde tasarlanmıştır. Orijinal kodların enerji tüketim ortalamaları, Otel Yönetim Sistemi projesi haricinde değiştirilen kodlardan daha yüksektir. Bu sonuç, türüne bakılmaksızın yeniden düzenleme işleminin enerji tüketimi açısından proje kaynak kodlarını önemli ölçüde iyileştirdiğini göstermektedir. Önerilen algoritma, Hastane Yönetim Sistemi projesi ve Bordro Yönetim Sistemi projesi dışında en düşük enerji tüketim ortalama değerini elde etmiştir. Bu, yazılım kalitesinin belirli bir yönünü iyileştirmek için grafik tabanlı bir algoritmanın gerekli olması durumunda, Prim tabanlı algoritmaların yazılım enerji verimliliği açısından daha fazla tercih edilebilir olduğu varsayımını destekler.

Projelere ait orijinal kodun enerji tüketimi sonuçlarının dağılımını üç karşılaştırma algoritması ile değerlendirmek için veri dağılımı konusunda önyargı oluşturmayan Wilcoxon işaretli sıra testi (Woolson, 2007) seçilmiştir. Test önemli düzey olan 0.05 düzeyinde çalıştırılmıştır. H_0 hipotezi, orijinal kodun enerji tüketim sonuçları dağılımının karşılaştırma algoritmaları ile aynı olduğunu belirtir. H_1 hipotezi ise karşılaştırma algoritmaları ile arasında önemli bir farkın olduğu anlamına gelir. Çizelge 5.1 ve 5.2'de verilen test sonuçlarına göre, üç karşılaştırma algoritması için dağılımın

farklı olduğu görülmektedir. Bu nedenle, test hedefleri arasındaki önemli farkın tam

Çizelge 5.1. Orijinal kod ile karşılaştırma algoritmalarının Wilcoxon işaretli sıra testi sonuçları ($H_0 : p > 0.05$, $H_1 : p < 0.05$)

	Önerilen algoritma	Kruskal	Boruvka
Orijinal Kod [a] (p değeri)	0.012	0.048	0.010

[a] *Projelere ait orijinal kodların enerji tüketim sonuçları dağılımı*

tersi olan H_0'ın reddedilmesidir. Karşılaştırma algoritmalarından elde edilen sıralar Çizelge 5.3, 5.4, 5.5'te verilmektedir. "X" etiketi kaynak kod üzerinde tespit edilen yeniden düzenleme tekniğinin uygunluğunu belirtirken, "-" etiketine sahip yeniden düzenleme teknikleri elenmiştir. Uygulanabilen yeniden düzenleme teknikleri, enerji tüketim sonuçlarını farklılaştıran sıralar oluşturmak için özellikle uygundur.

Çizelge 5.2. Enerji tüketim sonuçları. Tüm deneyler aynı zaman aralığında (300 saniye) ve on kez çalıştırma ile gerçekleştirilir

	Aritmetik Ortalamalar (Enerji Tüketim (J))			
Ad	Orijinal Kod Değerleri	Önerilen Algoritma Değerleri	Kruskal Algoritma Değerleri	Boruvka Algoritma Değerleri
Basit Hesap Makinesi Projesi	1365.23	1346.79	1351.04	1357.37
Oyun 2048 Projesi	1466.45	1399.83	1407.48	1446.65
Bordro Yönetim Sistemi Projesi	2003.20	1830.01	1644.26	1782.18
Otel Yönetim Sistemi Projesi	1455.45	1425.30	1494.04	1450.55
Hastane Yönetim Sistemi Projesi	1868.24	1798.90	1711.59	1680.50
Çalışan Yönetim Sistemi Projesi	1978.60	1724.02	1754.58	1789.80
Mobil-Oyun 2048 Projesi	117.73	98.11	108.08	99.53
Mobil-Hesap Makinesi Projesi	191.20	171.89	182.76	188.75

Çizelge 5.3. Kruskal algoritması ile üretilen optimal sıraların detayları

Ad	R_1	R_2	R_3	R_4	R_5	R_6	R_7	Sıra
Basit Hesap Makinesi Projesi	-	-	X	X	X	X	X	R_3, R_6, R_7, R_4, R_5
Oyun 2048 Projesi	X	X	-	X	X	X	X	$R_2, R_6, R_5, R_4, R_7, R_1$
Bordro Yönetim Sistemi Projesi	X	-	X	-	X	X	X	R_3, R_6, R_5, R_7, R_1
Otel Yönetim Sistemi Projesi	-	-	X	X	X	X	-	R_4, R_6, R_5, R_3
Hastane Yönetim Sistemi Projesi	X	X	X	X	X	X	X	$R_2, R_6, R_3, R_7, R_4, R_5, R_1$
Çalışan Yönetim Sistemi Projesi	-	-	X	X	-	X	X	R_3, R_6, R_7, R_4
Mobil-Oyun 2048 Projesi	-	X	-	X	X	-	X	R_4, R_5, R_7, R_2
Mobil-Hesap Makinesi Projesi	-	X	X	-	X	-	X	R_3, R_5, R_2, R_7

Her proje için en iyi tahminleme sonuçlarına sahip kaynak dosyanın optimal sırasının elde edilmesinde kullanılan bulgular sırayla anlatılmaktadır. Bulguların elde edilmesinde kullanılan projelere ait enerji ölçüm sonuçları

Çizelge 5.4. Boruvka algoritması ile üretilen optimal sıraların detayları

Ad	R_1	R_2	R_3	R_4	R_5	R_6	R_7	*Sıra*
Basit Hesap Makinesi Projesi	-	-	X	X	X	X	X	R_3, R_6, R_4, R_5, R_7
Oyun 2048 Projesi	X	X	-	X	X	X	X	$R_1, R_6, R_2, R_4, R_5, R_7$
Bordro Yönetim Sistemi Projesi	X	-	X	-	X	X	X	R_1, R_6, R_3, R_5, R_7
Otel Yönetim Sistemi Projesi	-	-	X	X	X	X	-	R_3, R_6, R_4, R_5
Hastane Yönetim Sistemi Projesi	X	X	X	X	X	X	X	$R_1, R_6, R_2, R_3, R_4, R_5, R_7$
Çalışan Yönetim Sistemi Projesi	-	-	X	X	-	X	X	R_3, R_6, R_4, R_7
Mobil-Oyun 2048 Projesi	-	X	-	X	X	-	X	R_2, R_4, R_5, R_7
Mobil-Hesap Makinesi Projesi	-	X	X	-	X	-	X	R_2, R_5, R_3, R_7

Çizelge 5.5. Önerilen algoritma ile üretilen optimal sıraların detayları

Ad	R_1	R_2	R_3	R_4	R_5	R_6	R_7	*Sıra*
Basit Hesap Makinesi Projesi	-	-	X	X	X	X	X	R_4, R_6, R_3, R_7, R_5
Oyun 2048 Projesi	X	X	-	X	X	X	X	$R_6, R_2, R_5, R_4, R_7, R_1$
Bordro Yönetim Sistemi Projesi	X	-	X	-	X	X	X	R_7, R_6, R_3, R_5, R_1
Otel Yönetim Sistemi Projesi	-	-	X	X	X	X	-	R_5, R_6, R_4, R_3
Hastane Yönetim Sistemi Projesi	X	X	X	X	X	X	X	$R_2, R_6, R_7, R_3, R_4, R_5, R_1$
Çalışan Yönetim Sistemi Projesi	-	-	X	X	-	X	X	R_7, R_3, R_6, R_4
Mobil-Oyun 2048 Projesi	-	X	-	X	X	-	X	R_5, R_4, R_7, R_2
Mobil-Hesap Makinesi Projesi	-	X	X	-	X	-	X	R_5, R_3, R_2, R_7

https://github.com/isanlialp/ThesisResults linkinde mevcuttur. Ayrıca projelerin yazılım sürdürülebilirliğinin değerlendirilmesinde geliştirilen matematiksel modeldeki kompleks metrik hesabından elde edilen bulgular sürdürülebilirlik değeri olarak analiz edilir.

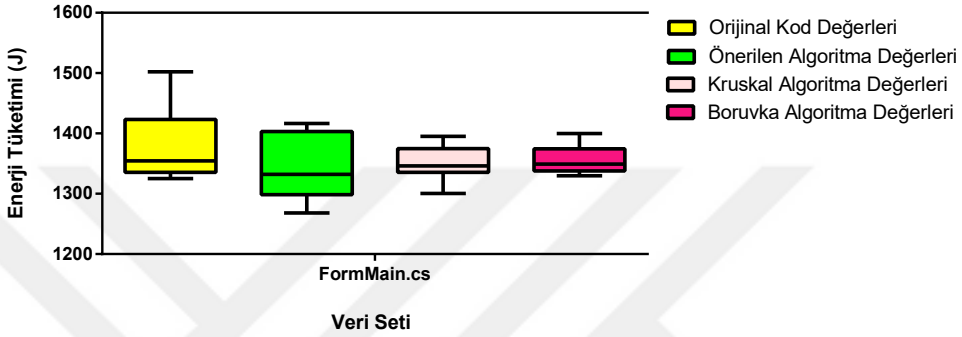
Basit Hesap Makinesi projesinin FormMain sınıfına ait orijinal kodlar üzerinde işlem gören kaynak dosyası ve dosyalara uygulanan yeniden düzenleme teknikleri Çizelge 5.6'da verilmektedir.

Çizelge 5.6. Basit hesap makinesi projesinde işlem yapılan kaynak dosya ve uygulanan yeniden düzenleme teknikleri

Kaynak dosya ismi	R_1	R_2	R_3	R_4	R_5	R_6	R_7
FormMain.cs	-	-	X	X	X	X	X

Basit Hesap Makinesi projesinde işlem yapılan FormMain dosyasına uygulanan yeniden düzenleme tekniklerinin metrik ölçüm sonuçları yardımıyla enerji tüketimine olumlu katkı yapabilecek en uygun yeniden düzenleme tekniklerinin kombinasyonunun sırası tespit edilebilmektedir. FormMain sınıfına ait nesne yönelimli kodlar üzerinde bireysel

olarak uygulanan yeniden düzenleme tekniklerinden ve önerilen algoritma ile üretilen yeniden düzenleme tekniklerine ait optimal sıranın uygulanmasından elde edilen bulguların detayları Çizelge 5.7'de verilmektedir. Önerilen algoritmadan elde edilen sırayı $R_{sıra}$ temsil etmektedir. Masaüstü uygulaması olan Basit Hesap Makinesi projesine ait orijinal ve yeniden düzenlenen kodların enerji tüketim sonuçları Şekil 5.1'de verilmektedir. Diğer projeler için işlem yapılan kaynak dosyası, kaynak dosyadan elde edilen enerji tüketim sonuçları ve metrik ölçüm sonuçları sırasıyla verilmektedir.



