KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

BASİT DOĞRU AKIM ELEKTRİK DEVRELERİ İÇİN E-ÖĞRENME DEVRE ANALİZ YAZILIMININ GELİŞTİRİLMESİ

YÜKSEK LİSANS TEZİ

Sultan ZAVRAK

TEMMUZ 2013
TRABZON

KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

BASİT DOĞRU AKIM ELEKTRİK DEVRELERİ İÇİN E-ÖĞRENME DEVRE ANALİZ YAZILIMININ GELİŞTİRİLMESİ

Bilgisayar Mühendisi Sultan ZAVRAK

Karadeniz Teknik Üniversitesi Fen Bilimleri Enstitüsünce
"BİLGİSAYAR YÜKSEK MÜHENDİSİ"
Unvanı Verilmesi İçin Kabul Edilen Tezdir.

Tezin Enstitüye Verildiği Tarih : 22.05.2013
Tezin Savunma Tarihi : 03.07.2013

Tez Danışmanı : Yrd. Doç. Dr. Hüseyin PEHLİVAN

Trabzon 2013

Karadeniz Teknik Üniversitesi Fen Bilimleri Enstitüsü

Bilgisayar Mühendisliği Anabilim Dalında

Sultan ZAVRAK tarafından hazırlanan

BASİT DOĞRU AKIM ELEKTRİK DEVRELERİ İÇİN E-ÖĞRENME DEVRE ANALİZ YAZILIMININ GELİŞTİRİLMESİ

başlıklı bu çalışma, Enstitü Yönetim Kurulunun 18 / 06 / 2013 gün ve 1510 / 01 sayılı kararıyla oluşturulan jüri tarafından yapılan sınavda

YÜKSEK LİSANS TEZİ

olarak kabul edilmiştir.

Jüri Üyeleri

Başkan : Prof. Dr. Sefa AKPINAR …...……

Üye : Yrd. Doç. Dr. Hüseyin PEHLİVAN …...……

Üye : Yrd. Doç. Dr. Bekir DİZDAROĞLU ……...……

Prof. Dr. Sadettin KORKMAZ

Enstitü Müdürü

ÖNSÖZ

Hızla gelişen günümüz teknolojisi elektronik ortamı ve bu ortamda çalıştırılan bilgisayar yazılımlarını öğretim faaliyetlerinin önemli bir bileşeni haline getirmiştir. Bu yazılımlar gerek eğiticilerin bilgiyi aktarmayı kolaylaştırmasına gerekse öğrencilerin öğrenme sürecinde bilgiyi daha iyi kavrayıp kalıcı bir şekilde anlamasına yardımcı olmaktadır.

Bu çalışmada, şu anda kullanılan elektriksel devre analiz ve simülasyon yazılımlarından farklı olarak basit doğru akım elektrik devrelerinin analizi yapılmakta, devre kanunlarına göre oluşturulan denklemleri ve ara hesaplama işlemlerini de içeren bir çıktı dokümanı üretilmektedir. Böylece verilen devre için hesaplama adımları gösterilerek çözüm sürecinin anlaşılabilirliği artırılmaktadır.

Bu çalışmada danışmanlığımı üstlenen değerli hocam Yrd. Doç. Dr. Hüseyin PEHLİVAN'a ilgi, alaka ve yardımlarından dolayı teşekkürü bir borç bilirim. Ayrıca her zaman yanımda olan aileme ve dostlarıma da destekleri için teşekkür ederim.

Sultan ZAVRAK

Trabzon 2013

TEZ BEYANNAMESİ

Yüksek Lisans Tezi olarak sunduğum "BASİT DOĞRU AKIM ELEKTRİK DEVRELERİ İÇİN E-ÖĞRENME DEVRE ANALİZ YAZILIMININ GELİŞTİRİLMESİ" başlıklı bu çalışmayı baştan sona kadar danışmanım Yrd. Doç. Dr. Hüseyin PEHLİVAN'ın sorumluluğunda tamamladığımı, örnekleri kendim topladığımı, deneyleri ilgili laboratuvarlarda yaptığımı, başka kaynaklardan aldığım bilgileri metinde ve kaynakçada eksiksiz olarak gösterdiğimi, çalışma sürecinde bilimsel araştırma ve etik kurallara uygun olarak davrandığımı ve aksinin ortaya çıkması durumunda her türlü yasal sonucu kabul ettiğimi beyan ederim. 03/07/2013

Sultan ZAVRAK

İÇİNDEKİLER

Sayfa No

Yüksel Lisans Tezi

ÖZET

BASİT DOĞRU AKIM ELEKTRİK DEVRELERİ İÇİN E-ÖĞRENME DEVRE ANALİZ YAZILIMININ GELİŞTİRİLMESİ

Sultan ZAVRAK

Karadeniz Teknik Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Ana Bilim Dalı
Danışman: Yrd. Doç. Dr. Hüseyin PEHLİVAN
2013, 83 Sayfa, 35 Ek Sayfa

Elektrik devreleri, elektrik ve elektronik mühendisliğindeki lisans düzeyinde birçok dersin temel konusunu oluşturur. Yükseköğrenimdeki çoğu öğrenci için böyle devrelerin analiz ve çözümünü öğrenmek zor bir süreçtir. Bu çalışmada, basit doğru akım elektrik devrelerinin analizini yapabilen ve bütün ara çözüm adımlarıyla birlikte öğretici dokümanlarını üretebilen bir eğitim programının tanıtımı yapılmaktadır. Devreler şu anda sadece dirençler, bağımsız akım ve gerilim kaynaklarından oluşmaktadır. Ayrıca, tanıtılan programa eşdeğer direncin hesaplanması ve elektrik devrelerinin otomatik üretimi gibi yararlı özellikler eklenmiştir.

Anahtar Kelimeler: Elektrik devreleri, graf analizi, e-öğrenme, devre girdi dili, otomatik devre üretimi.

Master Thesis

SUMMARY

IMPLEMENTATION OF E-LEARNING CIRCUIT ANALYSIS SOFTWARE FOR SIMPLE DIRECT CURRENT ELECTRICAL CIRCUITS

Sultan ZAVRAK

Karadeniz Technical University
The Graduate School of Natural and Applied Sciences
Computer Engineering Graduate Program
Supervisor: Assist. Prof. Dr. Hüseyin PEHLİVAN
2013, 83 Pages, 35 Pages Appendix

Electric circuits form the principal subject of many undergraduate courses in electrical and electronic engineering. For most higher education students, it is a difficult process to learn the analysis and solution of such circuits. In this study, we present the description of an education program that can analyze simple direct current (DC) electrical circuits and generate instructive documents with all their intermediate solution steps. The circuits currently comprise resistors, independent current and voltage sources only. Besides, the program is equipped with some useful features such as the calculation of equivalent resistance, and the automatic generation of electrical circuits.

Key Words: Electric circuits, graph analysis, e-learning, circuit input language, automatic circuit generation.

ŞEKİLLER DİZİNİ

Sayfa No

TABLOLAR DİZİNİ

Sayfa No

SEMBOLLER DİZİNİ

DC	Doğru Akım (Direct Current)
CFG	Bağlam Bağımsız Gramer (Context Free Grammar)
LL(k)	Soldan sağa ayrıştırma – Sola dayalı türetim – k kelime kontrolü (Left-to-Right Parsing – Leftmost Derivation – k lookahead)
BNF	Backus Normal Form
EBNF	Extented Backus Normal Form
BFS	Enine Arama (Breath-First Search)
KCL	Kirchhoff'un Akım Kanunu (Kirchhoff's Current Law)
KVL	Kirchhoff'un Gerilim Kanunu (Kirchhoff's Voltage Law)
WYSIWYG	What You See Is What You Get
XML	Extensible Markup Language
SPICE	Simulation Program with Integrated Emphasis
DGD	Devre Girdi Dili

1. GENEL BİLGİLER

1.1. Yorumlayıcılar

Yorumlayıcı (interpreter), bir kaynak dosyasını veya bir komutu çalıştırmaya yarayan programdır. Yorumlayıcı kaynak kodun başından itibaren her satır için çeviri işlemini gerçekleştirip ardından kodun çalıştırılmasını sağlar [1]. Yorumlama sonucunda herhangi bir çalıştırılabilir veya benzer bir dosya oluşturulmadığından her çalıştırmada kaynak metin üzerinden yorumlama işlemi yapılır. Ayrıca kaynak kodun çalışması anında kaynak programın yanında yorumlayıcının da belleğe yüklenmesi gereklidir.

Yorumlayıcının kaynak metni yorumlaması için bazı aşamaları takip etmesi gerekir. Bir dile uygun yazılmış kaynak kodun yorumlanabilmesi için yorumlayıcı ilk aşamada, kod yapısını ve içeriğini kontrol eder. Bu aşama analiz aşaması olarak adlandırılır. Şekil 1.1'de yorumlayıcının analiz aşaması gösterilmektedir.

Şekil 1.1. Yorumlayıcı aşamaları

Yorumlayıcılar analiz aşamasını gerçekleştirdikten sonra derleyiciler gibi sentez aşamasına geçmezler. Bu aşamadan sonra analiz edilmiş kaynak metnin çalıştırılması sağlarlar. Yorumlayıcılar için analiz süreci üç aşamadan oluşur. Bunlar;

- Kelimesel Analiz,
- Sözdizimsel Analiz,
- Anlamsal Analiz,

aşamalarıdır.

1.1.1. Kelimesel Analiz

Yorumlayıcı tasarımındaki ilk aşama kelimesel analiz aşamasıdır. Bu aşamada, bir dile ait kaynak metin içinde bulunan karakterler kümesi, anlamlı en kısa kelimeler (token) kümesine dönüştürülür.

Kelimesel çözümleyiciler, onlara girdi olarak verilen karakter dizisini, isimler dizisi, anahtar kelimeler ve noktalama işaretleri gibi daha önceden tanımlanmış yapılara çevirme işlemini gerçekleştirir. Bu aşamada, eğer gerekli tanımlamalar yapılmışsa, kaynak metinde bulunan açıklamalar ve özel karakterler gibi yorumlama işlemine dahil edilmeyecek yapılar dikkate alınmaz. Kelimesel analiz yapılırken kaynak metin içerisinde bulunan ve çözümleyici tarafından belirlenen dile uygun olmayan yapılar belirlenip hata mesajı üretilir. Kaynak metnin kelimesel analizi sonucu hata çıkması durumunda analiz işlemine devam edilmez ve sonraki aşamalara geçilmez.

Kelimesel analizin yapılabilmesi için bazı yararlı mekanizmalar geliştirilmiştir. Bu mekanizmalardan biri düzenli ifadelerdir. Düzenli ifadeler [2], verilen bir karakter topluluğunun içinden belirli karakter kümelerinin seçilmesini sağlayan bir mekanizmadır. Bu yapı kullanılarak belirli uzunluktaki kelimeler, sayılar ve noktalama işaretleri belirtilen şartlara göre uygun bir şekilde kolaylıkla eşleştirilebilir.

1.1.2. Sözdizimsel Analiz

Yorumlayıcı tasarımındaki ikinci aşama, sözdizimsel analiz işlemidir. Bu aşamada yapılan işlemler;

- Dilinin sözdizimsel kurallarının tanımlanması,
- Kaynak metnin, tanımlanan dilin sözdizimsel yapısına uyup uymadığını kontrol etmek ve varsa hataları belirlemek,
- Kelimesel analiz aşamasında üretilen kelimeler topluluğuna hiyerarşik bir yapı kazandırarak yeni bir veri yapısı üretmektir (Şekil 1.2.).

Bir dil, kelimeler kümesinden oluşmaktadır. Her bir kelime ise önceden belirlenmiş alfabede bulunan sembollerin sonlu dizisinden oluşmaktadır. Ayrıştırma işleminde kelimeler dizisi kaynak metnin, semboller dizisi kelimesel yapıların ve alfabe de kelimesel çözümleyicilerin geri döndürdüğü kelime türlerinin bir kümesidir.

Şekil 1.2. Sözdizimsel analiz

Bir dilin sözdizimsel yapısını ifade etmek için Bağlam Bağımsız Gramer (Context Free Grammar – CFG) mekanızması kullanılır. CFG'ler dört kavramı temel alır [3];

- Sonlu terminal dizisi V_t: Bu dizi kelimesel çözümleyiciler tarafından üretilen kelimelerden oluşur. Dil içinde, alfabedeki sembollerden oluşturulmuş kelimeleri ifade etmektedir. Bu ifadelerle geçiş sona erer.
- Sonlu non-terminal dizisi V_n: Üretim kuralının sol tarafında bulunan yapıyı ifade etmektedir. Terminal ya da non-terminal yapılara geçiş yapan ifadelerdir.
- Başlangıç sembolü S. Grameri başlatan non-terminali ifade etmektedir.
- Sonlu üretim (production) dizisi P: Terminal ve non-terminal yapılardan oluşan ve dilin gramer kurallarını tanımlayan yapıdır. Ayrıştırma ağaçlarının üretilmesinde bu yapılar kullanılır.

Üretim kurallarının formatı aşağıdaki biçimdedir;

SEMBOL ::= SEMBOL SEMBOL ... SEMBOL

Üretim kuralının sağ tarafında en az bir tane olmak şartı ile sınırsız sayıda terminal ya da non-terminal ifade bulunabilir [3].

1.1.2.1. Türetim, Ayrıştırma Ağaçları ve Belirsiz Gramer

Kaynak metnin ilgili dilin gramerine uygun olup olmadığını belirlemek için türetim işlemi gerçekleştirilir. Türetim işlemine, başlangıç sembolü ile başlanarak üretim kurallarına uygun bir şekilde eşleştirme yapılır.

Türetim işleminin farklı yöntemleri bulunmaktadır;

- Sola Dayalı Türetim: Bu yöntemde ilk olarak en soldaki non-terminal sembolü genişletilmeye çalışılır.

- Sağa Dayalı Türetim: Bu yöntemde ise ilk olarak en sağdaki non-terminal sembolü genişletilmeye çalışılır.

Ayrıştırma ağacı, bir kelime topluluğunun gramer kurallarına göre yapısal olarak parçalanıp ağaç şeklinde ifade edilmesidir. Ayrıştırma ağacında bulunan iç düğümler gramerde bulunan non-terminallere karşılık gelirken yaprak düğümler gramerin terminal yapılarına karşılık gelir.

Analiz sürecinde, ayrıştırma yönüne bağlı olarak, birden çok ayrıştırma ağacı ortaya çıkaran gramerler belirsiz gramer olarak nitelendirilir [4]. Yorumlama sırasında belirsiz gramerlerin olması sorunlara neden olmaktadır. Değerlendirilmelerin farklı ağaçlar üzerinden yapılması farklı sonuçların oluşmasına ve bu durumda kaynak metnin farklı anlamlandırılması bazı sorunlara sebep olmaktadır. Bundan dolayı diller için oluşturulan gramerlerdeki belirsizliklerin giderilmesi gerekir. Bu da verilen gramerdeki birleşme yönlerinin dikkate alınmasıyla tekrar düzenleme yapılarak sağlanır.

Ayrıştırma işlemi, birçok algoritma kullanılarak gerçekleştirilebilir. Bu algoritmalar genel olarak iki sınıfa ayrılabilir:

- Top – Down Ayrıştırma
    - Recursive Descent Ayrıştırma
    - LL Ayrıştırma
- Bottom – Up Ayrıştırma
    - LR Ayrıştırma

1.1.2.2. Recursive Descent Ayrıştırma

Bu ayrıştırma algoritmasında, tanımlanan gramerdeki her bir non-terminal için, non-terminalin kendisi tarafından üretilebilen cümleleri ayrıştırabilen bir alt program yazılır. Bu algoritma çok yaygın olmamasına rağmen, en önemli avantajı basit olması ve ayrıştırıcı üreteçlerinin kullanılmadığı durumlarda kolaylıkla oluşturulabilmesidir.

Recursive descent ayrıştırmada, tanımlanan gramerin her bir alt ifadesindeki ilk terminal sembolünün, bir sonraki adımda hangi kuralın kullanılacağını belirleyebilmesi için gerekli bilgiye sahip olması gerekir. Bu bilgiye sahip olunmazsa ayrıştırıcı çalışmaz.