Şekil 5.1. Basit hesap makinesi projesi için orijinal kodun ve karşılaştırma algoritmalarının kümülatif enerji tüketim sonuçları

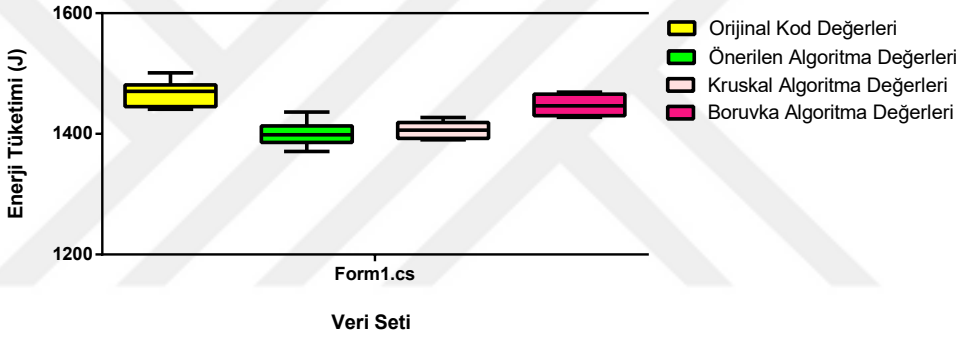
Çizelge 5.7. FormMain sınıfının metrik ölçüm sonuçları ve matematiksel model hesabı.

Yeniden Düzenleme Teknikleri	Metrikler										Matematiksel Model Hesabı
	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}	
Orijinal	89	49	10	78	24	58	1	4.83	5	8	2.41
R_3	88	45	10	75	24	59	1	4.67	5	8	2.36
R_4	89	49	10	87	24	58	1	5.33	5	8	2.41
R_5	89	49	10	81	24	58	1	5.17	5	8	2.41
R_6	88	47	10	78	24	58	1	5	4	8	2.26
R_7	88	49	10	78	24	58	1	5.17	5	8	2.40
$R_{sıra}$	86	47	10	75	24	59	1	3.88	5	8	2.32

Oyun 2048 projesinin Form1 sınıfına ait orijinal kodlar üzerinde işlem gören kaynak dosyası ve dosyaya uygulanan yeniden düzenleme teknikleri Çizelge 5.8'de verilmektedir. Projeye ait orijinal ve yeniden düzenlenen kodların enerji tüketim sonuçları Şekil 5.2'de verilmektedir. Form1 sınıfına ait nesne yönelimli kodlar üzerinde yeniden düzenleme tekniklerinin uygulanmasıyla elde edilen bulguların detayları ise Çizelge 5.9'da verilmektedir.

Çizelge 5.8. Oyun 2048 projesinde işlem yapılan kaynak dosya ve uygulanan yeniden düzenleme teknikleri

Kaynak dosya ismi	R_1	R_2	R_3	R_4	R_5	R_6	R_7
Form1.cs	X	X	-	X	X	X	X



Şekil 5.2. Oyun 2048 projesi için orijinal kodun ve karşılaştırma algoritmalarının kümülatif enerji tüketim sonuçları

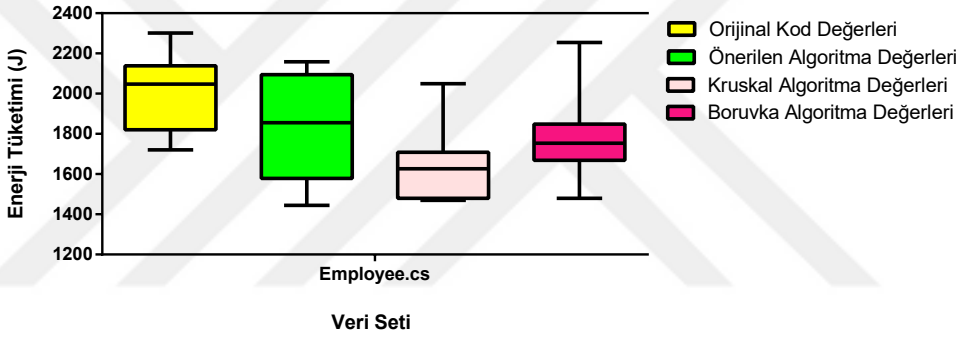
Çizelge 5.9. Form1 sınıfının metrik ölçüm sonuçları ve matematiksel model hesabı.

Yeniden Düzenleme Teknikleri	Metrikler										Matematiksel Model Hesabı
	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}	
Orijinal	878	428	136	1839	39	49	1	12.96	42	26	43.00
R_1	885	433	137	1842	39	49	1	12.11	43	27	44.57
R_2	868	426	134	1830	39	49	1	12.88	42	26	42.76
R_4	875	429	136	1842	39	49	1	13	42	26	42.94
R_5	874	428	136	1849	39	49	1	12.59	42	26	42.92
R_6	868	418	136	1809	39	49	1	12.58	41	26	42.27
R_7	878	428	136	1839	39	49	1	12.96	42	26	43.00
$R_{sıra}$	870	428	132	1833	39	50	1	12.01	42	27	42.74

Bordro Yönetim Sistemi projesinin Employee sınıfına ait orijinal kodlar üzerinde işlem gören kaynak dosyası ve dosyaya uygulanan yeniden düzenleme teknikleri Çizelge 5.10'da verilmektedir. Projeye ait orijinal ve yeniden düzenlenen kodların enerji tüketim sonuçları Şekil 5.3'te verilmektedir. Employee sınıfına ait nesne yönelimli kodlar üzerinde yeniden düzenleme tekniklerinin uygulanmasıyla elde edilen bulguların detayları ise Çizelge 5.11'de verilmektedir.

Çizelge 5.10. Bordro yönetim sistemi projesinde işlem yapılan kaynak dosya ve uygulanan yeniden düzenleme teknikleri

Kaynak dosya ismi	R_1	R_2	R_3	R_4	R_5	R_6	R_7
Employee.cs	X	-	X	-	X	X	X



Şekil 5.3. Bordro yönetim sistemi projesi için orijinal kodun ve karşılaştırma algoritmalarının kümülatif enerji tüketim sonuçları

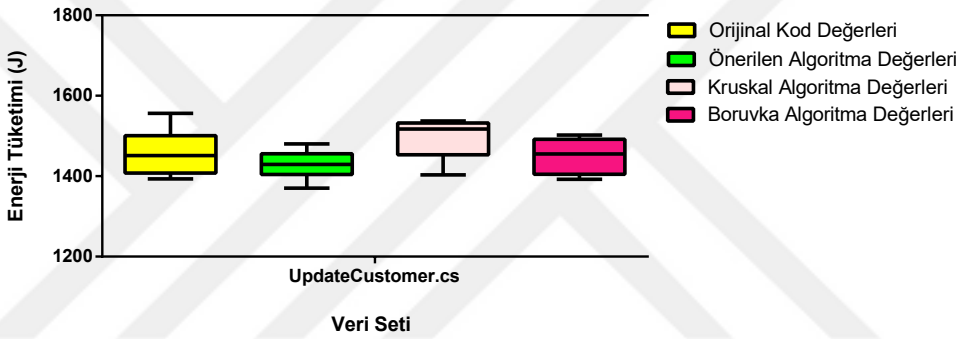
Çizelge 5.11. Employee sınıfının metrik ölçüm sonuçları ve matematiksel model hesabı.

Yeniden Düzenleme Teknikleri	Metrikler										Matematiksel Model Hesabı
	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}	
Orijinal	265	161	10	675	44	42	1	17.38	40	8	14.19
R_1	268	165	11	678	44	43	1	14.1	40	9	14.88
R_3	262	160	10	672	44	42	1	17.25	40	8	14.12
R_5	264	162	10	678	44	42	1	17.38	40	8	14.17
R_6	262	158	10	675	44	42	1	17.25	38	8	13.74
R_7	265	161	10	675	44	42	1	17.38	40	8	14.19
$R_{sıra}$	263	158	10	669	44	44	1	13.9	38	9	14.00

Otel Yönetim Sistemi projesinin UpdateCustomer sınıfına ait orijinal kodlar üzerinde işlem gören kaynak dosyası ve dosyaya uygulanan yeniden düzenleme teknikleri Çizelge 5.12'de verilmektedir. Projeye ait orijinal ve yeniden düzenlenen kodların enerji tüketim sonuçları Şekil 5.4'te verilmektedir. UpdateCustomer sınıfına ait nesne yönelimli kodlar üzerinde yeniden düzenleme tekniklerinin uygulanmasıyla elde edilen bulguların detayları ise Çizelge 5.13'te verilmektedir.

Çizelge 5.12. Otel yönetim sistemi projesinde işlem yapılan kaynak dosya ve uygulanan yeniden düzenleme teknikleri

Kaynak dosya ismi	R_1	R_2	R_3	R_4	R_5	R_6	R_7
UpdateCustomer.cs	-	-	X	X	X	X	-



Şekil 5.4. Otel yönetim sistemi projesi için orijinal kodun ve karşılaştırma algoritmalarının kümülatif enerji tüketim sonuçları

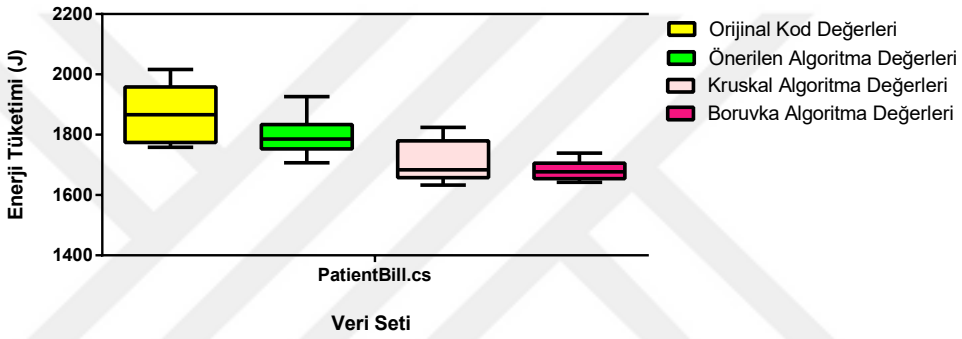
Çizelge 5.13. UpdateCustomer sınıfının metrik ölçüm sonuçları ve matematiksel model hesabı.