Bu algoritmada, ayrıştırma tablosunu oluşturabilmek için FIRST ve FOLLOW kümeleri kullanılır. FIRST kümesi, terminaller ve non-terminallerden oluşan bir gramerdeki herhangi bir kuralın hangi terminal sembolü ile başlayacağını belirler. Bir gramerin aynı sol

tarafa sahip en az iki kuralının, aynı FIRST kümesine sahip olması o gramerin bu algoritma yardımıyla ayrıştırılamayacağını gösterir. FOLLOW kümesi ise non-terminali takip eden ilk terminal olarak tanımlanır. Dolayısıyla yalnızca non-terminal ifadeler için hesaplama yapılabilir. Tablo 1.1'de aşağıda verilen gramer için hesaplanmış FIRST ve FOLLOW kümeleri bulunmaktadır.

Tablo 1.1. Bir gramer ve FIRST ve FOLLOW kümeleri

<table>
<tr><th>Gramer</th><th>FIRST ve FOLLOW kümeleri</th></tr>
<tr><td>Z → d
Z → X Y Z
Y →
Y → c
X → Y
X → a</td><td>
<table>
<tr><th></th><th>FIRST</th><th>FOLLOW</th></tr>
<tr><td>X</td><td>a c</td><td>a c d</td></tr>
<tr><td>Y</td><td>c</td><td>a c d</td></tr>
<tr><td>Z</td><td>a c d</td><td></td></tr>
</table>
</td></tr>
</table>

1.1.2.3. LL Ayrıştırma

Gramerin ayrıştırma tablosunda tekrarlama içermeyen yapılar LL(k) olarak isimlendirilir. LL algoritması, ayrıştırma işlemini, soldan sağa doğru ve sola dayalı türetim ile gerçekleştirmektedir. Bu algoritma ile ayrıştırılabilen gramerler LL gramer olarak adlandırılır. LL(k) kullanılması k tane kelimeye bakılıp karar verileceğini anlamını taşımaktadır.

LL Ayrıştırma;

- Girdi cümlesini saklayacak bir arabelleğe,
- Ayrıştırılmış terminal ve non-terminal yapıları saklamak için bir yığına,
- Girdi ve yığında bulunan verilere dayanarak hangi gramer kuralının uygulandığını gösteren bir ayrıştırma tablosuna sahiptir.

Ayrıştırma tablosunda bulunan kurallardan, giriş cümlesinde bulunan sembollere en çok uyanını bularak yığına atar. Bu işlem girilen cümlenin sonuna gelene kadar yinelemeli bir şekilde devam ettirilir.

Bir gramerin LL(k) olabilmesi için bazı özelliklere sahip olması gerekir. Bunlar:

- Gramer soldan yinelemeli olmamalı; Soldan yinelemeden kurtulmak için, soldan yineleme içeren kural sağdan yineleme kullanılarak tekrar yazılır.

- Sol faktörleme (Left Factoring); İki kuralın aynı sembol ile başlaması LL(k) grameri için diğer bir sorundur. Bunu ortadan kaldırmak için sol faktörleme yapılır.

1.1.2.4. LR Ayrıştırma

LL ayrıştırma, uygulanan kuralı tahmin yoluyla tespit ettiği için zayıf bir ayrıştırma tekniğidir. Bunun yerine daha güçlü bir teknik olan LR ayrıştırma geliştirilmiştir.

LR ayrıştırma girilen cümleyi soldan sağa okur, ancak sağa dayalı türetim üretir. LR ayrıştırma:

- Girdi cümlesini saklayacak bir arabelleğe,
- Hangi durumda olunduğunu belirten bir yığına,
- Yeni durumun ne olacağını belirleyen goto tablosuna,
- Girdi cümlesinde o anki terminale ve duruma göre uygulanacak gramer kurallarını tutan action tablosuna sahiptir.

LR ayrıştırma, tablo tabanlı bir yöntemdir. Bu yöntemin action ve goto olmak üzere iki bileşeni vardır. Action tablosu verilen bir ayrıştırıcı durumuna ve sonraki kelime için yapılması gereken işlemi belirler. Goto tablosu ise indirgeme işlemi yapıldıktan sonra yığın üzerine hangi durumun getirileceğini belirler.

Ayrıştırma algoritması aşağıdaki adımlardan oluşur;

- Başlangıç yapılandırması gerçekleştirilir.
- Verilen duruma ve giriş cümlesinde o anki terminal değerlerine bağlı olarak action tablosundan eşleştirme yapılır. Burada dört durum bulunmaktadır;
    - Kaydırma sn:
        - Terminal değeri giriş cümlesinden silinir,
        - n durumu yığına itilir ve şu anki durum olarak setlenir.
    - İndirgeme rk:
        - k numaralı kural çıktı olarak verilir,
        - m kuralının sağ tarafında bulunan her bir sembol yığından silinir,
        - m kuralının sol tarafı yeni durum olur ve yığının başına eklenir.
    - Kabul etme: Giriş cümlesi kabul edilir.
    - Hata: Hata olduğu tespit edilir ve raporlanır.
- Yukarıdaki adım hata meydana gelene kadar ya da giriş cümlesi kabul edilene kadar sürdürülür.

LR ayrıştırmanın birçok avantajı bulunmaktadır;

- Birçok programlama dili LR ayrıştırmanın türevleri kullanılarak ayrıştırılabilir,
- LR ayrıştırıcılar etkin bir biçimde gerçeklenebilir,
- Diğer ayrıştırıcılara göre sözdizimsel hataları daha kısa zamanda tespit eder ve raporlar.

LR ayrıştırmanın tek dezavantajı ise elle üretilemeyecek kadar zor bir yöntemdir. Bu amaçla bu işi gerçekleştiren ayrıştırıcı üreteçleri kullanılmaktadır.

1.1.3. Anlamsal Analiz

Yorumlayıcı tasarımının üçüncü ve son aşaması anlamsal analiz işlemidir. Anlamsal analiz aşamasında;

- Yorumlayıcı değişken tanımlamalarını, kullanımları ile ilişkilendirir.
- İfadelerin tip bilgilerinin doğruluğunu kontrol eder.
- Sözdizim ağacını inceleyip yorumlama aşamasına geçmeden önce son kez kodu kontrol eder.

Anlamsal analiz iki temel aşamadan oluşmaktadır. Bu aşamalar;

- Sembol tablosunun oluşturulması,
- Tip kontrolünün gerçekleştirilmesidir.

Kaynak metinde tip bildirimleri, değişken ve fonksiyon tanımlamaları yapıldığında bu tanımlayıcı bilgileri sembol tablosuna yerleştirilir. Kaynak metnin sonraki kısımlarında kullanımına rastlanan tanımlayıcıların gerekli bilgilerine sembol tablosundan bakılıp karar verilir.

Kaynak metin içinde kullanılan tanımlayıcıların tip bilgileri sembol tablosuna eklendikten sonra, bu tanımlayıcıların uygun şekilde kullanılıp kullanılmadığının belirlenmesi tip kontrolü yapılarak sağlanır. Tip kontrolünün gerçekleştirilebilmesi için sembol tablosu program içerisinde tanımlanmış bazı tür bilgilerini içermelidir.

- Değişken isimleri ve formal parametreler tip bilgileri ile ilişkilendirilmeli,
- Metot isimleri, bu metodun aldığı parametreler, sonuç tipi ve yerel değişkenleri ile ilişkilendirilmeli,
- Eğer nesneye dayalı bir dil ise sınıflar da, içinde tanımlanmış değişken ve metotlarla ilişkilendirilmelidir.

Anlamsal analiz ile kod analizi tamamlandıktan sonra kaynak kod yorumlanmaya hazır hale gelir. Yorumlama işleminde de anlamsal analizde olduğu gibi sözdizim ağacından faydalanılır. Bu ağacın düğümlerinde ilerleyerek kodun yorumlanması işlemi satır satır gerçekleştirilir. Bu aşamada, anlamsal analizde kullanılmış sembol tablosunda artık tanımlayıcı özelliklerin yerine değişkenlerin değerleri tutulmaktadır. Bu işlem için ya sembol tablosu silinip yerine yeni bir tablo oluşturulur ya da sembol tablosunda bulunan alanlar güncellenir.

Yorumlama esnasında, değişkenlerin değerleri değer tablosuna eklenir. İstenilen değerler gerektiğinde ya tablodan çekilir ya da tabloda güncelleme yapılır.

1.2. Kod Üretim Araçları

Kod üretim araçları, gerek kelimesel analiz gerekse ayrıştırma işlemlerini otomatikleştirir. Bu şekilde tanımlanan dile uygun kelimesel analiz ve sözdizimsel analiz yapacak olan kodlar uygun gelen programlama dillerinde üretilir. Bu araçlardan bazıları sadece kelimesel analiz işlemi yapan kodu üretir. Bu araçlara örnek olarak lex [5], Jlex [6] ve Flex [7] verilebilir. Kod üretim araçlarının bazıları hem kelimesel analizi hem de ayrıştırma işlemini yapmaktadır. Bu araçlardan en çok kullanılanları ürettikleri kodların programlama dilleriyle birlikte Tablo 1.2'de gösterilmektedir.

Bu araçlar gramerlerle tanımlanan yapıları, kod içinde belirleyebilmek için otomatik kod üretirler. Üreteçler Java, C, C++ programlama dillerine ait kodları üretmelerine rağmen hepsinin kendine özgü sözdizim yapıları bulunmaktadır. Örneğin Java kodu üreten JavaCC üreteci, Java'nın sözdiziminden bağımsızdır.

Tablo 1.2. Kod üretim araçları

Üreteç	**Ayrıştırma Algoritması**	**Programlama Dili**
ANTLR	LL(k)	C#, Java, Phyton
JAVACC	LL(k)	Java
SABLECC	LALR	Java, C, C++, C#, Phyton
YACC	LALR	C

Üreteçler yorumlayıcı geliştirme, derleyici geliştirme, doğal dil işleme gibi birçok alanda kullanılmaktadır. Ürettikleri otomatik kod sayesinde programcıya büyük kolaylık sağlamaktadır.

1.2.1. JavaCC

JavaCC [10], Java programlama dilinde uygulama geliştirmeye imkân sağlayan kelimesel çözümleyici üreteci ve aynı zamanda ayrıştırıcı üretecidir [8]. Bağlam bağımsız gramere bağlı olarak Java dilinde recursive descent ayrıştırma kodu üretir.

JavaCC ile uygulama geliştirme dört adımdan oluşmaktadır [9]:

- EBNF'de kullanılacak dilin grameri hazırlanır.
- EBNF dosyası, JavaCC gramer dosyasına çevrilir.
- JavaCC gramer dosyasından uygulama için Java koduna dönüşüm gerçekleştirilir.
- Oluşturulan Java dosyası geliştirilecek uygulama içinde kullanılır.

1.2.1.1. Gramerin Belirlenmesi

Bir gramer genellikle Backus Normal Form (BNF) notasyonu ile tanımlanır. BNF notasyonunda non-terminal ve terminal'lerin çok farklı gösterimleri olmakla birlikte, aşağıdaki gramer tanımlamalarında terminal yapılar '<' ve '>' işaretleri arasına alınarak gösterilmektedir.

Tablo 1.3. EBNF meta karakterleri ve anlamları

Simge	**Anlamı**	**Örnek**	**Veriler**
*	Veriyi herhangi sayıda tekrarla	a*	{Ø, a, aa, aaa, ...}
+	Veriyi en az bir defa tekrarla	b+	{b, bb, bbb, ...}
|	Veri alternatifleri oluştur	a | b	{a, b}
?	Veriyi en fazla bir defa tekrarla	c?	{Ø, c}
()	Veriyi grupla	(a|b)	{a, b}
[]	Bir veri aralığı oluştur	[a-c]	{a, b, c}

Gramerler BNF notasyonuna meta karakterler eklenerek geliştirilen Extended Backus Normal Form (EBNF) notasyonunda da tanımlanabilir. Tablo 1.3'te bazı meta karakterler ve anlamları listelenmiştir.

Aşağıda örnek bir gramerin [9] BNF özellik dosyası gösterilmiştir. Terminal yapılar '<' ve '>' işaretleri arasında bulunmaktadır.

```
Expression ::=  AndExpression ( <OR> AndExpression )*
AndExpression ::=
      EqComparisonExpression ( <AND> EqComparisonExpression )*
EqComparisonExpression  ::=
      ComparisonExpression ( ( <EQ> | <NE> ) ComparisonExpression )*
ComparisonExpression ::=
      AdditiveExpression ( ( <LT> | <GT> | <LE> | <GE>) AdditiveExpression )*
AdditiveExpression ::=
    MultiplicativeExpression ( ( <PLUS> | <MINUS> )MultiplicativeExpression )*
MultiplicativeExpression ::= ArithmeticUnaryExpression
    ( ( <MULT> | <SLASH> ) ArithmeticUnaryExpression )*
ArithmeticUnaryExpression ::= (<PLUS> | <MINUS> )
    ArithmeticUnaryExpression | BooleanUnaryExpression
BooleanUnaryExpression ::= (<TILDE> | <EXCL>) ArithmeticUnaryExpression
                                | PrimitiveExpression
PrimitiveExpression ::= Literal | Function | <IDENTIFIER>
                            | <LPAREN> Expression <RPAREN>
Function ::= <IDENTIFIER> Arguments
Literal ::= <INTEGER_LITERAL> | <FLOATING_POINT_LITERAL>
              | <STRING_LITERAL> | BooleanLiteral
BooleanLiteral ::= <TRUE> | <FALSE>
Arguments ::= <LPAREN> ( ArgumentList )? <RPAREN>
ArgumentList ::= Expression ( <COMMA> Expression )*
```

1.2.1.2. JavaCC Dosyasına Dönüşüm

EBNF dosyası uygun şekilde hazırlandıktan sonra JavaCC özellik dosyasına kolayca geçiş yapılabilir. JavaCC özellik dosyası dört ayrı bloktan oluşmaktadır. Birinci blok özellikler bloğudur ve bu blokta JavaCC'nin gramer dosyasında nasıl işlem yapacağını belirleyen özelliklerden yer alır. Örneğin;

```
options{
 LOOKAHEAD = 2;
}
```

özellik kodu ile birlikte ayrıştırma esnasında karar verebilmek için en az iki kelime gerektiği belirtilir.

İkinci blok, temel sınıf tanımlama bloğudur. Burada belirtilen sınıf ismi ile birlikte ayrıştırıcı sınıfının ismi aynıdır. Ayrıca bu blokta yazılan kodlar değiştirilmeden oluşturulacak Java sınıfının içinde yer alır. Aşağıda bu bloğa örnek bir yapı verilmiştir. Örnekte [9] PARSER_BEGIN ve PARSER_END arasında kalan ve temel sınıf özelliklerinin bulunacağı yer sınıf tanımlama bloğudur.

```
PARSER_BEGIN(ExpressionEvaluator) package ieee;
import java.util.*; import java.io.*;
public class ExpressionEvaluator {

public static void main(String[] args) throws ParseException {
        ...
        }
        private static Stack stack = new java.util.Stack();
        private static Hashtable state = null;
        public static boolean popBoolean() throws ClassCastException {
                Boolean a = (Boolean) stack.pop();
                if (a == null) return false; else return a.booleanValue();
        }
        .....
```

```
}
PARSER_END(ExpressionEvaluator)
```

Üçüncü blok ise kelimesel özelliklerin yer aldığı kısımdır. Bu blokta terminal değerler ve karakter ya da karakter dizisi şeklinde tanımlamalar yer alır. Burada tanımlanan terminaller JavaCC'ye ayrıştırma işleminde bulunması gereken anlamlı en kısa kelimelerin neler olduğunu ifade eder. Aşağıda kelimesel özellikler bloğuna örnek [9] gösterilmiştir.

```
TOKEN :
{  < ASSIGN: "=" > | < GT: ">" > | < LT: "<" > | < EXCL: "!" >
  | < TILDE: "~" > | < EQ: "==" > | < LE: "<=" >| < GE: ">=" >
  | < NE: "!=" > | < OR: "||" > | < AND: "&&" > | < PLUS: "+" >
  | < MINUS: "-" > | < MULT: "*" > | < SLASH: "/" >
}
TOKEN :
{   < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
  | < #LETTER: ["\u0041"-"\u005a", "\u005f", "\u0061"-"\u007a"] >
  | < #DIGIT:["\u0030"-"\u0039"] >
}
```

Son blok ise sözdizimsel kuralların yer aldığı kısımdır ve bu kısımda genişletilmiş Java metotları bulunur. Bu blok içerisine gramerde bulunan her bir kural için Java metotlarına benzer kodlar yazılır. Ayrıca blok içinde Java kodları gerek duyulduğu yere, küme parantezi içine alınarak yazılabilir. JavaCC gramer içinde tanımlanmış her bir kural için Java'da bir metot üretir. Metodun adı kuralın başında bulunan non-terminal ile aynı isimdedir. Metot isminden hemen sonra, JavaCC'nin küme parantezleri içinde tanımlamalara izin verdiği bir blok gelir. Bu blok içinde gerçekleştirilen tanımlamalar metot içinde geneldir. Bu yapının ardından non-terminal, terminal ya da non-terminallerle EBNF şeklinde tanımlanır. Tanımlamanın bu kısmında ihtiyaç duyuluyorsa yine Java metotları yazılabilir.