Yeniden Düzenleme Teknikleri	Metrikler										Matematiksel Model Hesabı
	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}	
Orijinal	167	98	9	240	30	45	1	9.75	23	11	9.56
R_3	159	97	9	237	30	45	1	9.63	23	11	9.38
R_4	165	101	9	240	30	46	1	8.75	23	11	9.30
R_5	158	99	9	243	30	45	1	9.75	23	11	9.36
R_6	165	97	9	240	30	45	1	9.63	22	11	9.27
$R_{sıra}$	160	102	9	240	30	46	1	8.56	22	11	8.96

Hastane Yönetim Sistemi projesinin PatientBill sınıfına ait orijinal kodlar üzerinde işlem gören kaynak dosyası ve dosyaya uygulanan yeniden düzenleme teknikleri Çizelge 5.14'te verilmektedir. Projeye ait orijinal ve yeniden düzenlenen kodların enerji tüketim sonuçları Şekil 5.5'te verilmektedir. PatientBill sınıfına ait nesne yönelimli kodlar üzerinde yeniden düzenleme tekniklerinin uygulanmasıyla elde edilen bulguların detayları ise Çizelge 5.15'te verilmektedir.

Çizelge 5.14. Hastane yönetim sistemi projesinde işlem yapılan kaynak dosya ve uygulanan yeniden düzenleme teknikleri

Kaynak dosya ismi	R_1	R_2	R_3	R_4	R_5	R_6	R_7
PatientBill.cs	X	X	X	X	X	X	X



Şekil 5.5. Hastane yönetim sistemi projesi için orijinal kodun ve karşılaştırma algoritmalarının kümülatif enerji tüketim sonuçları

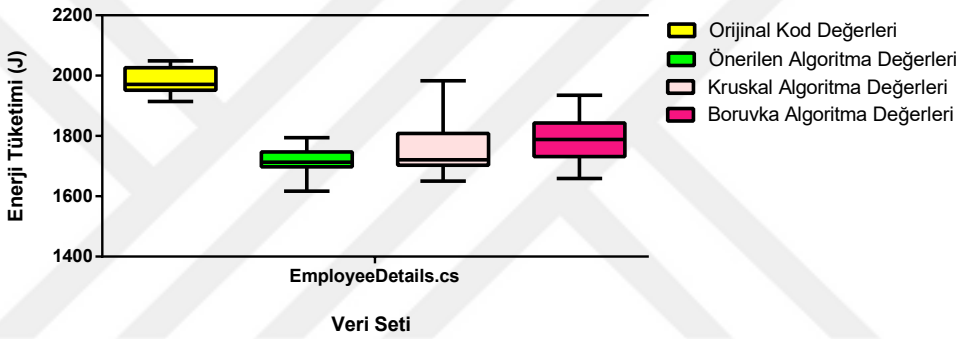
Çizelge 5.15. PatientBill sınıfının metrik ölçüm sonuçları ve matematiksel model hesabı.

Yeniden Düzenleme Teknikleri	Metrikler										Matematiksel Model Hesabı
	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}	
Orijinal	198	106	17	474	31	54	1	5.38	18	16	9.33
R_1	202	111	18	477	31	54	1	4.67	20	17	10.39
R_2	190	105	16	459	31	55	1	5.06	18	16	9.00
R_3	192	105	17	471	31	54	1	5.06	18	16	9.22
R_4	193	107	17	477	31	54	1	5.19	18	16	9.24
R_5	193	107	17	477	31	54	1	5.13	18	16	9.24
R_6	193	105	17	477	31	54	1	5.06	16	16	8.65
R_7	192	106	17	472	31	54	1	5.13	18	16	9.22
$R_{sıra}$	195	110	17	465	31	55	1	4.56	17	17	9.13

Çalışan Yönetim Sistemi projesinin EmployeeDetails sınıfına ait orijinal kodlar üzerinde işlem gören kaynak dosyası ve dosyaya uygulanan yeniden düzenleme teknikleri Çizelge 5.16'da verilmektedir. Projeye ait orijinal ve yeniden düzenlenen kodların enerji tüketim sonuçları Şekil 5.6'da verilmektedir. EmployeeDetails sınıfına ait nesne yönelimli kodlar üzerinde yeniden düzenleme tekniklerinin uygulanmasıyla elde edilen bulguların detayları ise Çizelge 5.17'de verilmektedir.

Çizelge 5.16. Çalışan yönetim sistemi projesinde işlem yapılan kaynak dosya ve uygulanan yeniden düzenleme teknikleri

Kaynak dosya ismi	R_1	R_2	R_3	R_4	R_5	R_6	R_7
EmployeeDetails.cs	-	-	X	X	-	X	X



Şekil 5.6. Çalışan yönetim sistemi projesi için orijinal kodun ve karşılaştırma algoritmalarının kümülatif enerji tüketim sonuçları

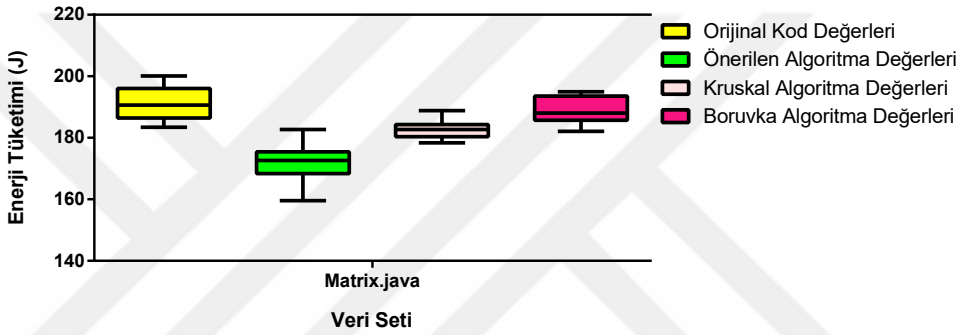
Çizelge 5.17. EmployeeDetails sınıfının metrik ölçüm sonuçları ve matematiksel model hesabı.

Yeniden Düzenleme Teknikleri	Metrikler										Matematiksel Model Hesabı
	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}	
Orijinal	323	126	15	588	55	49	7	8.65	26	17	16.06
R_3	322	125	15	585	54	49	7	8.59	26	17	16.04
R_4	324	127	15	591	55	49	7	8.71	26	17	16.08
R_6	322	125	15	582	55	49	7	8.59	26	17	16.04
R_7	323	126	15	588	55	49	7	8.65	26	17	16.06
$R_{sıra}$	310	125	15	582	54	49	7	8.59	26	17	15.80

Mobil-Oyun 2048 projesinin Matrix sınıfına ait orijinal kodlar üzerinde işlem gören kaynak dosyası ve dosyaya uygulanan yeniden düzenleme teknikleri Çizelge 5.18'de verilmektedir. Projeye ait orijinal ve yeniden düzenlenen kodların enerji tüketim sonuçları Şekil 5.7'de verilmektedir. Matrix sınıfına ait nesne yönelimli kodlar üzerinde yeniden düzenleme tekniklerinin uygulanmasıyla elde edilen bulguların detayları ise Çizelge 5.19'da verilmektedir.

Çizelge 5.18. Mobil-Oyun 2048 projesinde işlem yapılan kaynak dosya ve uygulanan yeniden düzenleme teknikleri

Kaynak dosya ismi	R_1	R_2	R_3	R_4	R_5	R_6	R_7
Matrix.java	-	X	-	X	X	-	X



Şekil 5.7. Mobil-Oyun 2048 projesi için orijinal kod ve karşılaştırma algoritmalarının kümülatif enerji tüketim sonuçları

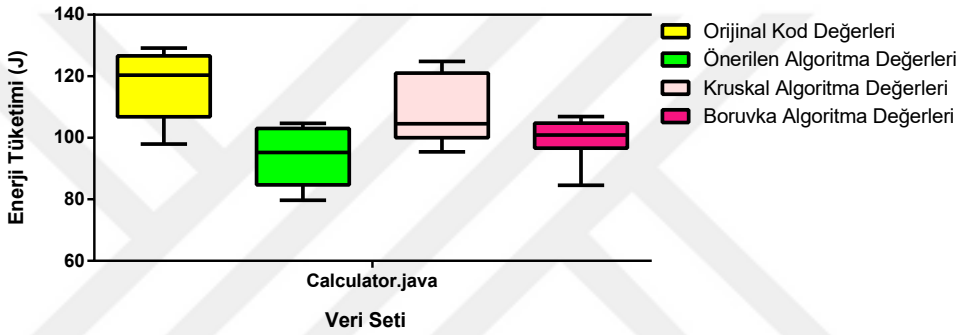
Çizelge 5.19. Matrix sınıfının metrik ölçüm sonuçları ve matematiksel model hesabı.

Yeniden Düzenleme Teknikleri	Metrikler										Matematiksel Model Hesabı
	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}	
Orijinal	399	247	88	978	11	144.67	1	9.22	35	23	8.94
R_2	395	245	88	969	11	144.03	1	8.98	35	23	8.95
R_4	400	242	87	975	11	145.56	1	9.02	35	23	8.88
R_5	398	247	88	978	11	144.67	1	9.09	35	23	8.93
R_7	399	247	88	978	11	144.67	1	9.09	35	23	8.94
$R_{sıra}$	395	240	87	969	11	147.23	1	9.22	35	23	8.75

Mobil-Hesap Makinesi projesinin Calculator sınıfına ait orijinal kodlar üzerinde işlem gören kaynak dosyası ve dosyaya uygulanan yeniden düzenleme teknikleri Çizelge 5.20'de verilmektedir. Projeye ait orijinal ve yeniden düzenlenen kodların enerji tüketim sonuçları Şekil 5.8'de verilmektedir. Calculator sınıfına ait nesne yönelimli kodlar üzerinde yeniden düzenleme tekniklerinin uygulanmasıyla elde edilen bulguların detayları ise Çizelge 5.21'de verilmektedir.