Aşağıda sözdizimsel kuralların tanımlandığı bloğa örnek [9] verilmektedir:

```
void Expression() :{}
{       AndExpression()    (   <OR> AndExpression()
```

```
{
try {
        boolean a = popBoolean();  boolean b = popBoolean();
        stack.push(new Boolean(b || a));
} catch (ClassCastException ex) {
        System.out.println("Non boolean operand supplied for <OR> operator");
}
}  )*
}
throw new ParseException();
```

AndExpression ifadesinden sonra yazılan Java kodları, her defasında programcı tarafından daha önceden tanımlanmış bir yığın içinden iki değer çekip bunların *OR* işlemine tabi tutulmasını sağlamaktadır.

Yukarıda örnek verilmiş kod parçası sadece tek bir kurala aittir ve diğer kurallar da buna benzer şekilde oluşturulmak zorundadır.

1.3. Graf Kavramları

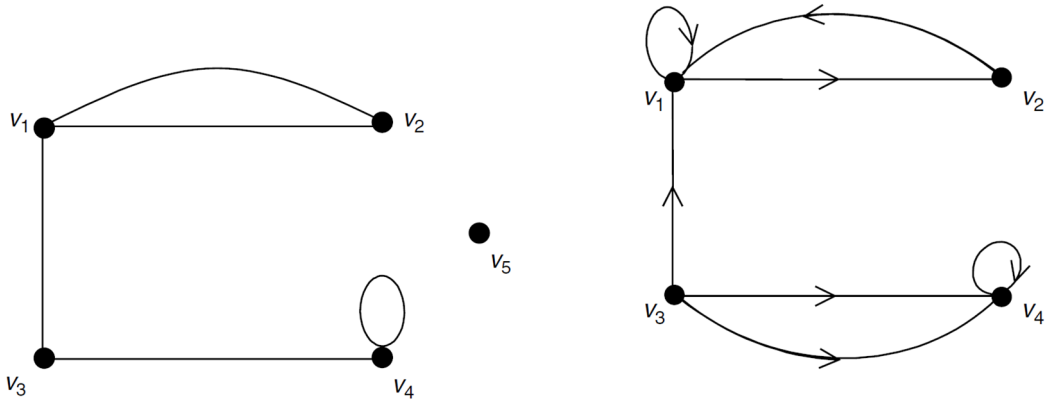
1.3.1. Temel Tanımlar

Bir $G=(V, E)$ grafı iki kümeden oluşur. Bunlar; $V=\{v_1,v_2,...v_n\}$ düğümler (nodes veya vertices) sonlu kümesi ve $E=\{e_1,e_2,...e_m\}$ kenarlar (edges) sonlu kümesidir. Eğer G'nin kenarları düğüm çiftlerinin sıralanmasıyla belirlenmişse yönlü (directed veya oriented) graf olarak adlandırılır. Diğer durumlarda ise yönsüz (undirected veya unoriented) graf olarak adlandırılır.

Graflar, geometrik çizimlerle gösterilebilir. Bu çizimlerde, her bir düğüm nokta veya çemberle (circle) ve her bir kenar ise o kenarla ilişkisi olan düğümleri birleştiren bir eğri veya çizgi ile gösterilir.

Yönlü graflarda, her bir kenara bir yön (direction veya orientation) verilir. Eğer bir kenar, (v_i, v_j) sıralı çiftiyle ilişkilendirilmişse, o zaman bu kenarın yönü v_i'den v_j'ye doğrudur. Eğer bir e kenarı v_i ve v_j düğümlerini bağlıyorsa o zaman bu ilişki $e=(v_i,v_j)$

şeklinde gösterilir. Yönlü bir grafta (v_i, v_j), v_i'den v_j'ye bir kenar olduğunu gösterir. Yönsüz (undirected) bir graf ve yönlü bir graf Şekil 1.3'te gösterilmektedir.



Şekil 1.3 Grafın geometrik çizimleri (a) Yönsüz graf (b) Yönlü graf

Açıkça belirtilmediği sürece, graf terimi yönlü bir grafa da yönsüz bir grafa da işaret edebilir. Bir kenarla ilişkilendirilmiş v_i ve v_j düğümleri kenarın uç noktaları (end points) olarak adlandırılır. Aynı uç nokta çiftine sahip kenarlar paralel kenar olarak adlandırılır. Yönlü bir grafta paralel kenarlar, v_i'den v_j'ye veya v_j'den v_i'ye aynı yöne işaret eder. Örneğin, Şekil 1.3(a)'da v_1 ve v_2 düğümlerini bağlayan kenarlar paralel kenarlardır. Şekil 1.3(b)'deki yönlü grafta v_3 ve v_4 düğümlerini bağlayan kenarlar paralel kenarlardır. Fakat v_1 ve v_2'yi bağlayan kenarlar paralel değildir.

Eğer bir kenarın uç düğümleri birbirinden farklı değilse, o kenar döngü (self-loop) olarak adlandırılır. Şekil 1.3(a)'daki grafta bir tane ve Şekil 1.3(b)'dekinde ise iki tane döngü (self loop) vardır. Yönlü bir grafta (v_i, v_j) kenarı v_i'den dışa doğru ilişkili (incident out of) ve v_j'ye doğru ilişkili (incident into) ilişkilidir denir. v_i ve v_j düğümleri eğer bir kenar tarafından bağlanıyorsa bunlar birbirleriyle komşudur (adjacent) denir.

Bir v_i düğümüyle ilişkili kenar sayısı v_i'nin derecesi olarak adlandırılır ve $d(v_i)$ ile gösterilir. Yönlü bir grafta $d_{in}(v_i)$, v_i yönü düğümüne doğru olan kenar sayısını gösterir. $d_{out}(v_i)$ ise yönü v_i'den dışa doğru olan kenar sayısını gösterir. Eğer $d(v_i) = 0$ ise v_i'ye

izole (isolated) düğüm denir. Eğer $d(v_i)=1$ ise v_i'ye asılı (pendant) düğüm denir. Bir v_i düğümündeki döngü (self loop), $d(v_i)$ hesaplanırken iki defa sayılır.

Şekil 1.3(a)'daki graf örnek olarak ele alınırsa $d(v_1)=3$, $d(v_4)=3$ ve v_5 izole düğümdür. Şekil 1.3(b)'deki yönlü grafta ise $d_{in}(v_1)=3$ ve $d_{out}(v_1)=2$'dir.

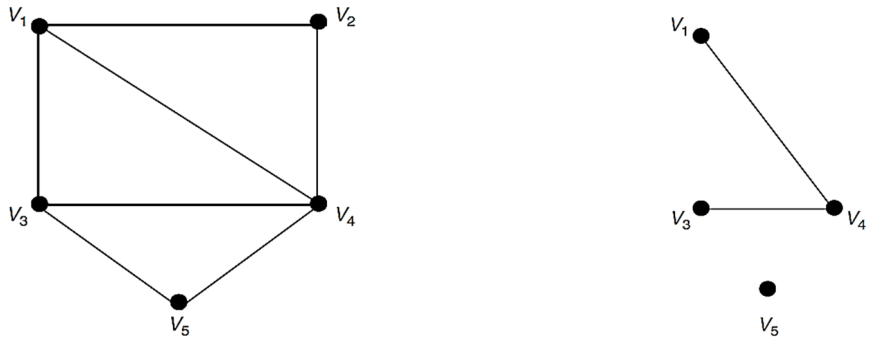
Yönlü bir grafta her v_i düğümü için $d(v_i)=\mathrm{d}_{in}(v_i)+\mathrm{d}_{out}(v_i)$'dir.

1.3.2. Temel Teoremler

Teorem 1: Bir G grafının kenar sayısı *m* olsun. O zaman G grafının düğümlerinin dereceleri toplamı $2m$'dir(1) ve m kenarlı yönlü bir grafta düğümlere giren-dereceler(in-degree) ile çıkan-derecelerin(out-degree) toplamının her ikisi de m'ye eşittir.

Teorem 2: Herhangi bir graftaki tek dereceli düğümler sayısı çifttir. $G=(V,E)$ grafını ele alalım. $G'=(V',E')$ grafı eğer $V'\in V$ ve $E'\in E$ ise G'nin alt grafıdır. Bir G grafı ve G'nin alt grafı Şekil 1.4'te verilmektedir.

Bir G grafında v_i ve v_j düğümlerini bağlayan bir P yolu (path), v_i ve v_j farklı olmak üzere, v_i ve v_j hariç, v_i'den başlayıp v_j'de sonlanan tüm kenar ve düğümlerin alternatif dizilimleridir. Yönlü bir grafta v_i ve v_j düğümlerini bağlayan P yolu, P'deki tüm kenarlar aynıysa yönlü yol (directed path) olarak adlandırılır.

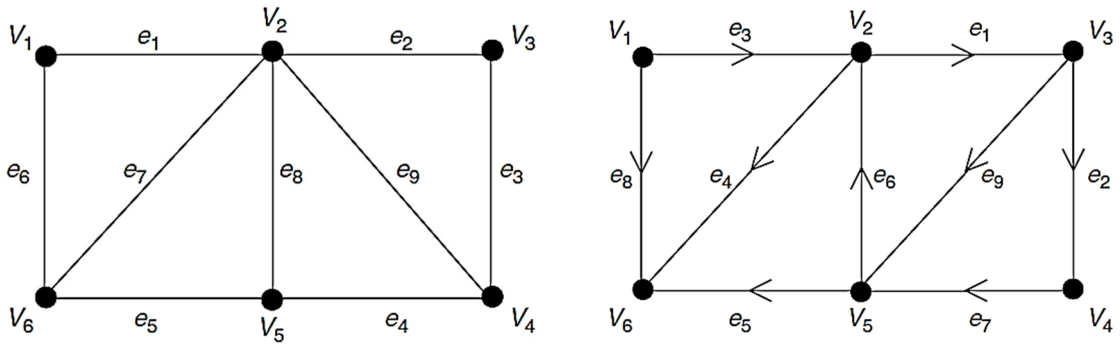


(a) (b)

Şekil 1.4. Graflar ve altgraflar (a) *G* grafı (b) *G*'nin altgrafı

Eğer bir yol aynı düğümde başlayıp bitiyorsa çevrim (circuit) olarak adlandırılır. Yönlü bir grafta aynı yönlü tüm kenarlardan oluşan çevrim (circuit), yönlü çevrim (directed circuit) olarak adlandırılır. Yollar ve çevrimler, onları oluşturan kenarlar kullanılarak gösterilir.

Örneğin, Şekil 1.5'teki yönsüz grafta v_1 ve v_5'i bağlayan bir P yolu $P: e_1, e_2, e_3, e_4$ ve $C: e_1, e_2, e_3, e_4, e_5, e_6$ ise bir çevrimdir. Şekil 1.5(b)'deki yönlü grafta $P: e_1, e_2, e_7, e_5$ bir yönlü yol ve $C: e_1, e_2, e_7, e_6$ ise yönlü bir çevrimdir. Yönlü graftaki $C: e_7, e_5, e_4, e_1, e_1$ çevriminin yönlü bir çevrim olmadığı açıkça görülmektedir.



Şekil 1.5. Yolları gösteren graflar (a) Yönsüz bir graf (b) Yönlü bir graf

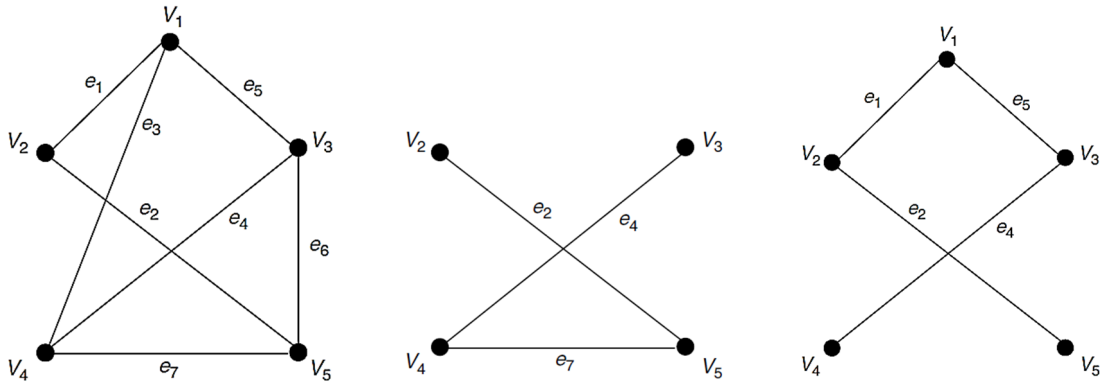
Bir graf, eğer graftaki tüm düğüm çiftleri arasında bir yol varsa bağlantılıdır (connected), aksi halde bağlantılı değildir denir. Örneğin Şekil 1.4(a)'daki graf bir bağlantılı graftır. Şekil 1.4(b)'deki ise bağlantılı değildir.

Bir ağaç (tree), bağlantılı ve hiçbir çevrimi olmayan bir graftır. G bağlantılı grafını ele alalım. Eğer G 'nin alt grafı bir ağaç ve bu ağaç G 'nin tüm düğümlerini içeriyorsa, o zaman bu ağaç G 'nin bir açılım ağacıdır (spanning tree).

Şekil 1.6(a)'daki grafın bir ağacı ve açılım ağacı sırasıyla Şekil 1.6(b) ve Şekil 1.6(c)'de gösterilmektedir. Bir T açılım ağacının kenarları, T 'nin dalları (branch) olarak adlandırılır. G bağlantılı grafının verilen bir T açılım ağacı için T 'de bulunmayan kenarların oluşturduğu G 'nin alt grafı, T 'ye göre yardımcı açılım ağacı (cospanning tree) olarak adlandırılır.

Örneğin Şekil 1.6(c)'deki T açılım ağacına göre yardımcı açılım ağacı "e_3, e_6, e_7" kenarlarından oluşur. Yardımcı açılım ağacının kenarlarına kiriş (chord) denir.

Bir ağaçtaki iki düğümün tam olarak bir yolla bağlandığı söylenebilir. Herhangi bir kenarın ağaçtan çıkarılması bağlantısız grafla sonuçlanacağından bir ağaç minimum şekilde bağlantılıdır.



(a) (b) (c)

Şekil 1.6. *G*'nin bir açılım ağacı (a) Bir *G* grafı (b) *G* grafının bir ağacı (c) *G*'nin bir açılım ağacı

Teorem 3: n düğümlü bir ağacın $n-1$ kenarı vardır. Eğer bir *G* bağlantılı grafı n düğüme ve m kenara sahipse ρ rankı (1.1)'deki gibi ve μ geçersizliği (nullity) ise (1.2)'deki gibi tanımlanır.

$$\rho(G) = n - 1 \tag{1.1}$$

$$\mu(G) = m - n + 1 \tag{1.2}$$

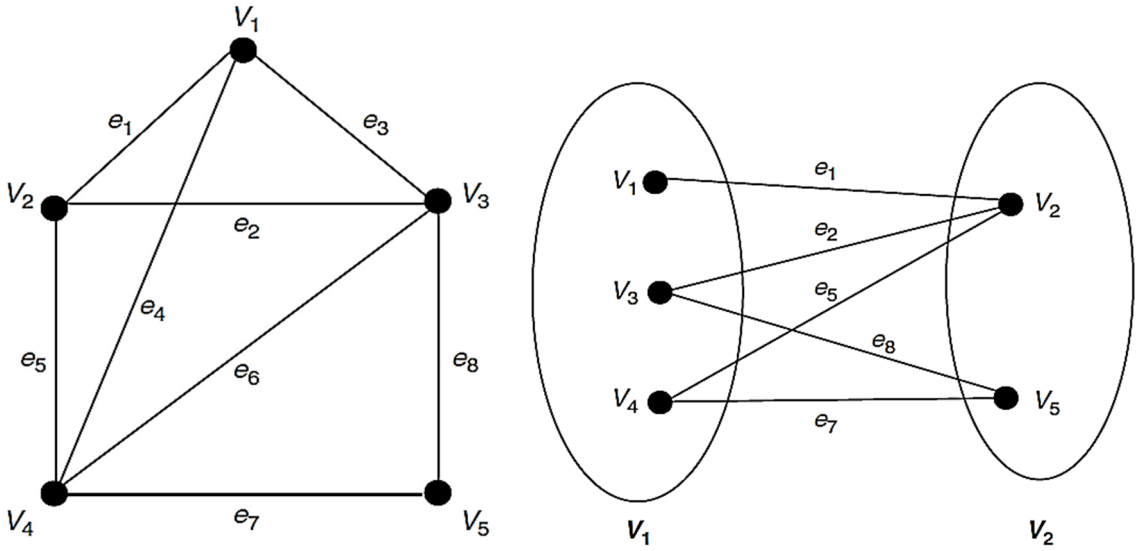
Eğer G bağlantılı ise G'nin herhangi bir açılım ağacı $\rho = n-1$ dala ve $\mu = m-n+1$ kirişe (chord) sahiptir.

1.3.3. Kesiler, Çevrimler ve Dikgenlik

Düğüm sayısı n ve kenar sayısı m olan bir $G=(V,E)$ grafını ele alın. $V = V_1 \cup V_2$ olsun. V_1 ve V_2, V'nin karşılık iki ayrı boş olmayan kümesi olsun. Bundan dolayı V_2, V'de V_1'in tümleyenidir. Bir uç düğümü V_1'de ve diğer uç düğümü V_2'de olan tüm kenarlar

kümesi G 'nin kesisi (cut) olarak adlandırılır ve $<V_1,V_2>$ şeklinde gösterilir. Örnek olarak Şekil 1.7'de bir G grafı ve onun bir kesisi olan $<V_1,V_2>$ gösterilmektedir.

Bir kesideki kenarlar kaldırıldıktan sonra oluşacak G grafı bağlantılı olmayacaktır. Bir G bağlantılı grafının S kesi kümesi (cutset), S 'nin kaldırılması sonucu G'yi bağlantısız yapan G'nin minimum kenarlar kümesidir. Bundan dolayı bir kesi kümesi aynı zamanda bir kesidir. Bir kesi kümesinin minimalite özelliği, bir kesi kümesinin alt kümelerinin bir kesi kümesi olmadığına işaret eder.



(a) (b)

Şekil 1.7. Bir graf ve onun bir kesisi (a) Bir G grafı (b) G'nin bir $<V_1,V_2>$ kesisi

G bağlantılı grafının T açılım ağacını ele alalım. b, T 'nin bir dalı olsun. b 'nin kaldırılması T 'yi T_1 ve T_2 gibi iki ağaca ayırır. V_1 ve V_2 sırasıyla T_1 ve T_2 'nin düğüm kümesi olsun. V_1 ve V_2 birlikte, G'nin tüm düğüm kümesini içerir. $<V_1,V_2>$ kesisinin, G'nin bir kesi kümesi olduğunu doğrulamak mümkündür ve bu kesi, T 'nin b dalına göre G'nin temel kesi kümesi (fundamental cutset) olarak adlandırılır. Bundan dolayı verilen bir G grafı ve G'nin T açılım ağacı ele alınırsa, T 'nin her bir dalı için $n-1$ tane kesi kümesi oluşturulabilir. Örnek olarak, Şekil 1.7'deki graf için, $T=[e_1,e_2,e_6,e_8]$ açılım ağacına karşılık gelen temel kesi kümeleri aşağıdaki gibi verilebilir.

Dal $e_1 : (e_1, e_3, e_4)$

Dal $e_2 : (e_2, e_3, e_4, e_5)$

Dal $e_6 : (e_6, e_4, e_5, e_7)$

Dal $e_8 : (e_8, e_7)$

b dalına karşılık gelen temel kesi kümesi b 'yi içerir. Dahası b dalının, T 'ye göre diğer temel kesi kümelerinin herhangi birinde b yer almaz.

T, G'nin bir açılım ağacı olsun. T'nin herhangi iki düğümü arasında tam olarak tek bir yol (path) olduğundan, c kirişinin T'ye eklenmesi benzersiz bir çevrim oluşturur. Bu çevrim, T'nin c kirişine karşılık gelen G'nin temel çevrimi (fundamental circuit) olarak adlandırılır. c kirişine karşılık gelen temel çevrim c'yi içerir ve T'nin diğer herhangi bir temel çevriminde c yer almaz.

Örnek olarak Şekil 1.7'de gösterilen grafın $T = (e_1, e_2, e_6, e_8)$ açılım ağacına karşılık gelen temel çevrimler kümesi aşağıdaki gibidir.

Kiriş $e_3 : (e_3, e_1, e_2)$

Kiriş $e_4 : (e_4, e_1, e_2, e_6)$

Kiriş $e_5 : (e_5, e_2, e_6)$

Kiriş $e_7 : (e_7, e_8, e_6)$

1.3.4. Graf Matrisleri

Graf matrisleri, graf algoritmalarında yaygın kullanılan komşuluk matrisi ve özellikle elektrik ağları analizinde kullanılan ilişki (incidence), çevrim (circuit) ve kesi (cut) matrislerinden oluşmaktadır.

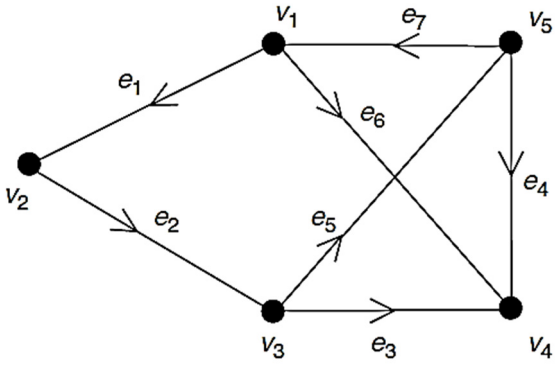
1.3.4.1. İlişki Matrisi

n düğümlü ve m kenarlı döngüsü (self-loop) olmayan bir bağlantılı ve yönlü G grafı olsun. G'nin $A_c = [a_{ij}]$ tüm-düğüm ilişki matrisi (all-vertex incidence matrix), her biri bir

düğüme karşılık gelen n satıra ve her biri bir kenara karşılık gelen m sütuna sahiptir. A_c'nin a_{ij} elemanı aşağıdaki gibi tanımlanır.

$$a_{ij} = \begin{cases} 1, & \text{kenar düğümden dışa doğru yönelmişse} \\ -1, & \text{kenar düğüme doğru yönelmişse} \\ 0, & \text{kenar düğümle ilişkili değilse} \end{cases}$$

Örnek olarak bir graf ve onun A_c matrisi Şekil 1.8'de gösterilmektedir.



	e_1	e_2	e_3	e_4	e_5	e_6	e_7
v_1	1	0	0	0	0	1	−1
v_2	−1	1	0	0	0	0	0
v_3	0	−1	1	0	1	0	0
v_4	0	0	−1	−1	0	−1	0
v_5	0	0	0	1	−1	0	1

(a) (b)

Şekil 1.8. Bir yönlü graf ve onun A_c matrisi (a) Bir yönlü G grafı (b) G'nin matrisi

A_c'nin tanımından da anlaşıldığı üzere bu matrisin her bir sütununun tam olarak iki sıfır olmayan veriye sahip olduğu açıktır. Bu verilerin biri 1 ve diğeri -1'dir. Bundan dolayı A_c'nin herhangi bir satırı, diğer kalan satırlardan elde edilebilir. Bu yüzden $\text{rank}(A_c) \le n-1$ 'dir.

A_c'nin $A_n(n-1)$ satır alt matrisi G 'nin temel ilişki matrisine karşılık gelir. A_c'de yer almayan düğüm ise referans düğüm olarak adlandırılır.

1.3.4.2. Komşuluk Matrisi

n düğümlü ve m kenarlı döngüsü (self-loop) olmayan bir bağlantılı ve yönlü G grafı olsun. G'nin $K_c = [a_{ij}]$ komşuluk matrisi her biri bir düğüme karşılık gelen n satır ve sütuna sahiptir. K_c'nin a_{ij}'inci elemanı aşağıdaki gibi tanımlanır.

$$a_{ij} = \begin{cases} 1, \text{ eğer kenarın yönü düğümden dışa doğru ise} \\ 0, \text{ eğer kenarın yönü düğüme doğru ise veya} \\ \quad \text{bu düğüm ile komşu değilse} \end{cases}$$

Örnek olarak Şekil 1.8(a)'daki yönlü grafın komşuluk matrisi aşağıdaki gibidir.

$$K_c = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

1.3.4.3. Kesi Matrisi

n düğümlü ve m kenarlı bağlantılı ve yönlü G grafında bir $<V_a, V_b>$ kesisi olsun. $<V_a, V_b>$'nin V_a'daki tüm düğümleri V_b'ye bağlayan tüm kenarları kapsar. Bu kesinin V_a'dan V_b'ye veya V_b'den V_a'ya atanmış bir yönü olabilir. $<V_a, V_b>$'nin yönünün V_a'dan V_b'ye olduğunu varsayalım. Bir (v_i, v_j) kenarının yönü, eğer $v_i \in V_a$ ve $v_j \in V_b$ ise aynı zamanda kesinin de yönüdür.

G'nin $Q_c = [q_{ij}]$ kesi matrisi, her biri bir kenara karşılık gelen m sütuna ve her biri bir kesiye karşılık gelen satırlara sahiptir. q_{ij} elemanı aşağıdaki gibi tanımlanır.

$$q_{ij} = \begin{cases} 1 \text{ , eğer kenar keside ve yönü kesi yönüyle aynı ise,} \\ -1, \text{ eğer kenar keside ve yönü kesi yönüyle aynı değilse,} \\ 0 \text{ , eğer kenar keside değilse.} \end{cases}$$

Q_c'nin her bir satırı, kesi vektörü (cut vector) olarak adlandırılır. Bir düğümle ilişkili kenarlar bir kesiyi oluşturur. Bu yüzden A_c, Q_c'nin bir alt matrisidir.

Bağlantılı G grafının T açılım ağacının her bir dalının bir temel kesi kümesi oluşturur. T ile tanımlanan $n-1$ kesi kümesine karşılık gelen Q_c'nin alt matrisi, G'nin T'ye göre Q_f temel kesi kümesi matrisi olarak adlandırılır.

$b_1, b_2, \ldots b_{n-1}$, T'nin dalları olsun. Temel kesi kümesinin yönünün ele alınan dalla aynı seçildiğini varsayalım. b_i ile tanımlanan temel kesi kümesine karşılık gelen $i.$ sütun olacak

şekilde Q_f'nin satırları ve sütunlarını düzenlersek o zaman Q_f matrisi (1.3)'teki gibi bir formda gösterilebilir.

$$Q_f = [U \mid Q_{fc}] \tag{1.3}$$

Burada U , $n-1$ dereceli bir birim matris ve sütunları T'nin dallarına karşılık gelir. Örnek olarak Şekil 1.8'deki grafın $T = (e_1, e_2, e_3, e_6)$ açılım ağacına göre kesi kümesi matrisi aşağıdaki gibidir.

$$Q_f = \begin{matrix} e_1 \\ e_2 \\ e_3 \\ e_6 \end{matrix} \begin{bmatrix} 1 & 0 & 0 & 0 & -1 & -1 & -1 \\ 0 & 1 & 0 & 0 & -1 & -1 & -1 \\ 0 & 0 & 1 & 0 & 0 & -1 & -1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Q_f'nin rankının $n-1$ olduğu açıktır. Bu yüzden $\text{rank}(Q_c) \geq n-1$'dir.

1.3.4.4. Çevrim Matrisi

n düğümlü ve m kenarlı, bağlantılı ve yönlü bir G grafının bir C çevrimi olsun. Bu çevrim, saat yönünde veya saat tersi yönünde gezinilebilir. C'yi gezinirken seçilen yöne C'nin yönü denir. Eğer C'deki v_i'den v_j'ye yönelmiş bir $e = (v_i, v_j)$ kenarı ise C'nin yönünde gezinirken v_i v_j'den önce gözüküyorsa o zaman yön e'nin yönü ile aynıdır. G'de her biri bir kenara karşılık gelen m sütunu ve her biri bir çevrime karşılık gelen satırları vardır. G'nin $B_c = [b_{ij}]$ çevrim matrisi aşağıdaki gibi tanımlanır.

$$b_{ij} = \begin{cases} 1 \text{ , eğer kenar, çevrimde ve yönü çevrim yönüyle aynı ise,} \\ -1\text{, eğer kenar, çevrimde ve yönü çevrim yönüyle aynı değilse,} \\ 0 \text{ , eğer kenar, çevrimde değilse.} \end{cases}$$

T açılım ağacının kirişleri ile tanımlanan temel çevrimlere karşılık gelen B_c'nin alt matrisi, G'nin T açılım ağacına göre B_f temel çevrim matrisi olarak tanımlanır.

$c_1, c_2, ..., c_{m-n+1}$, T 'nin kirişlerini göstersin. Kirişleri c_i ile tanımlanan ve *i.* satır temel çevrime karşılık gelecek şekilde ve *i.* sütun ise c_i kirişine karşılık gelecek şekilde B_f 'nin satır ve sütunları düzenlenir. Tanımlanan kirişin yönüyle aynı olması için B_f matrisi (1.4)'deki gibi yazılabilir.

$$B_f = [U \mid B_{ft}] \tag{1.4}$$

Burada U , $m-n+1$ 'inci dereceden bir birim matris ve sütunları T 'nin kirişlerine karşılık gelir.

Örnek olarak Şekil 1.8'deki grafın temel çevrim matrisi, $T = (e_1, e_2, e_5, e_6)$ 'ya göre aşağıdaki gibi verilebilir.

$$B_f = \begin{matrix} e_3 \\ e_4 \\ e_7 \end{matrix} \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & -1 \\ 0 & 1 & 0 & 1 & 1 & 1 & -1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

B_f 'nin rankının $m-n+1$ olduğu açıktır. Bu yüzden $\text{rank}(B_c) = m - n + 1$ 'dir.

Teorem 4 (Dikgenlik ilişkisi)**:** Bağlantılı bir graftaki bir çevrim ve kesi kümesi çift sayıda ortak kenara sahiptir. Yönlü bir grafta, eğer bir çevrim ve bir kesi kümesi $2k$ ortak kenara sahipse, o zaman bu kenarların k tanesi çevrimde ve kesi kümesinde aynı relatif yöne sahiptir. Kalan k kenar ise çevrimde bir yöne ve keside ise zıt bir yöne sahiptir.

Teorem 5: Eğer B_c çevrim matrisinin sütunları ve Q_c kesi matrisinin sütunlarının kenar sırası aynı olacak şekilde düzenlenirse, o zaman $B_c Q_c^t = 0$ 'dır.

Teorem 6:

$$\text{rank}(B_c) = m - n + 1 \tag{1.5}$$

$$\text{rank}(Q_c) = n - 1 \tag{1.6}$$

Yukarıdaki teoremin devamı olarak, çevrim matrisinin rankı grafın geçersizliğine (nullity) ve kesi matrisinin rankı ise grafın rankına eşittir. Bu sonuç grafın rankı ve geçersizliği tanımından gelir.

1.3.5. Graf Algoritmaları

Graflar gerçek hayattaki birçok problemin çözümünde kullanılmaktadır. Bu problemlerin çözümünde sık kullanılan graf algoritmaları aşağıdaki verilmektedir. Bu algoritmalarda özellikle graflarda arama veya gezinme algoritmaları diğer birçok graf algoritmalarında çok yaygın kullanılmaktadır.

1.3.5.1. Enine Arama Algoritması

Graflarda, enine arama (breath-first search-BFS) işlemi temel olarak iki işlemde yapılır. Bunlar;

a) Bir grafın bir düğümünü gezinmek ve incelemek,

b) Şu anki gezinilen düğümün komşu düğümlerine erişim sağlamaktır.

BFS, seçilen bir kök (root) düğümden başlar ve komşu düğümlerin tümünü inceler. Daha sonra bu komşu düğümlerin her biri için gezinilmemiş komşu düğümlerini inceler. Bu işlem graftaki tüm düğümler gezinilene kadar devam eder.

BFS [11] algoritması aşağıdaki gibi algoritmik adımlarla gösterilebilir. Bu algoritma ara sonuçları depolamak için kuyruk (queue) veri yapısını kullanır.

Adım 1: Kök düğümü (başlangıç düğümünü) kuyruğa eklenir

Adım 2: Kuyruktan düğümü çek ve incelenir

- Eğer aranılan eleman bu düğüm ise, aramadan çık ve bir sonuca dönülür.
- Diğer durumlarda ise gezinilmemiş komşu çocuk düğümleri kuyruğa eklenir.