Çizelge 5.20. Mobil-Hesap makinesi projesinde işlem yapılan kaynak dosya ve uygulanan yeniden düzenleme teknikleri

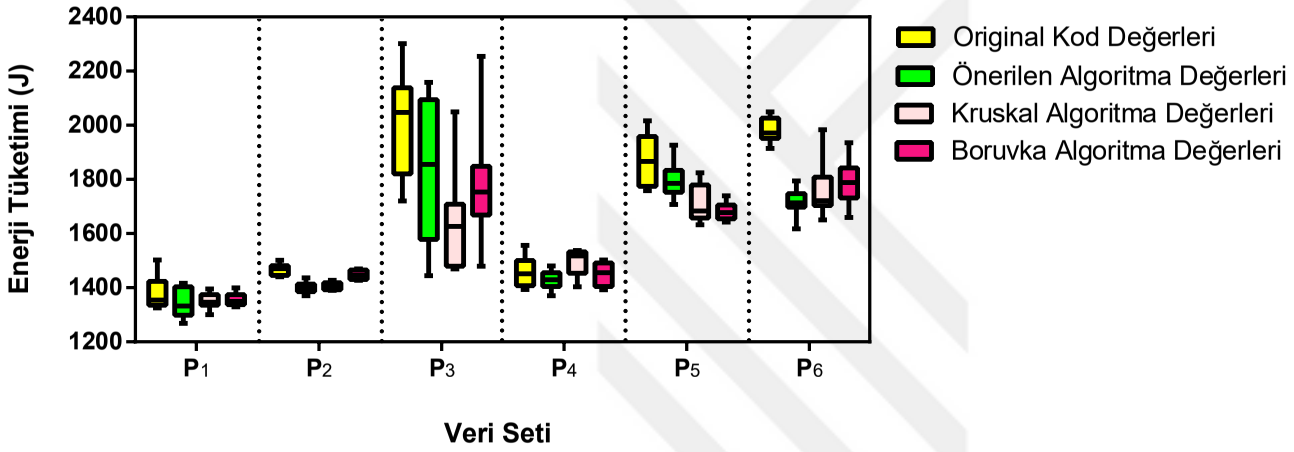
Kaynak dosya ismi	R_1	R_2	R_3	R_4	R_5	R_6	R_7
Calculator.java	-	X	X	-	X	-	X



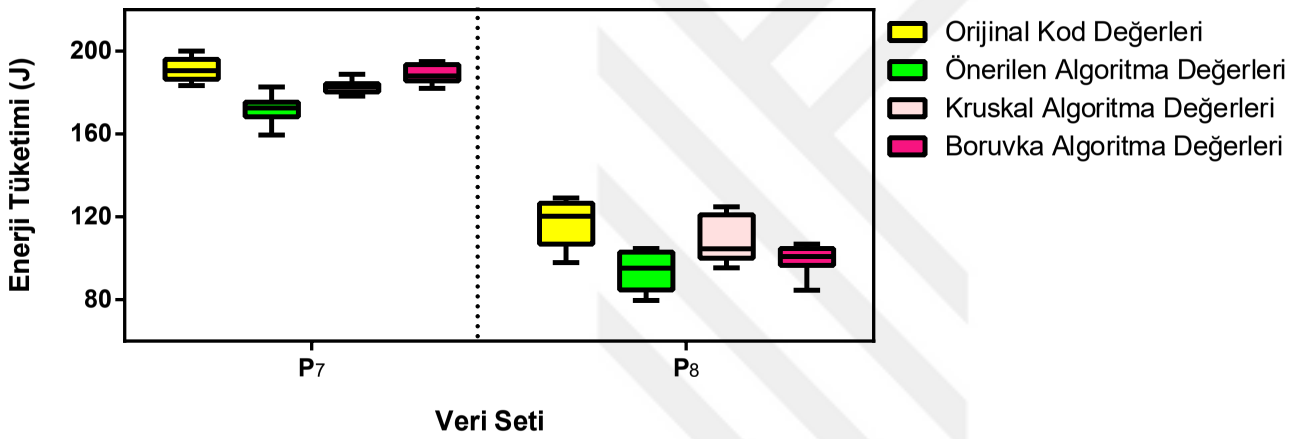
Şekil 5.8. Mobil-Hesap makinesi projesi için orijinal kodun ve karşılaştırma algoritmalarının kümülatif enerji tüketim sonuçları

Çizelge 5.21. Calculator sınıfının metrik ölçüm sonuçları ve matematiksel model hesabı.

Yeniden Düzenleme Teknikleri	Metrikler										Matematiksel Model Hesabı
	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}	
Orijinal	271	106	40	180	5	166.31	1	4.61	49	23	8.65
R_2	264	102	39	168	5	165.65	1	4.35	49	23	8.64
R_3	268	103	40	174	5	166.42	1	4.57	49	23	8.63
R_5	270	108	40	183	5	166.94	1	4.52	49	23	8.61
R_7	271	106	40	180	5	166.31	1	4.61	49	23	8.65
$R_{sıra}$	266	104	39	171	5	167.12	1	4.61	49	23	8.57



Şekil 5.9. Orijinal kodun ve karşılaştırma algoritmalarının kümülatif enerji tüketim sonuçları-I (toplu kutu grafik gösterimi): P_1, Basit hesap makinesi projesi; P_2, Oyun 2048 projesi; P_3, Bordro yönetim sistemi projesi; P_4, Otel yönetim sistemi projesi; P_5, Hastane yönetim sistemi projesi; P_6, Çalışan yönetim sistemi projesi



Şekil 5.10. Orijinal kodun ve karşılaştırma algoritmalarının kümülatif enerji tüketim sonuçları-II (toplu kutu grafik gösterimi): P_7, Mobil-Oyun 2048 projesi; P_8, Mobil-Hesap makinesi projesi

Önerilen algoritma Çizelge 5.2'de verilen ortalama değerlere göre projelerin toplam enerji tüketimini % 6.23 azaltabilmiştir. Basit Hesap Makinesi projesi için yöntemimizin alt çeyreği, orijinal kodun sonucundan daha düşük bir değer elde etmiştir. Bu projede, önerilen algoritma uygulandıktan sonra elde edilen optimum yeniden düzenleme sırası R_4, R_6, R_3, R_7, R_5, Çizelge 5.7'de gösterildiği gibi matematiksel model hesabına göre sürdürülebilirlik değerinde % 3.73 iyileşme sağlamıştır. Mobil-Hesap Makinesi projesinin enerji tüketim aralıkları Şekil 5.8'de gösterildiği gibi farklı büyüklük sırasına göre değişir. Bu proje için yöntemimizin alt çeyreği orijinal kodlara ve diğer karşılaştırma algoritmalarına göre daha düşük enerji tüketim değeri elde etmiştir. Bu projede, önerilen algoritma uygulandıktan sonra elde edilen optimum yeniden düzenleme sırası R_5, R_3, R_2, R_7, Çizelge 5.21'de gösterildiği gibi matematiksel model hesabına göre sürdürülebilirlik değerinde % 0.92 iyileşme sağlamıştır. Geliştirilen yöntemin küçük boyutlu uygulamalar olan hesap makineleri projeleri için veri hacimlerini azaltarak enerji tüketimine katkı sağladığı görülmüştür.

Şekil 5.2'de gösterildiği gibi Oyun 2048 projesi enerji tüketim aralıkları birkaç büyüklük sırasına göre değişir. Yöntemimiz diğer karşılaştırma algoritmaları ve orijinal kodlara göre enerji tüketim sonuçlarında üstünlük göstermektedir. Bu projede, önerilen algoritma uygulandıktan sonra elde edilen optimum yeniden düzenleme sırası $R_6, R_2, R_5, R_4, R_7, R_1$, Çizelge 5.9'da gösterildiği gibi matematiksel model hesabına göre sürdürülebilirlik değerinde % 0.6 iyileşme sağlamıştır. Diğer taraftan, Şekil 5.7'de verilen sonuçlara göre Mobil-Oyun 2048 projesinde de yöntemimiz diğer karşılaştırma algoritmalarından ve orijinal kodlardan daha düşük bir enerji tüketim değeri elde etmiştir. Önerilen algoritma uygulandıktan sonra elde edilen optimum yeniden düzenleme sırası R_5, R_4, R_7, R_2, Çizelge 5.19'da gösterildiği gibi matematiksel model hesabına göre sürdürülebilirlik değerinde % 2.12 iyileşme sağlamıştır.

Genel olarak veri aktarımı enerji tüketiminin ana kısmını oluşturur (Lyu vd., 2019; Sun vd., 2017). Bordro Yönetim Sistemi projesinin enerji tüketim değerleri tüm ölçümlerde geniş bir yelpazeye yayılmıştır. Çünkü proje yoğun bir veri aktarım işlemi içermektedir. Örneğin bu projede çok sayıda SQL sorgusu mevcuttur. Fakat tekniklerin farklı sıralarda uygulanması, bu proje için tüm algoritmaların sonuçlarında pozitif enerji tüketimi sağlamıştır. Önerilen algoritmadan elde edilen optimal sıra R_7, R_6, R_3, R_5, R_1, Çizelge

5.11'de gösterildiği gibi matematiksel model hesabına göre sürdürülebilirlik değerinde % 1.34 iyileşme sağlamıştır. Çalışan Yönetim Sistemi projesi test sürecinde de yoğun bir veri aktarımı olduğu gözlemlenmiştir. Geliştirilen yöntem uygulandıktan sonra elde edilen optimum yeniden düzenleme sırası R_7, R_3, R_6, R_4, Çizelge 5.17'de gösterildiği gibi matematiksel model hesabına göre sürdürülebilirlik değerinde % 1.62 iyileşme sağlamıştır. Tekniklerin farklı sıralarda uygulanması, bu proje için de tüm algoritmaların sonuçlarında pozitif enerji tüketimi sağlamıştır. Çünkü R_3 ve R_6'nın önceliklendirilmesi sonucunda değişkenlerin bellekte daha az yer ayrılmasına izin verilmiş ve bunların önbellekten alma süreleri kısalmıştır. Sonuç olarak, komut yürütme sırasındaki döngü sayısı azalması ile enerji tüketimi azalmıştır.

Şekil 5.5'te verilen sonuçlar incelendiğinde, Hastane Yönetim Sistemi projesinin orijinal kodları karşılaştırma algoritmalarından yüksek enerji tüketimine sahiptir. Önerilen algoritma uygulandıktan sonra elde edilen optimum yeniden düzenleme sırası $R_2, R_6, R_7, R_3, R_4, R_5, R_1$, Çizelge 5.15'te gösterildiği gibi matematiksel model hesabına göre sürdürülebilirlik değerinde % 2.14 iyileşme sağlamıştır. Fakat Boruvka algoritması uygulandıktan sonra elde edilen optimum yeniden düzenleme sırası $R_1, R_6, R_2, R_3, R_4, R_5, R_7$, daha az enerji tüketmiştir. Önerilen algoritma proje kodlarının sürdürülebilirliğini geliştirmesine yardımcı olmuş; fakat tekniklerin optimal sırasının enerji maliyetini istenilen seviyede düşürmediği görülmüştür.

Otel Yönetim Sistemi projesinin Şekil 5.4'te verilen sonuçları incelendiğinde, yöntemimizin alt çeyreği, orijinal kodun enerji tüketim sonucundan daha düşüktür. Yöntemimizin çeyrekler arası aralıkları diğer karşılaştırma algoritmalarına göre daha küçük olduğu için yöntemimiz daha tutarlı sonuçlar elde etmiştir. Önerilen algoritma uygulandıktan sonra elde edilen optimum yeniden düzenleme sırası R_5, R_6, R_4, R_3, Çizelge 5.13'te gösterildiği gibi matematiksel model hesabına göre sürdürülebilirlik değerinde % 6.27 iyileşme sağlamıştır. Sorgu işleminde çağrılan kod bloklarında yapılan R_3, R_4, R_5, R_6 tekniklerinin önceliklendirilmesi ile karmaşık ifadelerin bellekte birden fazla kopyası oluşmasının da önüne geçilmekte ve değer karmaşası sorunu ortadan giderilmektedir. Böylece kodun sürdürülebilirliği önemli derecede iyileştirilmiş ve enerji maliyeti azaltılmıştır.

Kullanılan yeniden düzenleme tekniklerine ait bulguların analizi sonucunda yazılım kalitesi açısından değerlendirilmesi aşağıda sırayla verilmektedir.

R_1, yazılım bakım sürecini kolaylaştıran yeniden düzenleme tekniklerinden biridir. Bu yeniden düzenleme tekniğinin temel amacı, bir değişkenin erişim izinlerini ayarlamaktır. Get ve set işlevleri, belirli bir alan seçilerek yapılandırılır. Bu yeniden düzenleme, sınıf içindeki bir öznitelik için getter ve setter oluşturur. Öznitelik private yapılır ve içerdiği sınıfın dışındaki herhangi bir manipülasyondan korunur. Bu genellikle bir sınıfın dahili alanlarının kapsüllenmesine izin veren iyi bir tasarım uygulaması olsa da, get ve set kullanılması, ek işlev çağrılarını temsil ettikleri için projelerde kulllanılan diğer tekniklerle etkileşiminde daha az performans göstermesine neden olduğu sonucuna varılmıştır. Herhangi bir performans endişesi varsa kapsülleme olumsuz bir etki oluşturabilir.

R_2 ile iç içe döngü yapıları tek döngülü yapılar haline dönüştürülebilmektedir. İterasyon sayısındaki artış fazla hesaplamaya ve hafızada fazla yer ayrılmasına sebep olmaktadır. Tek döngü yapısı iterasyon sayısını azaltmaktadır. Böylece enerji tüketimini azaltmaya yönelik olumlu etki göstermiştir. R_3 ile geçici değişkenin silinmesi, gereksiz geçici değişkeni önbelleklerden ve ana belleklerden alma süresini kısaltmaktadır. Bu durum optimal sıranın uygulanmasında enerji verimliliği açısından olumlu katkılar sağlamıştır.

R_4'ün uygulandığı yerlerde sınıfların bağlı olma eğilimlerinde değişiklik gözlenmemiştir. Fakat sonuçlar, düşük seviyeli kod yeniden düzenlemelerinin incelenmesinin sistemlere ilişkin içgörüleri nasıl ortaya çıkarabileceğini göstermektedir. Veri taşınması gereken işlemlerde nesnenin bütün verisi kopyalanmaz. Seçilen ifadenin tüm tekrarları, yeni oluşturulan değişken ile bir referans kullanılarak değiştirilebilir veya yalnızca seçilen ifade değiştirilebilir. Böylece işlemcideki fazla yük ortadan kaldırılır. Sonuç olarak enerji tasarrufu sağlanır. R_5 kullanımı bellekte iki yinelenen yöntem yerine bir adet yöntem için yer ayrılmasına olanak sağlar. Tek yöntem kullanımı fetch-decode-execute döngüsünü azaltmakta olup enerji tüketimini azalttığı sonucuna varılmıştır.

Sihirli sayılar genellikle kaynak kod içerisinde belirgin olmayan özel değerler içeren sayıları içermektedirler. Bu sayılar kod içerisinde birden fazla yerde farklı amaçlar

için kullanıldığı zaman kodun bakımında, değiştirilmesinde zorluklar yaşanmakta ve okunabilirliğini ters yönde etkileyebilmektedirler. R_6 kullanımı ile bu durumlar ortadan kaldırılmış ve bu tekniğin kullanıldığı projelerin kaynak kodları okunabilirlik niteliği açısından büyük gelişmeler kaydedildiği görülmüştür. Aynı zamanda projelerin kaynak kodlarında sayının bellekte birden fazla kopyası oluşmasının önüne geçilmiş fakat kaynak kodların bakım yapılabilirlik indeksi üzerinde olumlu katkı sağlamadığı görülmüştür. Diğer taraftan tekniklerle beraber kullanıldığında enerji verimliliği açısından olumlu gelişme gösterdiği tespit edilmiştir.

Bazen sınıfın içerdiği yöntemi gizlemek zengin bir arayüz oluşturmamızı sağlar ve sınıflara daha fazla davranış katar. Ek davranışların geliştirilmesi yolu ile bazen değişkenlere erişiminler doğrudan sağlanabilmekte ve sınıftaki ek yöntemlerin kaldırılması olanağı ortaya çıkabilmektedir. Böylece hafızadaki fazla yük durumunu ortadan kalkmış olur. Bunun sonucunda enerji tasarrufu sağlanır. R_7 kullanımı içeren projelerde bireysel kullanımın projelerin enerji verimliliği açısından olumlu yönde katkı sağladığı söylenemez. Fakat tekniklerin optimum sırasının uygulanmasında ve diğer tekniklerle etkileşime girmesi sonucunda yazılım kalitesi ve sürdürülebilirliğine olumlu katkı sağlamış aynı zamanda enerji verimliliğine pozitif yönlü etki yaptığı görülmüştür.

6. SONUÇ VE ÖNERİLER

Yeşil yazılım, enerji sızıntılarına neden olan faktörlerin tanımlanmasını ve bunlarla başa çıkmak için pratik çözümler getirmeyi hedefleyen yazılım mühendisliği dalıdır. Geliştirilen yazılımsal çözümler, kod işlevini bozmadan yazılım için izlenen yazılım geliştirme süreci, kullanılan yeniden düzenleme tekniği gibi konularda yapılan yenilik ve geliştirmeleri gözlemlemeye dayanır. Farklı özelliklerdeki yazılımlar için farklı kalite hedefleri ve çözümleri ön plana çıkmaktadır. Enerji sızıntılarını yeniden düzenleme teknikleri ile iyileştirmek çözümlerden biridir. Fakat enerji sızıntılarını iyileştirmek için yeniden düzenleme yapmak zorlu bir süreçtir. Çünkü yeniden düzenlemeler programın tasarım alanında gerçekleşirken programın derlenmiş sürümü belirli bir donanım yapılandırmasında çalıştırıldığında enerji tüketimi tespit edilebilmektedir. Diğer bir çözüm ise yeniden düzenleme sürecini otomatikleştirmektir. Enerji tüketimini azaltmak için yazılımın yeniden düzenlenmesini otomatikleştirmeye çalışmak bazı soruları da beraberinde gündeme getirmektedir. "Kaç yeniden düzenleme yapılmalı?" veya "nereye uygulanmalıdır?" gibi soruların cevapları önem arz etmektedir. Cevapların önemsiz olması düşünülemez. Hatta bazı yeniden düzenleme uygulamaları net olabilir; fakat pratikte yazılım mühendisleri tarafından yazılım geliştirilirken veya bakım yapılırken yeniden düzenlemenin nerede durdurulacağına dair bir çizgi çizilmelidir. Çünkü her yeniden düzenleme tekniğinin enerji verimliliğine etkisi tümleşik geliştirme ortamlarına ait yeniden düzenleme otomatik desteğinde gösterilmez. Ayrıca enerji israfına yol açan kodlar çalışma zamanında herhangi bir hata oluşturmasa da kaynak kodun bakımını zorlaştırmaktadır. Bu nedenle kodun dışsal davranışında hiçbir değişiklik olmayacak şekilde iç yapısını değiştirmek için yeniden düzenleme teknikleri tercih edilmektedir.

Tez kapsamında yazılımın enerji verimliliğini ve sürdürülebilirlik değerini iyileştiren optimum yeniden düzenleme teknikleri sırası tespit edildi. Yeniden düzenleme teknikleri optimum sırasını elde etmek için Prim tabanlı bir algoritma geliştirildi. Ayrıca enerji israfı olarak kabul edilebilecek ham kaynak kodları üzerinde yeniden düzenleme tekniklerini gerçekleştirmek için bir C# masaüstü uygulaması tasarlandı. Geliştirilen uygulama ile yeniden düzenleme kullanılarak enerji israfına yol açan kodların düzenlenmesi yazılımı daha sürdürülebilir, daha okunaklı ve enerji tüketimi

açısından daha verimli hale getirdiği tespit edildi. Sonuçlardaki değerlerin hesaplanması nesneye yönelik ölçülere dayanmaktadır.

Sonuçlar kodun karmaşıklığını ve boyutunu gösteren kriterlerin enerji tüketimi açısından bir önceliklendirme algoritması tasarlamak için faydalı olduğunu göstermektedir. Diğer bir gösterge ise yeniden düzenleme teknikleri sırasının esas olarak yararlanılacak kaynak koda bağlı olduğuna dair bazı işaretlerin varlığıdır. Bunlardan biri, bağlaşım sayısı yazılım metriğinin genelde projelerde aynı kalma eğiliminde iken geliştirilen matematiksel model yardımıyla elde edilen sıranın yazılım kalitesi ve sürdürülebilirlik değerinin yanında enerji verimliliğini üst düzeye çıkardığının tespitidir. Projelerdeki kaynak kodların içerdikleri karmaşıklık, yapışıklık gibi yazılım özellikleri ise kaynak kodlara göre farklılık göstermiş ve kaynak kodların detaylı incelenmesi ile optimum sıralar elde edilebilmiştir. Bu yüzden yazılımda kullanılan kaynak kodlara göre optimum sıralamalar değişiklik gösterebilmektedir. Bu kodların özelliklerinin iyi analiz edilmesi ile yazılımı daha sürdürülebilir ve enerji tüketimi açısından daha verimli hale getirebilmek mümkündür.