Adım 3: Eğer kuyruk boşsa, graftaki tüm düğümler incelenmiştir. Dolaysıyla arama iptal edilir.

Adım 4: Eğer kuyruk boş değilse, Adım 2'den sonraki işlemler tekrarlanır.

BFS algoritmasının yalancı kodu [12] aşağıdaki gibidir. Bu yalancı kodda *G* bir grafı ve *v* ise bir başlangıç düğümünü göstermektedir.

```
procedure BFS(G,v):
    create a queue Q
    enqueue v onto Q
    mark v
    while Q is not empty:
        t ← Q.dequeue()
```

```
    if t is what we are looking for:
        return t
    for all edges e in G.adjacentEdges(t) do
        u ← G.adjacentVertex(t,e)
        if u is not marked:
            mark u
            enqueue u onto Q
return none
```

1.3.5.2. Graf Bağlantılılık Testi

Bağlantılılık testi (connectivity test) bir grafın bağlantılı olup olmadığını kontrol eder. Yönsüz bir *G* grafında eğer *u* ve *v* gibi iki düğüm arasında bir yol (path) varsa bu düğümler bağlantılıdır denir. Eğer bir graftaki bütün düğümler arasında bir yol varsa o zaman bu graf bağlantılıdır denir.

Bir grafın bağlantılılık testini yapmak için enine arama algoritması (BFS) kullanılabilir. Graf bağlantılılık testini yapan algoritma [13] aşağıdaki gibi verilebilir.

Adım 1: G grafının keyfi seçilmiş bir düğümünden başlanır.

Adım 2: BFS kullanarak bu düğümden itibaren işleme devam edilir ve erişilen tüm düğümler sayılır.

Adım 3: Grafın tümü düğümleri gezildikten sonra eğer gezinilen düğüm sayısı G'nin düğüm sayısına eşitse, G grafi bağlantılıdır; diğer durumda ise bağlantısızdır.

1.3.5.3. Rastgele Bağlantılı Graf Üretimi

Rastgele bağlantılı graf (random connected graph), verilen *n* düğüm sayısı ve *m* kenar sayısı için düğümleri bağlayan kenarlar tamamen rastgele olacak şekilde üretilir. Rastgele graf üretiminde üretilecek grafın ilk önce bir rastgele açılım ağacı üretilir. Daha sonra kalan kenarlar belirtilen kenar sayısına varılana kadar grafa rastgele eklenir.

Rastgele açılım ağacı üretiminde, verilen bir n düğüm sayısı için açgözlü (greedy) yöntemle bir yönsüz açılım ağacı üretilir. Burada ilk önce *n* tane nesnenin rastgele dizilimi (permütasyonu) üretilir. Üretilen dizilimindeki (permütasyondaki) ilk düğümü içeren bir kısmi ağaçla işleme başlanır. Daha sonra şu anda kısmi ağaçta var olan düğümlerden bir *k*

düğümü seçilir. Ondan sonra dizilimdeki bir sonraki düğüm ile bu iki düğümü bağlayan bir kenar kısmi ağaca eklenir. Tüm *n* düğümleri kısmi ağaca eklendiğinde işlem sonlandırılır.

Rastgele bağlantılı graf üretilirken, bir grafın bağlantılı olması şartlarından biri olan *m*'nin *(n-1)*'den büyük olmasına bakılır. Ayrıca *m* kenar sayısı, *n* düğüm sayısı için bir tam grafın (complete graph) kenar sayısı olan *(n-(n-1))/2*'den küçük olması gerekir. Aşağıda rastgele bağlantılı graf üreten algoritmanın yalancı kodu [13] verilmektedir.

```
procedure randomConnectedGraph(n, m)

    permute[n+1]

    if m is less than (n-1)
        return false
    if m is grater than (n*(n-1))/2
        return false

    // rastgele açılım ağacını açgözlü yöntemle üret
    permute ← randomPermutation(n);
    for  nodea =2 and nodea<=n
        nodeb ← random(nodea-1)+1
        connect(permute[nodea], permute[nodeb])

    // kalan kenarları ratgele ekle
    for all edges
        nodea ← random(n)+1
        nodeb ← random(n)+1
        if  nodea is not connected to nodeb
            connect(nodea, nodeb)
```

1.4. Elektrik Devreleri

Bu bölümde, elektrik devrelerini oluşturan temel elemanlardan bahsedilmektedir. Ayrıca, elektrik devre analizi veya çözümünde yardımcı olan devre analiz yöntemlerinden de bahsedilmektedir.

Başlangıç olarak akım (current) ve gerilim (voltage) kavramlarıyla bu bölüme giriş yapılacaktır. Çünkü bunlar devreyle ilgili en temel kavramlardır.

1.4.1. Akım, Gerilim ve Devre Yükleri

Bir iletkenin birim saniyede belirli bir kısmından geçen toplam yük miktarına amper denir. Matematiksel olarak (1.7)'deki gibi gösterilebilir.

$$I = \frac{Q}{t} \tag{1.7}$$

Burada Q yükün simgesidir. Birimi coulombtur ve C ile gösterilir. Akımın simgesi I'dır. Birimi amperdir ve A ile gösterilir. t ise saniye (s) cinsinden zamandır.

Bir elektrik devresindeki iki nokta arasındaki gerilim veya potansiyel fark, eğer bu iki nokta arasındaki 1 C'lik yük transferinde 1 Joulelük enerji harcanıyorsa 1 volttur.

Genelde V ile sembolize edilir. Birimi volttur ve V ile sembolize edilir V sembolü aynı zamanda sabit bir gerilimi (doğru akım) de belirtir.

Bir gerilim kaynağı akımın akması için gerekli enerjiyi sağlar. Bunun aksine akım, ancak bir potansiyel fark ve akması için bir yol varsa vardır.

Bir yük (load), elektrik devresine bağlı bir bileşene işaret eder. Yük, aşağıdaki devre elemanlarının herhangi biri veya bunların kombinasyonları ile temsil edilir.

- Direnç (resistor)
- Endüktans (inductor)
- Kapasite (capasitor)

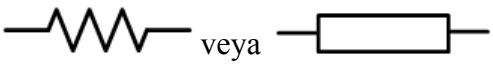
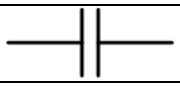
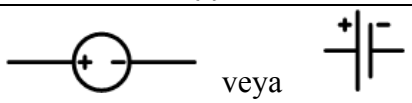
Bir yük resistif, endüktif veya kapasitif veya bunların birleşiminden oluşabilir. Örneğin bir elektrik ampulü resistiftir. Oysa bir transformatör hem endüktif hem de resistiftir.

Bir devre yükü, enerjiyi harcadığından hedef (sink) olarak gösterilir. Gerilim veya akım beslemesi ise kaynak (source) olarak adlandırılır.

Tablo 1.4'te elektrik devresinde kullanılan temel devre elemanları ve sembolleri ve şematik haliyle gösterilmektedir. R, L, C elemanları devredeki kaynaklardan enerjiyi alıp onu ya başka bir şekle dönüştüren ya da elektrik veya manyetik alan olarak depolayan elemanlar olduğundan pasif elemanlardır [17]. Gerilim veya akım kaynakları ise devreye enerji sağladıklarından dolayı aktif elemanlardır.

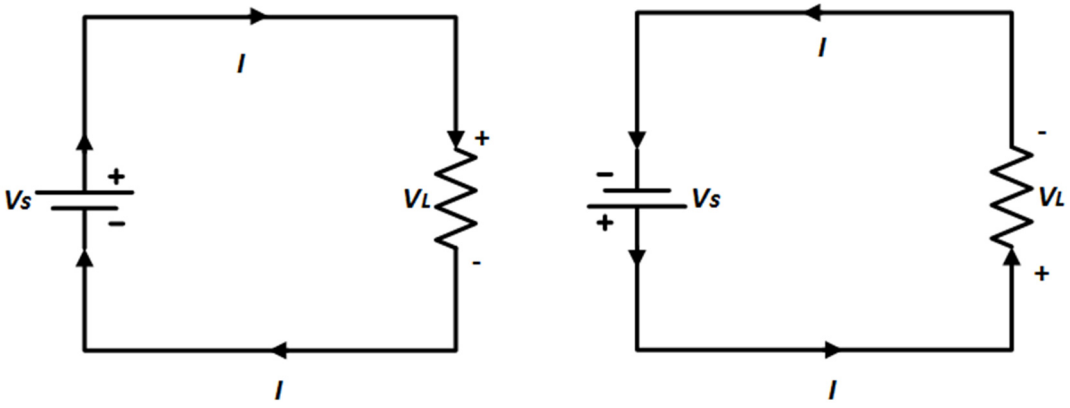
Akımı elektronların akışı olarak düşünmek yaygındır. Fakat akımın yönünü belirlemek için standart olan kural, protonların akışı olarak ele almaktır.

Tablo 1.4. Temel devre elemanları ve elektrik devresindeki gösterimleri

Devre elemanı	Sembolü	Şematik gösterimi
Direnç	R	 veya
Endüktans	L	
Kapasite	C	
DC gerilim kaynağı	V_s	 veya
DC akım kaynağı	I_s	

Verilen bir devrede, akımın yönü, gerilim kaynağının polaritesine bağlıdır. Akım her zaman kaynağın pozitif tarafından (yüksek potansiyel) negatif tarafına doğru akar. Şekil 1.10'da bu ifadenin şematik gösterimi verilmektedir. V_s gerilim kaynağı, V_L yükün uçları arasındaki gerilim ve I ise saat yönünde akan çevre (loop) akımıdır.

Dikkat edilirse gerilim polaritesi ve akımın yönü kaynağınkine zıttır. Kaynakta, akım pozitif uçtan çıkar. Yükte (sink, load) ise akımın girdiği uç pozitiftir.



(a) (b)

Şekil 1.9. İşaret kuralı

Gerilim kaynağındaki polaritenin tersine çevrilmesi Şekil 1.9(a)'da ve Şekil 1.9 (b)'de gösterildiği gibi akımın akış yönünü değiştirir ve bu ifadenin tersi de doğrudur.

1.4.2. Direnç, Kapasite ve Endüktans

Bir direncin karakteristiğini anlatmak için Ohm kanununu tanımlamak önemlidir.

Ohm kanununu [14], devre analizinde kullanılan en temel kanundur. Bir iletken üzerindeki gerilim-akım ilişkisini tanımlayan basit bir formülü verir. Sözel olarak ifade edilirse Ohm kanunu, "Bir iletkenin uçları arasındaki gerilim veya potansiyel fark, o iletken üzerindeki akımla doğru orantılıdır".

Matematiksel olarak (1.8)'deki gibi ifade edilir.

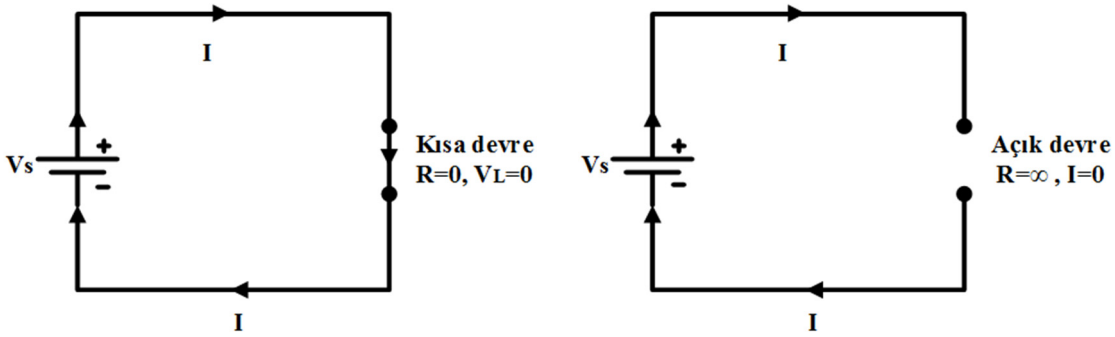
$$V = I.\mathrm{R} \tag{1.8}$$

Burada, R direnç olarak adlandırılır, birimi ohmdur ve Ω ile gösterilir.

İki nokta arasındaki kısa devre (short circuit) sıfır direnci temsil eder. Açık devre (open circuit) ise sonsuz dirence karşılık gelir. Şekil 1.10'da açık devre ve kısa devre gösterilmektedir.

Ohm kanunu göre;

- R=0 (kısa devre) iken, V = 0 V ve
- R=∞ (açık devre) iken, I = 0 A'dır.



Şekil 1.10. Kısa devre ve açık devre direnç karakteristikleri

İletkenlik (conductance), direncin tam zıttıdır ve G ile sembolize edilir. Matematiksel olarak (1.9) ve (1.10)'daki gibi ifade edilir.

$$G = \frac{1}{R} \tag{1.9}$$

$$I = \frac{V}{R} = V.G \tag{1.10}$$

Kapasite, bir elektrik alandaki yükü depolamak için kullanılan pasif devre elemanıdır. Kapasitenin *V-I* bağıntısı (1.11)'deki gibi ifade edilebilir.

$$i = C.\frac{dv}{dt} \tag{1.11}$$

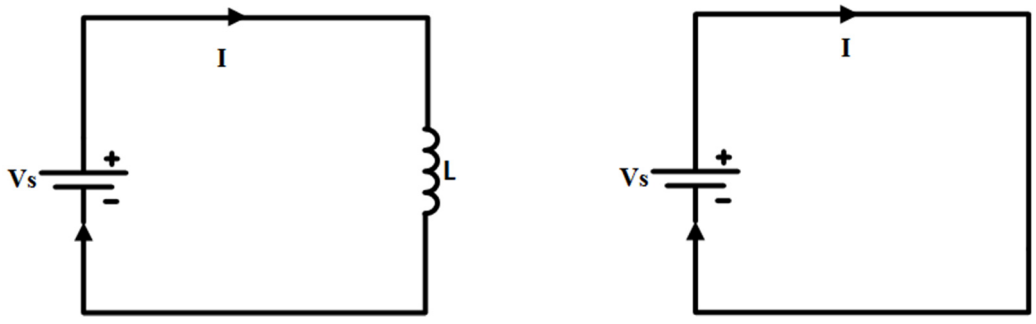
Kapasite, C ile gösterilir ve birimi faradtır. *v(0)* ise başlangıç gerilimi veya kapasitede depolanmış başlangıç yüküdür. v=V (sabit doğru akım gerilimi) iken, $\frac{dv}{dt} = 0$ ve $i = 0$ dır. Dolaysıyla kapasite doğru akıma açık devre olarak davranır.

Endüktans, ferromagnetik maddenin çekirdeği etrafına sarılı iletken bir kablo parçasıdır. Bir endüktansta *V-I* arasındaki ilişki (1.12)'deki gibidir.

$$v = L.\frac{di}{dt} \tag{1.12}$$

Burada L, endüktanstır ve birimi henrydir. Ayrıca $i(0)$ ise endüktansın manyetik alanında depolanan başlangıç akımıdır.

$i = I$ (sabit doğru akım) iken $\frac{di}{dt} = 0, v = 0$ 'dır. Bu yüzden endüktans doğru akıma kısa devre gibi davranır. İdeal bir endüktans, iç direnci ve kapasitesi olmayan bir iletkendir. Kaynak gerilim doğru akım iken, Şekil 1.11'deki çizimler birbirine denktir.



Şekil 1.11. İdeal bir endüktansın kısa devre gösterimi

1.4.3. Doğru Akım Kaynakları

Bir elektrik devresinde genel olarak iki ana türde doğru akım kaynağı vardır. Bunlar;

- Bağımsız gerilim ve akım kaynakları
- Bağımlı gerilim ve akım kaynaklarıdır.

Bağımsız bir kaynak kendi akım veya gerilimini, devredeki diğer gerilim veya akım değişkenlerine bağlı olmadan sağlar. Diğer taraftan, bağımlı kaynak ise bir devre elemanının gerilim veya akım değişimine bağlıdır. Burada sadece bağımsız kaynaklar ele alınmaktadır.

İdeal bir gerilim kaynağı, yükteki değişimlerden bağımsız bir uç (terminal) gerilimine sahiptir. Diğer bir ifadeyle, ideal bir gerilim kaynağı için yükteki değişimle akım da değişir. Fakat V_L uç gerilimi her zaman sabit kalır.

Akım kaynağının doğru gerilim kaynağından farklı olarak bir gerçekliği yoktur. Fakat transistör gibi yarı iletken aygıtların eşdeğer devre modellerini türetirken oldukça faydalı olmaktadır. İdeal bir akım kaynağı, yükteki gerilimlerden bağımsız olarak akım üretir. Diğer bir deyişle, ideal bir akım kaynağı tarafından sağlanan bir akım yük gerilimiyle değişmez.