Bu tez çalışmasının bazı kısıtları vardır: Bunlardan birincisi, önerilen algoritman belirli yeniden düzenleme teknikleri için çalışır. İkincisi, sınıfların en kritik projelerden seçilmesi önemli bir zaman sınırlaması oluşturmaktadır. Üçüncüsü, tezin kapsamının iki tür programlama diliyle sınırlı olmasıdır. Son olarak, yeniden düzenleme tekniklerinden biri olan Simplify Nested Loop'un uyumluluğu, geliştirilen araçla belirlenebilir; ancak, tekniğin tespiti sonucunda kod statik kod analizine ve manuel kod incelemesine tabi tutulur. Çünkü iç içe döngü yapısının tek bir döngüye uygunluğu statik kod analizi ve manuel kod incelemesi ile mümkündür. Manuel ile yeniden düzenleme genellikle hataya açıktır ve zaman alır (Roberts, 1999). Bu, geliştirilen araç için bir kısıtlama oluşturur.

Sonuç olarak, bu tezin yeniden düzenleme ve enerji verimliliği önceliklendirmesi hakkında ampirik bilginin yapısı açısından yazılım mühendisliği camiasına önemli bir katkı sağladığına inanıyoruz; ancak, sonuçlarımızın bizimki dışında belirli bir ortam için genellenebileceğini varsayamayız. Daha geniş bir yazılım sistemlerinde

daha fazla doğrulama arzu edilir ve geliştirilen aracın diğer dillerde kullanışlılığının genelleştirilmesine yardımcı olmak için daha fazla araştırma gerekir.

Çalışmanın kapsamı, yeniden düzenleme tekniklerinin kullanım sırası ile kaynak kodun yürütme zamanı azaltılacak şekilde enerji verimliliğinin geliştirilmesine yönelik yaklaşımlara uyarlanabilir. Kaynak koduna sıralı yeniden düzenleme tekniklerinin uygulanmasının yazılım güvenliği için faydalı olup olmadığını araştırmak yeşil yazılım açışından yeni çözümler getirebilir. Geçmiş veriler kaydedilirse enerji tüketimini tahmin eden modeller, yeniden düzenleme tekniklerinin birleştirilmesinde yardımcı olabilir. Bunlara ek olarak kapsam, diğer programlama dilleri ile oluşturulmuş yazılımlar için sezgisel yöntemler gibi farklı yaklaşımlar ile genişletilebilir.

KAYNAKLAR

Abreu, F.B., Carapuça, R., 1994. Candidate metrics for object-oriented software within a taxonomy framework. Journal of Systems and Software, 26(1), 87–96.

Agarwal, S., Nath, A., Chowdhury, D., 2012. Sustainable approaches and good practices in green software engineering. International Journal of Research and Reviews in Computer Science, 3(1), 1425.

Aggarwal, K., Zhang, C., Campbell, J.C., Hindle, A., Stroulia, E., 2014. The power of system call traces: predicting the software energy consumption impact of changes. CASCON. (pp. 219–233).

Ahn, Y., Suh, J., Kim, S., Kim, H., 2003. The software maintenance project effort estimation model based on function points. Journal of Software maintenance and evolution: Research and practice, 15(2), 71–85.

Al Qutaish, R.E., Abran, A., 2005. An analysis of the design and definitions of Halstead's metrics. 15th Int. Workshop on Software Measurement (IWSM'2005). Shaker-Verlag. (pp. 337–352).

Alpernas, K., Feldman, Y.M., Peleg, H., 2020. The wonderful wizard of LoC: paying attention to the man behind the curtain of lines-of-code metrics. Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. (pp. 146–156).

Anwar, H., Pfahl, D., Srirama, S.N., 2019. Evaluating the impact of code smell refactoring on the energy consumption of Android applications. 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE, (pp. 82–86).

Ardito, L., Coppola, R., Barbato, L., Verga, D., 2020. A tool-based perspective on software code maintainability metrics: a systematic literature review. Scientific Programming.

Arora, D., Khanna, P., Tripathi, A., Sharma, S., Shukla, S., 2011. Software quality estimation through object oriented design metrics. Int. J. Computer Science and Network Security, 11(4), 100–104.

Bangash, A.A., Sahar, H., Beg, M.O., 2017. A methodology for relating software structure with energy consumption. 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, (pp. 111–120).

Bansiya, J., Davis, C.G., 2002. A hierarchical model for object-oriented design quality assessment. IEEE Transactions on software engineering, 28(1), 4–17.

Barack, O., Huang, L., 2018. Effectiveness of Code Refactoring Techniques for Energy Consumption in a Mobile Environment. Proceedings of the International

Conference on Software Engineering Research and Practice (SERP). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing, (pp. 165–171).

Berry, M.J., Linoff, G.S., 2004. Data mining techniques: for marketing, sales, and customer relationship management. John Wiley & Sons.

Bessa, T., Gull, C., Quintão, P., Frank, M., Nacif, J., Pereira, F.M.Q., 2019. JetsonLEAP: A framework to measure power on a heterogeneous system-on-a-chip device. Science of Computer Programming, 173, 21–36.

Biggs, N., Lloyd, E.K., Wilson, R.J., 1986. Graph Theory. Oxford University Press.

Bill, W., 2015. Statements - C Programming Guide | Microsoft Docs. `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/statements`. Erişim Tarihi: 01.02.2021.

Binkley, D., Ceccato, M., Harman, M., Ricca, F., Tonella, P., 2006. Tool-supported refactoring of existing object-oriented code into aspects. IEEE Transactions on Software Engineering, 32(9), 698–717.

Bondy, J.A., Murty, U.S.R., vd., 1976. Graph theory with applications. Macmillan London.

Boruvka, O., 1926. O jistém problému minimálním.

Candela, I., Bavota, G., Russo, B., Oliveto, R., 2016. Using cohesion and coupling for software remodularization: Is it enough? ACM Transactions on Software Engineering and Methodology (TOSEM), 25(3), 1–28.

Chan-Jong-Chu, K., Islam, T., Exposito, M.M., Sheombar, S., Valladares, C., Philippot, O., Grua, E.M., Malavolta, I., 2020. Investigating the correlation between performance scores and energy consumption of mobile web apps. Proceedings of the Evaluation and Assessment in Software Engineering. (pp. 190–199).

Chen, S.M., Chang, T.H., 2001. Finding multiple possible critical paths using fuzzy PERT. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 31(6), 930–937.

Chen, W.K., 2012. Applied graph theory. Elsevier.

Chen, W.K., Wang, J.C., 2012. Bad smells and refactoring methods for gui test scripts. 2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing. IEEE, (pp. 289–294).

Chhikara, A., Chhillar, R., Khatri, S., 2011. Evaluating the impact of different types of inheritance on the object oriented software metrics. International Journal of Enterprise Computing and Business Systems, 1(2), 1–7.

Chidamber, S.R., Kemerer, C.F., 1991. Towards a metrics suite for object oriented design. Conference proceedings on Object-oriented programming systems, languages, and applications. (pp. 197–211).

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Transactions on software engineering, 20(6), 476–493.

Chung, S., Condon, A., 1996. Parallel implementation of Bouvka's minimum spanning tree algorithm. Proceedings of International Conference on Parallel Processing. IEEE, (pp. 302–308).

Civelekoğlu, G., Bıyık, Y., 2020. Isparta ilinde karayolu kaynaklı karbon ayak izinin hesaplanması. Bilge International Journal of Science and Technology Research, 4(2), 78–87.

CodeCount™, 2020. USC's Center for Systems and Software Engineering. `http://csse.usc.edu`. Erişim Tarihi: 02.01.2021.

Coleman, D., Ash, D., Lowther, B., Oman, P., 1994. Using metrics to evaluate software system maintainability. Computer, 27(8), 44–49.

Connolly Bree, D., Cinnéide, M.Ó., 2020. Inheritance versus Delegation: which is more energy efficient? Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. (pp. 323–329).

Counsell, S., Liu, X., Swift, S., Buckley, J., English, M., Herold, S., Eldh, S., Ermedahl, A., 2015. An exploration of the'introduce explaining variable'refactoring. Scientific Workshop Proceedings of the XP2015. (pp. 1–5).

Cox, R., 2007. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby,...). URL: http://swtch. com/rsc/regexp/regexp1. html.

Cruz, L., Abreu, R., 2017. Performance-based guidelines for energy efficient mobile applications. 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft). IEEE, (pp. 46–57).

Dang, S., Ahmad, P.H., 2014. Text mining: Techniques and its application. International Journal of Engineering & Technology Innovations, 1(4), 22–25.

Dang, S., Ahmad, P.H., 2015. A review of text mining techniques associated with various application areas. International Journal of Science and Research (IJSR), 4(2), 2461–2466.

Demeyer, S., Ducasse, S., Nierstrasz, O., 2000. Finding refactorings via change metrics. ACM SIGPLAN Notices, 35(10), 166–177.

Dey, A., Pal, A., 2013. Fuzzy graph coloring technique to classify the accidental zone of a traffic control. Annals of Pure and Applied Mathematics, 3(2), 169–178.

Dey, A., Pradhan, R., Pal, A., Pal, T., 2015. The fuzzy robust graph coloring problem. Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: theory and applications (FICTA) 2014. Springer, (pp. 805–813).

Dick, M., Drangmeister, J., Kern, E., Naumann, S., 2013. Green software engineering with agile methods. 2013 2nd international workshop on green and sustainable software (GREENS). IEEE, (pp. 78–85).

Dijkstra, E., 1960. Some theorems on spanning subtrees of a graph. Indag. math, 22(2), 196–199.

Dijkstra, E.W., vd., 1959. A note on two problems in connexion with graphs. Numerische mathematik, 1(1), 269–271.

Dinita, R.I., Wilson, G., Winckles, A., Cirstea, M., Jones, A., 2013. Hardware loads and power consumption in cloud computing environments. 2013 IEEE International Conference on Industrial Technology (ICIT). IEEE, (pp. 1291–1296).

Dubey, S.K., Rana, A., 2011. Assessment of maintainability metrics for object-oriented software system. ACM SIGSOFT Software Engineering Notes, 36(5), 1–7.

Durnová, H., 2006. Borvka, Otakar: About Otakar Borvka.

Elish, K.O., Alshayeb, M., 2011. A classification of refactoring methods based on software quality attributes. Arabian Journal for Science and Engineering, 36(7), 1253–1267.

Ergasheva, S., Khomyakov, I., Kruglov, A., Succil, G., 2020. Metrics of energy consumption in software systems: a systematic literature review. IOP Conference Series: Earth and Environmental Science. IOP Publishing, Volume 431, (s. 012051).