1.4.4. Devre Teoremleri

Bu bölümde, elektrik devrelerini analiz etmek için en sık kullanılan kanunlar ve teoremlerden bahsedilmektedir.

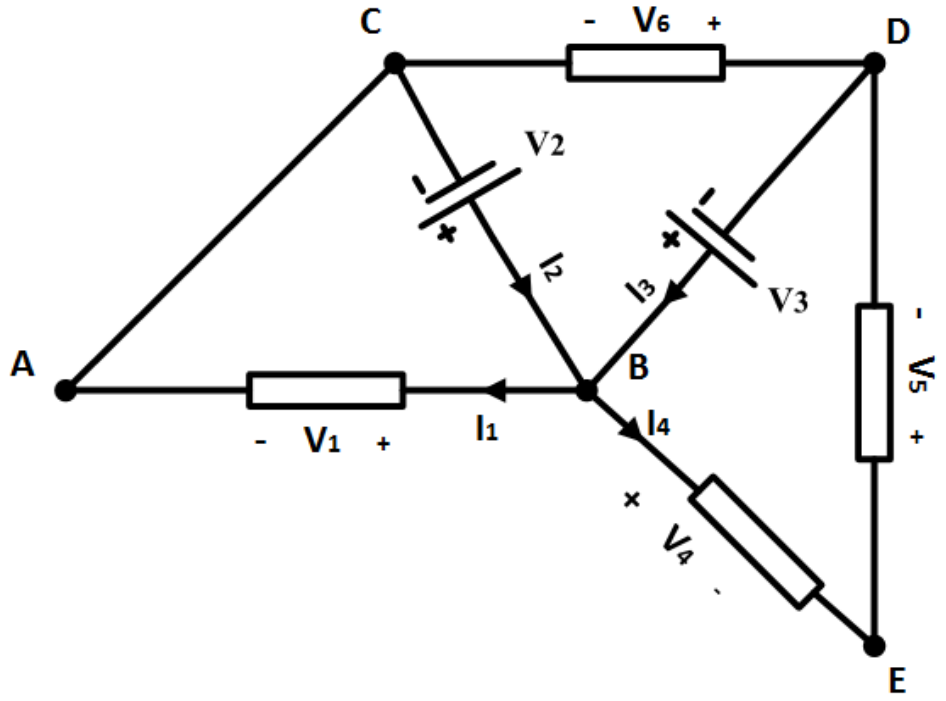
Aşağıda, devre analizinde sık kullanılan çeşitli tanımlar ve terminolojiler verilmektedir.

- Elektrik ağı: Çeşitli devre elemanlarının bağlantısı elektrik ağı olarak adlandırılır. Şekil 1.13'teki devre diyagramı bir elektrik ağıdır.
- Elektrik devresi: Elektrik ağındaki çeşitli devre elemanlarının bağlanmasıyla bir kapalı yolu şekillendirme elektrik devresi olarak adlandırılır. Kapalı yok genellikle çevre (loop, mesh) olarak adlandırılır. Şekil 1.13'te BDEB, ABCA ve BCDB çevreleri kapalı bir yol oluşturmaktadır. Genellikle bütün devreler (circuit) ağ olmasına rağmen tüm ağlar devre değildir.
- Düğüm (node): Birkaç devre elemanının bağlantı noktası düğüm olarak adlandırılır. Örneğin Şekil 1.13'teki A, B, C, D ve E birer düğümdür. A ve C arasına bir eleman bağlı olmadığından bunlar aynı düğüm olarak ele alınabilirler.

- Kol (branch): Bir elektrik devresindeki iki düğüm arasındaki yol kol olarak adlandırılır. Şekil 1.13'teki AB, BE, BD, BC, CD ve DE olmak üzere altı tane kol vardır.

Elektrik devre analizlerinde en sık kullanılan kanunlar Kirchhoff'un akım ve gerilim kanunlarıdır.

Kirchhoff'un gerilim kanunu [14], "Kapalı bir çevre etrafındaki gerilimlerin cebirsel toplamı sıfırdır." şeklinde ifade edilir. Diğer bir deyişle, bir kapalı çevre etrafındaki, tüm gerilim artışlarının cebirsel toplamı, tüm gerilim düşüşlerinin cebirsel toplamına eşittir.



Şekil 1.12. Düğüm, kol, eleman ve çevreleri gösteren elektrik ağı

Şekil 1.12'deki BEDB çevresi ele alınırsa, KVL'ye göre (1.13)'teki denklem yazılabilir.

$$V_3 - V_4 - V_5 = 0 \tag{1.13}$$

Kirchhoff'un akım kanunu [14] , "Bir elektrik devresindeki bir düğüme giren ve çıkan tüm akımların cebirsel toplamı sıfıra eşittir." şeklinde ifade edilir.

Diğer bir deyişle, bir düğüme giren akımlar toplamı, aynı düğümden çıkan akımlar toplamına eşittir. Şekil 1.13'teki B düğümü ele alınırsa, KCL'ye göre aşağıdaki denklem oluşturulabilir.

$$I_1 + I_4 - I_2 - I_3 = 0 \tag{1.14}$$

Doğru akım (DC) elektrik analizinde KVL, KCL ve Ohm kanunları temel araçlardır. Düğüm analizi (nodal analysis) terimi, elektrik devresi genelde KCL ile analiz ediliyorsa, çevre (mesh) analizi ise elektrik devresi KVL kullanılarak analiz ediliyorsa kullanılır. Çevre ve düğüm analiz yöntemleri aşağıda sistematik olarak verilmektedir.

1.4.5. Çevre Analizi ve Düğüm Analizi

Çevre analizi (Mesh Analysis) yöntemi [14] aşağıdaki adımlardan oluşmaktadır.

Adım 1: Eğer mümkünse tüm akım kaynaklarını gerilim kaynaklarına çevrilir.

Adım 2: Elektrik ağındaki her bir çevreyi belirle ve her birine bir akım ataması yapılır(tercihen aynı yönde).

Adım 3: Oluşan denklem sistemlerini çözülür. Burada kurulan denklem sayısı ağdaki çevre sayısına eşittir.

Düğüm analiz (Nodal Analysis) yöntemi [14] aşağıdaki adımlardan oluşmaktadır.

Adım 1: Mümkünse tüm gerilim kaynaklarını akım kaynaklarına çevrilir.

Adım 2: Ağdaki her bir düğümü belirle ve referans düğüm dahil olmak üzere tüm akımların düğümlerden çıktığını varsayarak her bir düğüme bir gerilim ataması yapılır. Genellikle, referans düğümün geriliminin sıfır olduğu kabul edilir.

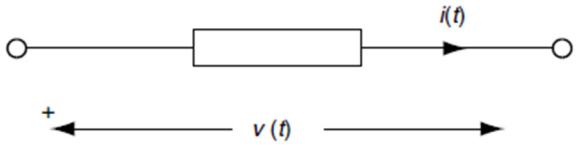
Adım 3: KCL kullanarak düğüm denklemlerini yaz ve onları sadeleştirilir.

Adım 4: Oluşan denklem sistemlerini çözülür.

Burada referans düğüm dahil ağdaki düğüm sayısı *n* ise denklem sayısı *n-1*'dir.

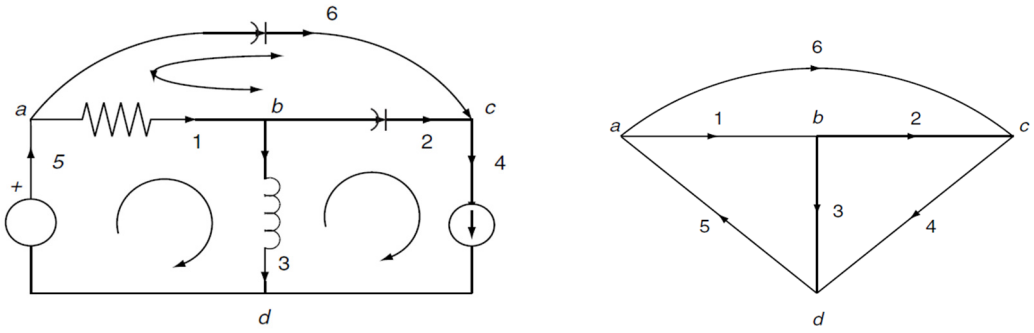
1.4.6. Graf Analiz Yöntemi

Bir elektriksel ağın direnç, kapasite, endüktans, gerilim ve akım kaynakları gibi elektriksel elemanlarının bağlanması sonucu oluşturulur. Her bir ağ elemanı $v(t)$ gerilim değişkeni ve $i(t)$ akım değişkeni gibi iki değişkenle ilişkilendirilir. Şekil 1.13'te gösterildiği gibi ağ elemanlarına referans yönleri atanır. $i(t)$ akımı ile okun yönü aynı olduğunda pozitif ve $v(t)$ ise ağ elemanındaki gerilim düşüşü ile okun yönü aynıysa pozitiftir.



Şekil 1.13. Bir ağ elemanı ve onun referans yönü

Her bir eleman ve ilişkili olduğu referans yönünün yönlü bir kenarla yer değiştirilmesi sonucu, ağı temsil eden yönlü graf oluşturulur. Örneğin Şekil 1.14'te basit bir elektriksel ağ ve ona karşılık gelen yönlü graf gösterilmektedir [16].



(a) (b)

Şekil 1.14. KVL ve KCL denklemlerinin graf örnekleri (a) N elektriksel ağı
(b) N'nin yönlü graf gösterimi

Ağ elemanlarının akım ve gerilim değişkenleri arasındaki ilişki Ohm kanunu ile belirlenir. Gerilim ve akım kaynaklarının gerilim ve akım değişkenleri belirli değerleri olmasını gerektirir. Ağdaki gerilim değişkenleri arasındaki lineer bağımlılık ve akım değişkenleri arasındaki lineer bağımlılık sırasıyla Kirchhoff'un akım (KCL) ve gerilim

(KVL) kanunları ile yönetilir. Örnek olarak Şekil 1.14'teki *"1,3,5"* çevrimi için KVL denklemi (1.15)'teki gibi ve b düğümü için KCL denklemi ise (1.16)'daki gibi verilir.

$$V_1 + V_3 + V_5 = 0 \tag{1.15}$$

$$-I_1 + I_2 + I_3 = 0 \tag{1.16}$$

Bir N elektriksel ağının KVL denklemi (1.17)'deki gibi ve KCL denklemi ise (1.18)'deki gibi yazılabilir.

$$A_c \mathrm{I}_e = 0 \tag{1.17}$$

$$B_c \mathrm{V}_e = 0 \tag{1.18}$$

Burada, A_c ve B_c sırasıyla N'yi temsil eden yönlü grafın ilişki ve çevrim matrisleridir. I_e ve V_e ise N'deki ağ elemanlarının sırasıyla akım ve gerilim sütun vektörleridir. Q_c'deki her bir satır matrisin satırlarının lineer kombinasyonu olarak ifade edilebileceğinden yukarıdaki A_c, Q_c ile yer değiştirebilir. Sonuç olarak KCL için (1.19) denklemi ve KVL için ise (1.20) denklemi yazılabilir.

$$Q_c \mathrm{I}_e = 0 \tag{1.19}$$

$$B_c \mathrm{V}_e = 0 \tag{1.20}$$

Böylelikle KCL aynı zamanda şöyle tanımlanabilir. N'nin herhangi bir kesisindeki akımların cebirsel toplamı sıfıra eşittir.

Eğer N ağı n düğüme ve m elemana sahipse ve karşılık gelen grafı bağlantılı ise, o zaman sadece $n-1$ tane lineer bağımsız kesi ve $m-n+1$ tane lineer bağımsız çevrim vardır. Bu yüzden KVL ve KCL denklemlerini yazarken sadece sırasıyla B_f ve Q_f kullanılması yeterlidir. Sonuç olarak KCL için (1.21) denklemi ve KVL için ise (1.22) denklemi yazılabilir.

$$Q_f \mathrm{I}_e = 0 \tag{1.21}$$

$$B_f \mathrm{V}_e = 0 \tag{1.22}$$

Burada B_f, temel çevrim matrisi ve Q_f ise temel kesi kümesi matrisidir.

Bu bölümünün kalan kısmında elektrik ağ analizinin bu sonuçlarla olan ilişkisinden bahsedilmektedir. Bu sonuçlarda, bir N ağı ve onun yönlü graf gösterimi N ile simgelenmektedir.

1.4.6.1. Çevre ve Kesi Kümesi Dönüşümleri

T, bir elektriksel ağın açılım ağacı olsun. I_c ve V_t, T'ye göre kiriş akımları ve dal gerilimlerinin sütun vektörleri olsun.

- Çevre dönüşümü:

$$\mathrm{I}_e = B_f^t \mathrm{I}_c \tag{1.23}$$

- Kesi kümesi dönüşümü

$$\mathrm{V}_e = Q_f^t \mathrm{V}_t \tag{1.24}$$

Eğer kesi kümesi dönüşümünde Q_f, A ilişki matrisiyle yer değiştirirse, o zaman aşağıda verilen düğüm dönüşümü (node transformation) elde edilir.

$$\mathrm{V}_e = Q^t \mathrm{V}_n \tag{1.25}$$

Burada V_n vektöründeki elemanlar, r referans düğümüne karşılık gelen düğümlerin gerilimleri olarak yorumlanabilir. (Not: A matrisi r düğümüne karşılık gelen satırı içermez).

Yukarıdaki dönüşümler, ağ analizinin farklı yöntemlerini geliştirmede kapsamlı olarak çalışılmıştır. Aşağıdaki bölümlerde bu yöntemlerin iki tanesi anlatılmaktadır.

1.4.6.2. Çevre ve Kesi Kümesi Denklem Sistemleri

Daha öncede bahsedildiği gibi ağ analizi problemi, bir elektriksel ağın elemanlarıyla ilişkili akım ve gerilimlerini belirlemektir. Bu akım ve gerilimler Kirchhoff'un denklemleriyle ve gerilim-akım ($v-i$) bağıntıları ise Ohm kanunu ile belirlenir. Fakat bu denklemler çok fazla değişken içermektedir. Çevre ve kesi kümesi dönüşümlerinden görüldüğü gibi bu değişkenlerin hepsi bağımsız değildir. Dahası KCL denklemleri yerine, sadece kiriş akımlarını dikkate alan çevre dönüşümleri kullanılabilir. Benzer şekilde KVL denklemleri ise sadece kol gerilim değişkenlerini dikkate alan kesi kümesi dönüşümleriyle yer değiştirebilir. Bu dönüşümlerin kullanımı çevre ve kesi kümesi sistemleri olarak bilinen farklı ağ denklem sistemlerinin kurulmasına imkan sağlar.

Çevre sistemlerinin türetiminde, çevre dönüşümü, KCL'nin yerine kullanılır. Bu durumda çevre değişkenleri(kiriş akımları) bağımsız değişkenler olarak hizmet eder. Kesi kümesi sistemi türetiminde, KVL yerine kesi kümesi dönüşümü kullanılabilir. Bu durumda kesi kümesi değişkenleri (ağaç dal gerilimleri) bağımsız değişkenler olarak hizmet ederler. N elektriksel ağının sadece direnç, kapasite, endüktans bağımsız gerilim ve akım kaynaklarından oluştuğu varsayılmaktadır. Aynı zamanda başlangıç endüktans akımları ve kapasite gerilimlerinin uygun kaynaklarla yer değiştirildiği varsayılmaktadır. Dahası, gerilim ve akım değişkenlerinin hepsi s kompleks frekans değişkeninin Laplace dönüşümüdür. N'de sadece gerilim kaynaklarından oluşan hiçbir çevrim olamaz. Eğer böyle bir kaynağı içeren bir çevrim varsa, KVL'den dolayı karşılık gelen gerilimler arasında lineer bir ilişki olacaktır ve dolaysıyla bu da gerilim kaynaklarının bağımsız olmasını ihlal edecektir. Aynı nedenle N'de sadece akım kaynaklarından oluşan hiçbir kesi kümesi olamaz. Dolaysıyla N'de tüm gerilim kaynaklarını içeren fakat akım kaynaklarını içermeyen bir açılım ağacı vardır. Böyle bir ağaç, çevre ve kesi kümesi denklem sistemlerinin geliştirilmesi için başlangıç noktasıdır.

T verilen bir ağın açılım ağacı olsun. Öyle ki T tüm gerilim kaynaklarını içersin fakat hiçbir akım kaynağı içermesin. V_e eleman gerilim vektörü ve I_e akım vektörü parçalanırsa aşağıdaki gibi olur.

$$\mathrm{V}_e = \begin{bmatrix} \mathrm{V}_1 \\ \mathrm{V}_2 \\ \mathrm{V}_3 \end{bmatrix} \tag{1.26}$$

$$\mathrm{I}_e = \begin{bmatrix} \mathrm{I}_1 \\ \mathrm{I}_2 \\ \mathrm{I}_3 \end{bmatrix} \tag{1.27}$$

1, 2 ve 3 altsimgeleri ile gösterilen değişkenler sırasıyla akım kaynaklarına, R, L, C elemanlarına ve gerilim kaynaklarına karşılık gelen vektörlerdir.

B_f, N 'nin temel çevrim matrisi olsun ve Q_f 'de T açılım ağacına göre N 'nin temel kesi küme matrisi olsun. O zaman KVL denklemi (1.28)'deki gibi ve KCL denklemi ise (1.29)'daki gibi yazılabilir.

$$B_f \mathrm{V}_e = \begin{bmatrix} U & B_{12} & B_{13} \\ 0 & B_{22} & B_{23} \end{bmatrix} \begin{bmatrix} \mathrm{V}_1 \\ \mathrm{V}_2 \\ \mathrm{V}_3 \end{bmatrix} = 0 \tag{1.28}$$

$$Q_f \mathrm{I}_e = \begin{bmatrix} Q_{11} & Q_{12} & 0 \\ Q_{21} & Q_{22} & U \end{bmatrix} \begin{bmatrix} \mathrm{I}_1 \\ \mathrm{I}_2 \\ \mathrm{I}_3 \end{bmatrix} = 0 \tag{1.29}$$

1.4.6.3. Çevre Yöntemi ile Ağ Analizi

Çevre yöntemi ile devre analizi aşağıdaki adımlardan oluşmaktadır [17].

Adım 1: I_l vektörü için aşağıdaki denklem çözülür. I_l, T açılım ağacının kaynak olmayan kirişlerindeki akım vektörüdür.

$$Z_l I_l = -B_{23} V_3 - B_{22} Z_2 B_{12} I_1 \tag{1.30}$$

Burada Z_2 matrisi, R, L, C elemanlarının empedans matrisidir.

$$Z_1 = B_{22} Z_2 B_{22} \tag{1.31}$$

(1.30) denklemi çevre denklem sistemleri olarak adlandırılır.

Adım 2: I_2 hesaplanır.

$$I_2 = B_{12} I_1 + B_2 I_1 \tag{1.32}$$

$$V_2 = Z_2 I_2 \tag{1.33}$$

Adım 3: V_1 ve I_3 aşağıdaki denklemler kullanılarak bulunur.

$$V_1 = B_{12} V_2 - B_{13} V_3 \tag{1.34}$$

$$I_3 = B_{13} I_1 + B_{23} I_1 \tag{1.35}$$

1.4.6.4. Kesi Kümesi Yöntemi ile Ağ Analizi

Kesi kümesi yöntemi ile devre analizi aşağıdaki adımlardan oluşmaktadır [21].

Adım 1: V_b vektörü için aşağıdaki denklem çözülür. V_b, T açılım ağacının kaynak olmayan dallarının gerilim vektörüdür.

$$Y_b V_b = -Q_{11} I_1 - Q_{12} Y_2 Q_{22} V_b \tag{1.36}$$

Burada Y_2 matrisi, R, L, C elemanlarının geçiri (admittance) matrisidir.

$$Y_b = Q_{12} Y_2 Q_{12} \tag{1.37}$$

(1.36), kesi kümesi denklem sistemleri olarak adlandırılır.

Adım 2: V_2 hesaplanır.

$$V_2 = Q_{12} V_b + Q_{22} V_3 \tag{1.38}$$

$$I_2 = Y_2 V_2 \tag{1.39}$$

Adım 3: V_1 ve I_3, (1.40) ve (1.41) kullanılarak bulunur.

$$V_1 = Q_{21}V_b + Q_{21}V_3 \tag{1.40}$$

$$I_3 = -Q_{21}I_1 + Q_{22}I_2 \tag{1.41}$$

I_1 ve V_3 akım ve gerilim kaynakları olduğundan dolayı belirli değerlere sahiptir.

1.5. Doküman Biçimlendirme

Doküman biçimlendirme, gerek kullanıcıya gösterilen gerekse editör arayüzleri kullanılarak kullanıcıya gösterilmeyen doküman biçimlendirme dilleri kullanılarak yapılır. Doküman biçimlendirme dili, bir dokümanın metninden sözdizimsel olarak ayrı olan ek açıklamalar ekleyen modern bir sistemdir. Biçimlendirme (markup), metni görüntüleyecek olan programa uygun eylemleri gerçekleştirmesi için gereken direktifleri (instructions) içerir. Bu direktifler, metnin kullanıcıya gösterilen versiyonunda yer almaz. Elektronik biçimlendirme genel olarak üç kategoriye ayrılabilir [18].

Gösterimsel biçimlendirme (Presentational Markup): Bu biçimlendirme çeşidi, Microsoft Word ve OpenOffice gibi geleneksel kelime-işlemci sistemleri tarafından kullanılır. WYSIWYG (What You See Is What You Get) efektini üreten kodlamalar doküman metnine gömülür ve bu kodlamalar, editör veya yazar olsalar bile kullanıcıya gösterilmeyecek şekilde yapılır. Kullanıcı sadece kelime-işlemci editörünü kullanarak bu biçimlendirme efektlerini elde etmeye çalışır.

Yordamsal Biçimlendirme (Procedural Markup): Bu biçimlendirme çeşidinde, biçimlendirme, metne gömülür ve metni işleyecek olan programlar için direktifler (instructions) içerir. Çok iyi bilinen örnekleri troff [19], LaTeX [20] ve PostScript [21]'tir. Program, girdi olarak verilen metni başından sonuna kadar karşılaşılan direktifleri işleyerek çıktı dokümanını üretir. Biçimlendirilmiş metin, istendiğinde gerek asıl metin gerekse gömülü olan direktifler değiştirilerek düzenlenebilir. Popüler yordamsal biçimlendirme sistemleri genellikle programlama yapıları içerir. Bu sayede yeni makro veya altyordamlar tanımlanarak yeni biçimlendirme direktifleri oluşturulabilir.

Betimsel Biçimlendirme (Descriptive Markup): Bu biçimlendirme çeşidinde ise, biçimlendirme, dokümanın nasıl işleneceğini belirten direktiflerden ziyade doküman

parçalarını etiketlemek için yapılır. Amaç dokümanın yapısını, herhangi bir davranışı veya yorumlamasından ayırmaktır. Betimsel biçimlendirmeye önde gelen dillerden biri olan XML örnek verilebilir.

Biçimlendirme çeşitlerine bakıldığında, aralarında bir anlam kargaşası olduğu görülmektedir. Örneğin modern kelime-işlemci sistemlerinde gösterimsel biçimlendirme çoğu kez XML gibi betimsel biçimlendirme tabanlı olarak saklanır ve yapılan gerçeklemelerle yordamsal olarak işlenir.

1.5.1. LaTeX

LaTeX [20], TeX'e dayalı bir makro paketidir. Amacı, özellikle matematiksel formüller içeren belgeler için, TeX dizgilemesini basitleştirmektir.

LaTeX'in bir grup TeX komutu kapsamasından beri, LaTeX belge işleme aslında programlamadır. Bir LaTeX biçimlendirme türünde bir metin dosyası oluşturulur. Daha sonra LaTeX makro bu kaynak metin dosyasını okuyarak çıktı dokümanı üretir.

1.5.2. Temel Kurallar

Temel LaTeX sadece dizgileme (typesetting) komutlarından oluşur. Dizgileme komutları daima "\" ile başlar ve her hangi bir argüman daima küme parantezleri "*{}*" içine yerleştirilir.

LaTeX komutları küçük harf duyarlıdır ve aşağıdaki iki biçimden birini alır:

- Ters bölü (\) ile başlarlar ve sonra sadece harflerden oluşan bir isme sahiptirler. Komut isimleri bir boşluk, bir numara veya başka bir harf olmayan karakter ile sona erer.
- "\" içerir ve tamamen bir harf olmayan karakterden oluşur.

Bazı komutlar bir argüman gerektirir ve bu argüman komut isminden sonra küme parantezleri "*{ }*" arasında verilmelidir. Bazı komutlar isteğe bağlı parametreleri destekler ve bu parametreler köşeli parantez "*[]*" içinde komut isminden sonra eklenir. Genel sözdizimi "*\komutismi[seçenek1,seçenek2,...]{argüman1}{argüman2}...*" şeklindedir.

"# $ % ^ & _ { } ~ \" sembolleri saklı karakterlerdir. Eğer bu karakterler metin içerisine doğrudan girilirse bunların baskısı yapılmaz. Bu karakterlerin hepsi belgelerinizde ters bölü (backslash) ön eki ekleyerek kullanılabilir.

LaTeX içindeki ortamlar (environment) komutlara oldukça benzer bir işleve sahip olmasına rağmen genellikle belgenin daha geniş bir bölümü üzerinde etkileri vardır. Ortamların sözdizimi Tablo 1.5'teki gibidir.

Tablo 1.5. Ortamların genel sözdizimi

\begin{environmentname} *Etkilenecek metin* *\end{environmentname}*

"\begin" ve *"\end"* arasına diğer komutlar ve iç içe ortamlar koyulabilir. Genel olarak ortamlar argüman kabul edebilirler. LaTeX içinde her şey komutlar ve ortamlar cinsinden ifade edilebilir.

1.5.3. Doküman Oluşturma

LaTeX için girdi dosyası düz ASCII metin dosyasıdır. Herhangi bir metin düzenleyici ile oluşturulabilir. Bu metin dosyasında hem belgenin metni hem de metnin nasıl dizgileneceğini belirten komutlar yer alır.

LaTeX girdi dosyası, belirli bir yapıya uygun yazılması gerekir. Her girdi dosyası *"\documentclass{...}"* komutuyla başlamalıdır. Bu komut hangi tür belge yazılacağını belirtir. Bundan sonra, tüm belgenin şeklini etkileyen komutlar eklenebilir veya LaTeX sistemine yeni özellikler ekleyen paketler yüklenebilir. Yeni bir paket yüklemek için *"\usepackage{...}"* komutu kullanılır. Tüm kurulum çalışmaları tamamlandığında, metnin gövdesine *"\begin{document}"* komutuyla başlanır. Daha sonra LaTeX komutları ile kurallara uygun biçimde metin yazılarak *"\end{document}"* komutuyla da belgenin sonunun geldiği belirtilir. Bu komutu takip eden komutlar LaTeX tarafından görmezden gelinir.

Herhangi bir metin düzenleyici de oluşturulan kaynak metin "*yeni_dosya.tex*" gibi *".tex"* uzantılı bir dosya halinde kaydedilir. Kaynak metin artık derlenmek üzere hazırdır. Windows'ta çalışan MiKTeX gibi hazır paketler yardımıyla derlenerek *".pdf"* uzantılı çıktı dosyaları oluşturulur.

1.5.4. Doküman Düzeni

Hazırlanan dokümanda okuyuculara, aradıkları kısmı kolayca bulmalarında yardımcı olmak için doküman, bölümlere ve alt bölümlere parçalanır. Bu şekilde doküman daha okunabilir formatta hazırlanmış olur.

1.5.4.1. Paragraf Oluşturma ve Bölümleme

LaTeX, birbirine komşu satırları, aynı paragrafmış gibi yorumlar. Yeni bir paragraf başlatmak için ekstra bir *"return"* yerleştirilir.

Tablo 1.6. Örnek bir paragraf metni ve çıktısı

Kaynak metin	**Çıktı**
Bu bir paragraftır. Bu da diğer bir paragraftır.	Bu bir paragraftır. Bu da diğer bir paragraftır.

Yeni bir paragraf başlatmadan yeni bir satıra geçmek için *"\\"* kullanılır. LaTeX girdi dosyası işlenirken, bir *"%"* karakteri ile karşılaştığında, bu satırın geri kalanı yok sayılır ve ayrıca satırın sonu ve sonraki satırın başlangıcının tamamını boşluk sayar. Bu girdi dosyasına not veya açıklama alırken kullanılabilir. Bu girilen metin basılı versiyonda görülmez.

Tablo 1.7. Örnek açıklama girdisi ve çıktısı

Kaynak Metin	**Çıktı**
Bu bir % deneme % Daha iyi: öğretici <---- örnektir: Supercal% ifragilist% icexpialidocious	Bu bir örnektir: Supercalifragilisticexpialidocious

LaTeX dokümanı benzer konular içeren parçalara ayırmak için bölümleme komutunu da desteklemektedir. Genelde kullanılan iki çeşit bölümleme komutu vardır.

Bölüm oluşturmak için kullanılan komut *"\section{Bölümün adı}"* ve altbölüm için kullanılan komut ise *"\subsection{Altbölümün adı}"* komutudur.

1.5.4.2. Tablolar

LaTeX'te tablolar oluşturmak için *"tabular"* ortamı (environment) kullanılır. *"tabular"* ortamı yatay ve dikey çizgi seçeneği ile tablo oluşturulmasında kullanılır. LaTeX sütun genişliklerini otomatik olarak kendisi ayarlar. Bu ortamın ilk satırı "*\begin{tabular}[konum]{"tabloBelirt"}*" biçimindedir. Burada, *tabloBelirt* argümanı LaTeX'e her sütunun nasıl hizalanacağını söyler. Ayrıca tabloya dikey çizgi koyulması bilgisini de verir. Sütun sayısı bilgisi, *tabloBelirt* argümanında yer aldığı için bu sayının ayrıca belirtilmesine gerek yoktur. Tablo 1.8'de sütun hizalaması ve dikey çizgiden sorumlu argümanlar yer almaktadır.

Tablo 1.8. LaTeX hizalama argümanları

Hizalama argümanı	**Anlamı**
l	Sola dayalı sütun
c	Ortalanmış sütun
r	Sağa dayalı sütun
p{genişlik}	Paragraf sütunu, yazı dikey olarak sütunun tavanına hizalanır
m{genişlik}	Paragraf sütunu, yazı sütunun ortasına koyulur
b{genişlik}	Paragraf sütunu, yazı dikey olarak ütunun tabanına hizalanır
|	Dikey çizgi
||	Çifte dikey çizgi

Bir sütundaki yazı aşırı geniş ise, LaTeX kendiliğinden sütun genişliğini buna göre ayarlamaz. Bu durumda, *"p{genişlik}"* yaparak, yazıyı içine alacak özel bir sütun tipi tanımlanır. Burada genişlik herhangi bir pt, cm gibi geçerli uzunluk birimi cinsinden olabilir. Bunun yerine *"\textwidth"* gibi bir uzunluk komutu da kullanılabilir.

İsteğe bağlı değişken konum, yazıya göre tablonun nasıl konumlanması gerektiğini belirtir. Birçok durumda bu seçeneği kullanmaya gerek yoktur. Bu tip konumlamalar Tablo 1.9'daki anahtar harfler kullanılarak yapılır.

Tablo 1.9. Konumlama anahtar harfleri

Konumlama anahtar harfi	**Anlamı**
b	Alt
c	Orta (varsayılan konum)
t	Üst

Ortam içinde tablo hücrelerini belirtmek ve yeni satırlar girmek için Tablo 1.10'daki komutlar kullanılabilir.

Tablo 1.10. Tablo komutları

Tablo komutu	**Anlamı**
&	Sütun ayıracı
\\	Yeni satır başlat
\hline	Yatay çizgi
\newline	Bir tablo hücresi içinde yeni bir satır başlat
\cline{i-j}	Kısmi yatay çizgi, i. ile j. sütun arasında.

1.5.4.3. Listeler

Listeleri oluşturmak için *"enumerate"* ve *"itemize"* komutları kullanılır. Bunlardan *"enumerate"* komutu sıralı listeler oluşturmak için kullanılırken *"itemize"* komutu ise sırasız listeler oluşturmak için kullanılır. Liste oluşturma komutlarına örnek Tablo 1.11'de verilmektedir.

Tablo 1.11. Örnek bir liste girdisi ve çıktısı

Kaynak metin	**Çıktı**
Sırasız Liste: \begin{itemize} \item This is one item. \item This is another. \end{itemize} Sıralı Liste: \begin{enumerate} \item This is the first item. \item This is the second. \end{enumerate}	Sırasız Liste: • This is one item. • This is another. Sıralı Liste: 1. This is the first item. 2. This is the second.