Erickson, J., 2019. Minimum spanning trees. `https://jeffe.cs.illinois.edu/teaching/algorithms/`. Erişim Tarihi: 09.03.2021.

Feldman, R., Sanger, J., vd., 2007. The text mining handbook: advanced approaches in analyzing unstructured data. Cambridge university press.

Fenton, N., 1994. Software measurement: A necessary scientific basis. IEEE Transactions on software engineering, 20(3), 199–206.

Fenton, N.E., Neil, M., 2000. Software metrics: roadmap. Proceedings of the Conference on the Future of Software Engineering. (pp. 357–370).

Fowler, M., 2018. Refactoring: improving the design of existing code. Addison-Wesley Professional.

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., 1999. Refactoring: improving the design of existing code, ser. Addison Wesley object technology series., Addison-Wesley.

Gabow, H.N., Galil, Z., Spencer, T., Tarjan, R.E., 1986. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. Combinatorica, 6(2), 109–122.

García-Martín, E., Rodrigues, C.F., Riley, G., Grahn, H., 2019. Estimation of energy consumption in machine learning. Journal of Parallel and Distributed Computing, 134, 75–88.

Gottschalk, M., Jelschen, J., Winter, A., 2013. Energy-efficient code by refactoring. Softwaretechnik-Trends, 33(2).

Gottschalk, M., Josefiok, M., Jelschen, J., Winter, A., 2012. Removing energy code smells with reengineering services. Informatik.

Graham, R.L., Hell, P., 1985. On the history of the minimum spanning tree problem. Annals of the History of Computing, 7(1), 43–57.

Gray, D., Bowes, D., Davey, N., Sun, Y., Christianson, B., 2011. The misuse of the NASA metrics data program data sets for automated software defect prediction. 15th Annual Conference on Evaluation & Assessment in Software Engineering (EASE 2011). IET, (pp. 96–103).

Grechanik, M., McMillan, C., DeFerrari, L., Comi, M., Crespi, S., Poshyvanyk, D., Fu, C., Xie, Q., Ghezzi, C., 2010. An empirical investigation into a large-scale Java open source code repository. Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. (pp. 1–10).

Gross, J.L., Yellen, J., 2003. Handbook of graph theory. CRC press.

Gross, J.L., Yellen, J., 2005. Graph theory and its applications. CRC press.

Gupta, S., Chug, A., 2020. Software maintainability prediction using an enhanced random forest algorithm. Journal of Discrete Mathematical Sciences and Cryptography, 23(2), 441–449.

Gupta, V., Lehal, G.S., vd., 2009. A survey of text mining techniques and applications. Journal of emerging technologies in web intelligence, 1(1), 60–76.

Halstead, M.H., 1977. Elements of software engineering. Elsevier, 7(127).

Harary, F., 2015. A seminar on graph theory. Courier Dover Publications.

Hegedűs, P., Kádár, I., Ferenc, R., Gyimóthy, T., 2018. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. Information and Software Technology, 95, 313–327.

Hennessy, J.L., Patterson, D.A., 2011. Computer architecture: a quantitative approach. Elsevier.

Higo, Y., Kamiya, T., Kusumoto, S., Inoue, K., 2004. Refactoring support based on code clone analysis. International Conference on Product Focused Software Process Improvement. Springer, (pp. 220–233).

Hindle, A., 2015. Green mining: a methodology of relating software change and configuration to power consumption. Empirical Software Engineering, 20(2), 374–409.

Hoque, M.A., Siekkinen, M., Khan, K.N., Xiao, Y., Tarkoma, S., 2015. Modeling, profiling, and debugging the energy consumption of mobile devices. ACM Computing Surveys (CSUR), 48(3), 1–40.

Hotho, A., Nürnberger, A., Paaß, G., 2005. A brief survey of text mining. Ldv Forum. Citeseer, Volume 20, (pp. 19–62).

Hotta, K., Sasaki, Y., Sano, Y., Higo, Y., Kusumoto, S., 2012. An empirical study on the impact of duplicate code. Advances in Software Engineering, 2012.

Hsu, C.H., Kremer, U., 2003. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation. (pp. 38–48).

Ikerionwu, C., 2010. Cyclomatic Complexity As a Software Metric. International Journal of Academic Research, 2(3).

İlhan, S., Duru, N., Karagöz, Ş., Sağır, M., 2008. Metin madenciliği ile soru cevaplama sistemi. Elektronik ve Bilgisayar Mühendisliği Sempozyumu (ELECO), Bursa, 26, 30.

Intel, 2020. Intel® Power Gadget. `https://software.intel.com/content/www/us/en/develop/articles/intel-power-gadget.html`. Erişim Tarihi: 18.03.2021.

Jarník, V., 1930. O jistém problému minimálním.(Z dopisu panu O. Borvkovi).

Jones, C., 1994. Software metrics: good, bad and missing. Computer, 27(9), 98–100.

Kan, S.H., 2003. Metrics and models in software quality engineering. Addison-Wesley Professional.

Karger, D.R., Klein, P.N., Tarjan, R.E., 1995. A randomized linear-time algorithm to find minimum spanning trees. Journal of the ACM (JACM), 42(2), 321–328.

Kataoka, Y., Imai, T., Andou, H., Fukaya, T., 2002. A quantitative evaluation of maintainability enhancement by refactoring. International Conference on Software Maintenance, 2002. Proceedings. IEEE, (pp. 576–585).

Kaur, A., Kaur, M., 2016. Analysis of code refactoring impact on software quality. MATEC Web of Conferences. EDP Sciences, (pp. 02–012).

Kaur, J., Singh, S., 2016. Neural network based refactoring area identification in software system with object oriented metrics. Indian Journal of Science and Technology, 9(10), 1–8.

Kaxiras, S., Martonosi, M., 2008. Computer architecture techniques for power-efficiency. Synthesis Lectures on Computer Architecture, 3(1), 1–207.

Keong, C.K., Wei, K.T., Ghani, A.A.A., Sharif, K.Y., 2015. Toward using software metrics as indicator to measure power consumption of mobile application: A case study. 2015 9th Malaysian Software Engineering Conference (MySEC). IEEE, (pp. 172–177).

Kexugit, 2021. Maintainability Index Range and Meaning. `https://docs.microsoft.com/tr-tr/archive/blogs/codeanalysis/maintainability-index-range-and-meaning`. Erişim Tarihi: 17.01.2021.

Kim, D., Hong, J.E., Yoon, I., Lee, S.H., 2018. Code refactoring techniques for reducing energy consumption in embedded computing environment. Cluster computing, 21(1), 1079–1095.

King, R.S., 2015. Cluster analysis and data mining: An introduction. Stylus Publishing, LLC.

Koziel, S., Yang, X.S., 2011. Computational optimization, methods and algorithms. Springer.

Kruskal, J.B., 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical society, 7(1), 48–50.

Kumar, L., Bhatia, P.K., 2013. Text mining: concepts, process and applications. Journal of Global Research in Computer Science, 4(3), 36–39.

Kumar, R.P., 2017. Internal and External Measures of Code Quality Impact on Refactoring. SSRG Int. J. Civ. Eng.–ICCREST'17, 70–72.

Kwon, Y., Lee, Z., Park, Y., 2013. Performance-based refactoring: identifying & extracting move-method region. Journal of KIISE: Software and Applications, 40(10), 567–574.

Kwon, Y.W., Tilevich, E., 2013. Reducing the energy consumption of mobile applications behind the scenes. 2013 IEEE International Conference on Software Maintenance. IEEE, (pp. 170–179).

Levitin, A., 2011. Introduction To Design And Analysis Of Algorithms, 3/E. Pearson.

Li, H., Xia, Q., Wang, Y., vd., 2017. Research and improvement of kruskal algorithm. Journal of Computer and Communications, 5(12), 63.

Li, W., Henry, S., 1993. Object-oriented metrics that predict maintainability. Journal of systems and software, 23(2), 111–122.

Loberman, H., Weinberger, A., 1957. Formal procedures for connecting terminals with a minimum total wire length. Journal of the ACM (JACM), 4(4), 428–437.

LocMetrics, 2019. LocMetrics Tool. `https://www.cheonghyun.com/blog/120`. Erişim Tarihi: 01.02.2021.

Lorenz, M., Kidd, J., 1994. Object-oriented software metrics: a practical guide. Prentice-Hall, Inc.

Lyu, Y., Alotaibi, A., Halfond, W.G., 2019. Quantifying the performance impact of SQL antipatterns on mobile applications. 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, (pp. 53–64).

Mahmoud, S.S., Ahmad, I., 2013. A green model for sustainable software engineering. International Journal of Software Engineering and Its Applications, 7(4), 55–74.

Mahouachi, R., Kessentini, M., Cinnéide, M.Ó., 2013. Search-based refactoring detection using software metrics variation. International Symposium on Search Based Software Engineering. Springer, (pp. 126–140).

Makkar, G., Chhabra, J.K., Challa, R.K., 2012. Object oriented inheritance metric-reusability perspective. 2012 International Conference on Computing, Electronics and Electrical Technologies (ICCEET). IEEE, (pp. 852–859).

Mancebo, J., Calero, C., García, F., 2021a. Does maintainability relate to the energy consumption of software? A case study. Software Quality Journal, 29(1), 101–127.

Mancebo, J., García, F., Calero, C., 2021b. A process for analysing the energy efficiency of software. Information and Software Technology, 134, 106560.

Manning, C., Raghavan, P., Schütze, H., 2010. Introduction to information retrieval. Natural Language Engineering, 16(1), 100–103.

Manotas, I., Bird, C., Zhang, R., Shepherd, D., Jaspan, C., Sadowski, C., Pollock, L., Clause, J., 2016. An empirical study of practitioners' perspectives on green software engineering. 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, (pp. 237–248).

Marpaung, F., 2020. Comparative of prim's and boruvka's algorithm to solve minimum spanning tree problems. Journal of Physics: Conference Series. IOP Publishing, (pp. 012–043).

McCabe, T.J., 1976. A complexity measure. IEEE Transactions on software Engineering, (4), 308–320.

Mens, T., Tourwé, T., 2004. A survey of software refactoring. IEEE Transactions on software engineering, 30(2), 126–139.

Menzies, T., Di Stefano, J.S., Chapman, M., McGill, K., 2002. Metrics that matter. 27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings. IEEE, (pp. 51–57).

Microsoft, 2021. Visual Studio IDE, Kod Düzenleyicisi, Azure DevOps ve App Center. `https://visualstudio.microsoft.com/tr/`. Erişim Tarihi: 11.02.2021.

Microsoft, 2022. Dizin yöntemleri. `https://docs.microsoft.com/en-us/dotnet/api/system.string.clone?view=net-6.0`. Erişim Tarihi: 02.02.2022.

Mikheev, A., 2003. Text segmentation. The Oxford handbook of computational linguistics. Oxford University Press, (pp. 201–218).

Misra, S., Akman, I., Koyuncu, M., 2011. An inheritance complexity metric for object-oriented code: A cognitive approach. Sadhana, 36(3), 317–337.

Moore, I., 1996. Automatic inheritance hierarchy restructuring and method refactoring. ACM SIGPLAN Notices, 31(10), 235–250.

Morales, R., Saborido, R., Khomh, F., Chicano, F., Antoniol, G., 2017. Earmo: an energy-aware refactoring approach for mobile apps. IEEE Transactions on Software Engineering, 44(12), 1176–1206.

Najm, N., 2014. Measuring Maintainability Index of a Software Depending on Line of Code Only. 16, 64–69.

Naumann, S., Dick, M., Kern, E., Johann, T., 2011. The greensoft model: A reference model for green and sustainable software and its engineering. Sustainable Computing: Informatics and Systems, 1(4), 294–304.

Nešetřil, J., Milková, E., Nešetřilová, H., 2001. Otakar Borvka on minimum spanning tree problem Translation of both the 1926 papers, comments, history. Discrete mathematics, 233(1-3), 3–36.

Neumann, F., Witt, C., 2010. Combinatorial optimization and computational complexity. Bioinspired computation in combinatorial optimization, Springer. (pp. 9–19).

Newell, D.B., Tiesinga, E., 2019. The international system of units (SI). NIST Special Publication, 330, 1–138.

Nguyen, V., Deeds-Rubin, S., Tan, T., Boehm, B., 2007. A SLOC counting standard. Cocomo ii forum. Citeseer, (pp. 1–16).

Ogala, J.O., Ojie, D.V., 2020. Comparative analysis of c, c++, c# and java programming languages. GSJ, 8(5).

Opdyke, W.F., Johnson, R.E., 1993. Creating abstract superclasses by refactoring. Proceedings of the 1993 ACM conference on Computer science. (pp. 66–73).

Padhy, N., Satapathy, S., Singh, R., 2018. State-of-the-art object-oriented metrics and its reusability: a decade review. Smart Computing and Informatics, 431–441.

Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., De Lucia, A., 2019. On the impact of code smells on the energy consumption of mobile applications. Information and Software Technology, 105, 43–55.

Papadimitriou, C.H., Steiglitz, K., 1998. Combinatorial optimization: algorithms and complexity. Courier Corporation.

Papadopoulos, L., Marantos, C., Digkas, G., Ampatzoglou, A., Chatzigeorgiou, A., Soudris, D., 2018. Interrelations between software quality metrics, performance and energy consumption in embedded applications. Proceedings of the 21st International Workshop on software and compilers for embedded systems. (pp. 62–65).

Park, J.J., Hong, J.E., 2013. An Approach to improve software safety by Code refactoring. Proc. of Korea Computer Congress. (pp. 532–534).

Park, J.J., Hong, J.E., Lee, S.H., 2014. Investigation for Software Power Consumption of Code Refactoring Techniques. SEKE. (pp. 717–722).

Park, R.E., 1992. Software size measurement: A framework for counting source statements. Technical report, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst.

Parsai, A., Murgia, A., Soetens, Q.D., Demeyer, S., 2015. Mutation testing as a safety net for test code refactoring. Scientific Workshop Proceedings of the XP2015. (pp. 1–7).

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J.P., Saraiva, J., 2021. Ranking programming languages by energy efficiency. Science of Computer Programming, 205, 102609.

Prim, R.C., 1957. Shortest connection networks and some generalizations. The Bell System Technical Journal, 36(6), 1389–1401.

Qualcomm, 2018. Trepn Power Profiler. `https://developer.qualcomm.com/forums/software/trepn-power-profiler`. Erişim Tarihi: 18.03.2021.

R, 2021. R for Windows. `https://cran.r-project.org/bin/windows/`. Erişim Tarihi: 18.02.2021.

Ramadhan, Z., Siahaan, A.P.U., Mesran, M., 2018. Prim and Floyd-Warshall Comparative Algorithms in Shortest Path Problem. Proceedings of the Joint Workshop KO2PI and The 1st International Conference on Advance & Scientific Innovation. (pp. 47–58).

Refactoring, 2021. Hello, world! https://refactoring.guru/. Erişim Tarihi: 02.03.2021.

Roberts, D.B., 1999. Practical analysis for refactoring. University of Illinois at Urbana-Champaign.

Rosenberg, L.H., Hyatt, L.E., 1997. Software quality metrics for object-oriented environments. Crosstalk journal, 10(4), 1–6.

Ryder, C., Thompson, S.J., 2005. Software metrics: measuring Haskell. Trends in Functional Programming. (pp. 31–46).

Sahin, C., Pollock, L., Clause, J., 2014. How do code refactorings affect energy usage? Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. (pp. 1–10).

Sehgal, R., Mehrotra, D., Nagpal, R., Sharma, R., 2020. Green software: Refactoring approach. Journal of King Saud University-Computer and Information Sciences.

Seker, S.E., 2008. Külliyat (corpus). https://bilgisayarkavramlari.com/2008/03/25/kulliyat-corpus/. Erişim Tarihi: 16.01.2022.

Seker, S.E., 2015a. Çizge teorisi (graph theory). YBS ansiklopedi, 2(2), 17–29.

Seker, S.E., 2015b. Metin Madenciliği (Text Mining). YBS ansiklopedi, 2(3), 30–32.

Shenoy, S.S., Eeratta, R., 2011. Green software development model: An approach towards sustainable software development. 2011 Annual IEEE India Conference. IEEE, (pp. 1–6).

Silva, W.G., Brisolara, L., Corrêa, U.B., Carro, L., 2010. Evaluation of the impact of code refactoring on embedded software efficiency. Proceedings of the 1st Workshop de Sistemas Embarcados. (pp. 145–150).

Srivastava, K., Tyagi, R., 2013. Shortest Path Algorithm For Satellite Network. International Journal of Innovative Research and Development (ISSN 2278–0211), 2(5).

Sudhamani, M., Rangarajan, L., 2015. Structural similarity detection using structure of control statements. Procedia computer science, 46, 892–899.

Sumathy, K., Chidambaram, M., 2013. Text mining: concepts, applications, tools and issues-an overview. International Journal of Computer Applications, 80(4).

Sun, Y., Wang, Y., Yang, H., 2017. Energy-efficient SQL query exploiting RRAM-based process-in-memory structure. 2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA). IEEE, (pp. 1–6).

Taina, J., 2010. How green is your software? International Conference of Software Business. Springer, (pp. 151–162).

Taina, J., Pohjalainen, P., 2009. In search for green metrics. Workshop on Software Research and Climate Change. Orlando, FL.

Tarwani, S., Chug, A., 2016. Sequencing of refactoring techniques by Greedy algorithm for maximizing maintainability. 2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI). IEEE, (pp. 1397–1403).

Tarwani, S., Chug, A., 2020. Assessment of optimum refactoring sequence to improve the software quality of object-oriented software. Journal of Information and Optimization Sciences, 41(6), 1433–1442.

Tewarie, P., van Dellen, E., Hillebrand, A., Stam, C.J., 2015. The minimum spanning tree: an unbiased method for brain network analysis. Neuroimage, 104, 177–188.

Tiwari, V., Malik, S., Wolfe, A., 1994. Compilation techniques for low energy: An overview. Proceedings of 1994 IEEE Symposium on Low Power Electronics. IEEE, (pp. 38–39).

Tokuda, L., Batory, D., 2001. Evolving object-oriented designs with refactorings. Automated Software Engineering, 8(1), 89–120.

Tolga, A.Ç., Turgut, Z.K., 2018. Sürdürülebilir ve Yenilenebilir Enerji Santrallerinin Bulanık TODIM Yöntemiyle Değerlendirilmesi. Alphanumeric Journal, 6(1), 49–68.

Triwijoyo, B.K., Gaol, F.L., Soewito, B., Warnars, H.L.H.S., 2017. Software reliability measurement base on failure intensity. 2017 3rd International Conference on Science in Information Technology (ICSITech). IEEE, (pp. 176–181).

Tunalı, V., 2011. Metin madenciliği için iyileştirilmiş bir kümeleme yapısının tasarımı ve uygulaması. Ph.D. thesis, Marmara Üniversitesi, 103s, İstanbul.

Turver, R.J., Munro, M., 1994. An early impact analysis technique for software maintenance. Journal of Software Maintenance: Research and Practice, 6(1), 35–52.

Umeyama, S., 1988. An eigendecomposition approach to weighted graph matching problems. IEEE transactions on pattern analysis and machine intelligence, 10(5), 695–703.

Ural, E., Umut, T., Feza, B., 2008. Nesneye Dayalı Yazılım Metrikleri ve Yazılım Kalitesi. Yazılım Kalitesi ve Yazılım Geliştirme Araçları Sempozyumu.

Vijayarani, S., Ilamathi, M.J., Nithya, M., vd., 2015. Preprocessing techniques for text mining-an overview. International Journal of Computer Science & Communication Networks, 5(1), 7–16.

Visa, A., 2001. Technology of text mining. International Workshop on Machine Learning and Data Mining in Pattern Recognition. Springer, (pp. 1–11).

Watson, A.H., Wallace, D.R., McCabe, T.J., 1996. Structured testing: A testing methodology using the cyclomatic complexity metric.

West, D.B., vd., 2001. Introduction to graph theory, Volume 2. Prentice hall Upper Saddle River.

Witten, I.H., 2004. Text Mining.

Woolson, R., 2007. Wilcoxon signed-rank test. Wiley encyclopedia of clinical trials, 1–3.

Xuan, J., Cornu, B., Martinez, M., Baudry, B., Seinturier, L., Monperrus, M., 2016. B-Refactoring: Automatic test code refactoring to improve dynamic analysis. Information and Software Technology, 76, 65–80.

Yılmaz, F., 2014. Enerji Verimliliği ve Karbon Ayak İzi.

Zeil, S.J., 2017. Integrated Development Environments. `https://www.cs.odu.edu/~zeil/cs350/f17/Public/IDEs/index.html`. Erişim Tarihi: 08.03.2021.

Zotos, K., Litke, A., Chatzigeorgiou, A., Nikolaidis, S., Stephanides, G., 2005. Energy complexity of software in embedded systems. arXiv preprint nlin/0505007.