1.5.5. Matematiksel Komutlar

LaTeX, özellikle matematiksel ifadeleri dizgilemede (typesetting) oldukça başarılıdır. Diğer kelime işlemcilerden de üstün tarafı budur.

Matematiksel komutlar, ya *"$"* işareti ile matematik kipine girilerek ya da *"\begin{multline}"* ve *"\end{multline}"* komutları kullanılarak kaynak metin içine yazılabilir.

LaTeX'te matematiksel ifadeler, normal metinden ayrıdır. Metin oluşturulurken matematik ortamına geçmek için *"$"* işareti kullanılır. İki dolar arasındaki girilen her şey matematiksel formül olarak yorumlanır. Tablo 1.12'de bu duruma bir örnek verilmektedir.

Tablo 1.12. Matematik kipi örnek girdisi ve çıktısı

Kaynak Metin	**Çıktı**
Aşağıdaki denklem önemlidir: $$E = mc^2$$	Aşağıdaki denklem önemlidir: $E = mc^2$

Tablo 1.13'te bir matematiksel formül örneği ve örnek çıktısı verilmiştir. Burada denklemler, *"&"* boyunca hizalanmakta ve her bir satır *"\\"* ile sonlandırılmaktadır. Ayrıca her bir denklem numaralandırılmaktadır. Denklem numaralandırılması yapılması istenmiyorsa *"align*"* argümanı kullanılır.

Tablo 1.13. Matematik formül girdisi ve çıktısı

Kaynak metin	**Çıktı**
Bazı önemli denklemler: \begin{align} \label{einstein} E &= mc^2 \\ \label{newton} F &= ma \end{align} Lütfen Denklem \ref{einstein} ezberleyin.	Bazı önemli denklemler: $E = mc^2$ (7.1) $F = ma$ (7.2) Lütfen Denklem 7.1 ezberleyin.

Matematik kipindeki komutların çoğu kendisinden sonra gelen ilk karaktere etki ederler. Bir komutun çok sayıda karaktere uygulanmasını istiyorsanız, *"{"* ve *"}"* küme parantezleri kullanılarak onları gruplandırmak gerekir.

Bazı denklemler çok uzun olabilir ve dolaysıyla bir satıra sığmayabilir. Dolaysıyla bu denklemlerin biden fazla satıra bölünmesi gerekir. Bu durum *"\begin{multline}"* ve *"\end{multline}"* komutları arasında formül yazılarak sağlanır. Bölünmenin yapılacağı kısma *"\\"* komutu eklenerek sığdırma işlemi yapılır. Tablo 1.14'te bu duruma örnek verilmektedir.

Tablo 1.14. Denklemlerin satırlara bölme örnek girdisi ve çıktısı

Kaynak metin	**Çıktı**
\begin{multline} a + b + c + d + e + f + g + h + i \\ = j + k + l + m + n \end{multline}	$a+b+c+d+e+f+g+h+i$ $=j+k+l+m+n$

Aşağıda sık kullanılan matematik komutları verilmektedir.

- Üstsimge ve altsimgeler *"^"* ve *"_"* karakterleri kullanılarak yapılır. Eğer altsimge ve üstsimgeler birden fazla karakter koyulacaksa *"{" ve "}"* küme parantezleri alınarak gruplandırılması gerekir.
- Kesirli ifadeler *"\frac{pay}{payda}"* komutuyla elde edilir.
- $\neq$, $\leq$ ve $\geq$ işaretlerinin komutları sırasıyla *"\neq"*, *"\geq"*, ve *"\leq"* dir.
- $+, -, \times, \div$ işaretlerinin komutları sırasıyla *"+"*, *"-"*, *"\times"* ve *"\div"* dir.
- Toplam ve çarpım sembolleri sırasıyla *"\sum"* ve *"\prod"* komutlarıyla yapılır.

1.6. Devre Analiz Yazılımları

Devre analiz yazılımları, girdi olarak verilen devredeki elemanların akım ve gerilim değerlerini hesaplar. Aynı zamanda devre bütünlüğünü kontrol etmek ve verilen parametrelerine göre devrenin davranışını tahmin etme işlemini de gerçekleştirir. Literatüre bakıldığına SPICE isimli devre analiz ve simülasyon programı yer almakta ve diğer geliştirilen gerek ticari gerekse akademik devre analiz ve simülasyon programlarının birçoğu SPICE-tabanlıdır.

1.6.1. SPICE

SPICE (Simulation Program with Integrated Emphasis) genel amaçlı, açık kaynak kodlu bir analog elektronik devre simülatörüdür. Devre tasarımlarının bütünlüğünü kontrol etmek ve girdi olarak verilen parametrelere göre devrenin davranışını tahmin etmek veya incelemek için bütünleşik devre ve devre kartı (board) düzeyinde tasarımda kullanılan gelişmiş bir yazılımdır.

Devre simülasyon programları, devre elemanlarını ve bağlantılarını tanımlayan bir netlist metnini girdi olarak alır ve çözülmesi gereken denklemlere çevirir. Elde edilen denklemler, matris çözüm yöntemleri, Newton yöntemi ve kapalı integrasyon yöntemleri gibi yöntemler kullanılarak çözülür.

SPICE, günümüze kadar gelişimine bağlı olarak SPICE1, SPICE2 ve SPICE3 olarak üç farklı versiyonda sunulmuştur. SPICE1, ilk olarak 1973'teki bir konferansta sunulmuştur. FORTRAN'da kodlanmış olup devre analizini gerçekleştirmek için düğüm analizi (nodal analysis) yöntemini kullanır. Düğüm analizi yöntemi endüktanslar ve kontrollü kaynakların çeşitli formlarını temsil etmekte yetersiz kalmaktadır. Ayrıca SPICE1'de desteklenen devre elemanları çeşidi yetersizdi. Bu yüzden bazı eklemeler ve geliştirmeler yapılarak 1975'te SPICE2 yayınlanmıştır. SPICE2'nin geliştirilmesiyle SPICE yazılımının popülaritesi artmıştır. SPICE2 de yine FORTRAN da kodlanmış olup devre analizini, modifiye düğüm analiz (modified nodal analysis) yöntemini kullanarak daha gelişmiş bir seviyeye getirmiştir. SPICE3 ise 1989'da yayınlanmıştır. SPICE2 gibi aynı ağ listesi (netlist) sözdizimini kullanarak girdi almakta olup farklı olarak C programlama dilinde kodlanmıştır. Aynı zamanda bunlara ek olarak X Window sisteminde grafik çizimi de SPICE3'te yer almıştır.

SPICE, gerek akademik gerekse diğer ticari amaçlı şirketlerde diğer devre simülasyon programları için ya bir temel oluşturmuş ya da bu tip yazılımlara ilham kaynağı olmuştur. SPICE-tabanlı önde gelen devre simülasyon programlarına HSPICE ve PSPICE örnek verilebilir. Akademik alanda geliştirilen yazılıma ise XSPICE örnek gösterilebilir.

Tümleşik devre endüstrisindeki tümleşik devre tasarım yapan şirketler SPICE'yi hızlı bir şekilde benimseyerek ticari yazılımların gelişmişlik düzeyi yeterli olana kadar SPICE'nin çeşitli yazılımlarını kullanmıştır. Şu anda birkaç tümleşik devre üreticisi şirket SPICE-tabanlı programlar geliştirmeye devam etmektedir. Bu programlara örnek olarak Analog Devices şirketinin ADICE ve Texas Instruments'in TISPICE'si örnek gösterilebilir.

1.6.2. Program Özellikleri ve Yapısı

SPICE, zamana bağlı değişen tümleşik devreler tasarlamak için gereken analizler ve modeller içerdiğinden çok tercih edilen bir simülasyon programıdır. SPICE2'nin desteklediği analizler aşağıdaki gibi verilebilir.

- Doğru Akım Analizi: SPICE'nin doğru akım (DC) analiz kısmı endüktanslar kısa devre ve kapasiteler açık devre yapıldığında devrenin DC çalışma notasını belirler. DC analiz seçenekleri *".DC", ".TF"*, ve *".OP"* kontrol satırlarıdır.
- Alternatif Akım analizi
- Doğru Akım Transfer eğri analizi
- Gürültü Analizi
- Transfer fonksiyon analizi

Diğer devre simülatörleri, değişen ve gelişen endüstri gereksinimlerini karşılamak için SPICE2'ye ekstra analiz çeşitleri eklemiştir.

1.6.3. Girdi ve Çıktı

SPICE2, devre elemanları ve bağlantılarından oluşan bir netlisti girdi olarak alır ve çıktıları satır-listelemeleri halinde üretir. Bu çıktılar ya devre elemanlarının hesaplanan gerilim ve akım değerlerine karşılık gelen sayı sütunları ya da satır-yazıcı karakter grafikleridir. SPICE3'te devre tanımı için netlist aynen kullanılır. Farklı olarak C-shelle benzer bir komut satırı arayüzü ile analizlerin kontrolüne izin verilir. SPICE3'te ayrıca X-Windows çizimlerine de yer verilmektedir.

Analizi yapılacak olan devre SPICE'ye devre tanımlamalarını içeren bir ağ listesi (netlist) ile girdi olarak verilir. Netlist, devre topolojisi ve eleman değerlerinin tanımlayan eleman satırlar, model parametreleri ve çalıştırılacak kontrolleri tanımlayan kontrol satırlarından oluşur. Girdi dosyası, bir başlık ile başlamalı ve "*.ENDS*" satırı ile bitmelidir. Kalan diğer satırların düzenlenmesi keyfi olarak yapılabilmektedir.

Devredeki her bir eleman, elemanın adını, bağlanacağı devre düğümlerini ve elemanın elektriksel karakteristiklerini belirleyen parametre değerlerini içeren eleman satırı ile belirtilir. Eleman adının ilk harfi elemanın türünü belirtir. Tablo 1.15'te bazı elemanların

genel sözdizimi ve bir örneği verilmektedir. Diğer elemanları sözdizimleri ve örnekleri [22]'de bulunabilir.

Tablo 1.15. SPICE bazı devre elemanları, sözdizimleri ve örnekleri

Eleman türü	**Sözdizim yapısı ve örnek kullanımı**
Direnç	Sözdizim : Radi n1 n2 değeri Örnek : Rin 2 0 100
Gerilim Kaynakları	Sözdizim : Vname n+ n- <DC<> DC/TRAN VALUE> <AC <ACMAG <ACPHASE>>> <DISTOF1 <F1MAG <F1PHASE>>> <DISTOF2 <F2MAG <F2PHASE>>> Örnek: VCC 10 0 DC 6
Akım Kaynakları	Sözdizim: Iname n+ n- <<DC> DC/TRAN VALUE> <AC <ACMAG <ACPHASE>>> <DISTOF1 <F1MAG <F1PHASE>>> <DISTOF2 <F2MAG <F2PHASE>>> Örnek: Igain 12 15 DC 1

Bir satırdaki alanlar bir daha fazla boşlukla, virgülle ve eşit işareti veya sol veya sağ parantez ile ayrılır. Bir satıra *"+"* yazılarak takip eden satırın birinci sütunuyla devam edilebilir. Bir eleman adı bir harfle başlamak zorundadır ve herhangi bir sınırlayıcı (delimeter) içeremez.

Tablo 1.16. SPICE sayı çarpanı listesi

$T=10^{12}$	$T=10^{9}$	$Meg=10^{6}$
$k=10^{3}$	$m=10^{-3}$	$u=10^{-6}$
$n=10^{-9}$	$p=10^{-12}$	$f=10^{-15}$

Sayı alanı, tamsayı (22 veya -54), kayan noktalı sayı (3.2321 veya 1e-14 veya 2.65e3) veya bu sayıları takip eden ölçekleme çarpanlarından birini içerebilir. Tablo 1.16'da bu çarpanlar listesi verilmektedir.

Düğümler, keyfi bir karakter stringi olabilir. Toprak *"0"* olmak zorundadır. SPICE3'te düğümler karakter stringleri olarak işlendiğinden *"0"* ile *"00"* düğümleri farklı olarak yorumlanır. Devre gerilim kaynakları ve/veya endüktansların bir çevresini (loop) ve

akım kaynakları ve/veya kapasitelerin bir kesi kümesini içeremez. Devredeki her bir düğümün toprağa bir yolu olmalıdır. Ayrıca her bir düğümün en az iki bağlantısı olmalıdır.

Çoğu basit devre elemanı tipik olarak birkaç parametre değerinden oluşur. Fakat SPICE'de yer alan bazı cihazlar (devices) çok fazla parametre değeri gerektirir. Genelde devredeki çoğu cihaz aynı cihaz model parametreleriyle tanımlanır. Bu nedenlerden dolayı bir takım cihaz model parametreleri ayrı bir *".MODEL"* satırında benzersiz bir isim verilerek tanımlanır.

SPICE'de benzer devre elemanlarından oluşan altdevreler (subcircuits) tanımlanıp refereans olarak alınabilir. Altdevreler, eleman satırları gruplandırılarak girdi dosyasında tanımlanır. Program altdevrenin referans alındığı yere eleman gruplarını otomatik olarak yerleştirir. Altdevrelerin boyutunda ve karmaşıklığında herhangi bir sınırlama yoktur. Altdevre tanımlamasının genel ifadesi *".SUBCKT subnam N1 &ltN2; N3 ...>"* gibi verilebilir. Altdevrenin tanımına *".SUBCKT*" satırıyla başlanır. *"SUNBNAM"* altdevrenin ismi ve *"N1, N2..."* ise sıfır olmayan düğümleridir. *".SUBCKT"* satırının akabinde altdevreyi tanımlayan elemanların satır gruplarına yer verilir. Altdevre *".ENDS"* satırıyla sonlandırılır. X harfiyle başlayan yalancı-elemanlar (pseudo-elements) belirtilerek ve ardından genişletilen altdevrede kullanılacak devre düğümleri verilerek altdevreler SPICE'de kullanılır.

Devre tanımlamalarının bazı kısımları diğer birkaç girdi dosyasında özellikle ortak model ve altdevrelerde sıkça kullanılabilir. Herhangi bir SPICE girdi dosyasında *".INCLUDE"* satırı, diğer girdi dosyalarının içeriğini kopyalamak için kullanılabilir. *".INCLUDE"* satırının genel formu *".INCLUDE dosyaismi"* gibi verilebilir.

Bazı ticari yazılımlarda ve çeşitli ücretsiz yazılım projelerinde SPICE'ye şematik çizim ön-yüzleri eklenmiştir. Bu sayede kullanıcını devrenin şematik çizimini yapması sonucu, devreye karşılık gelen netlistin otomatik üretimi yapılır. Ayrıca yapılacak olan simülasyonların seçimi ve akım ve gerilim çıktılarını değiştirmek için grafiksel kullanıcı ara yüzleri de eklenmiştir. Bunu yanında dalga şekilleri ve parametre bağımlılık grafiklerini de görmek için çeşitli grafik araçları da bu yazılımlara eklenmiştir.

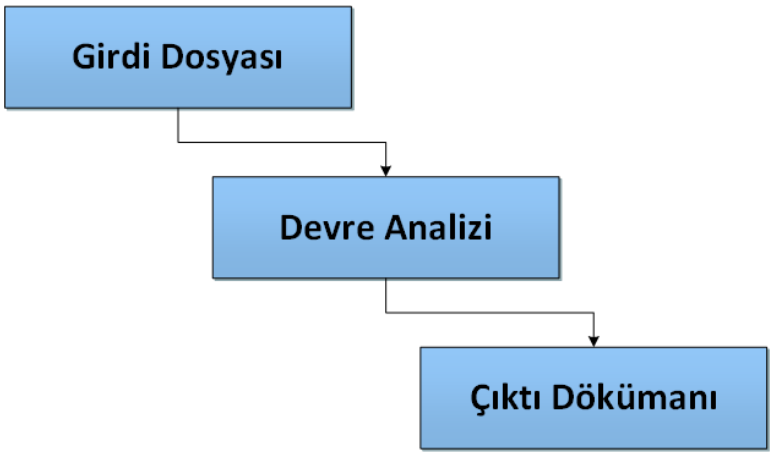
2. YAPILAN ÇALIŞMALAR, BULGULAR VE İRDELEME

2.1. Giriş

Yapılan bu çalışmada, doğru akım elektrik devrelerinin analizini yapan bir yazılım geliştirilmiştir. Bu yazılım verilen bir elektrik devresinin analizini yaparak, ara işlemlerini adım adım içeren bir çıktı dokümanı üretmektedir.

Analiz yapılacak devrede elemanlar sadece bağımsız gerilim kaynağı, akım kaynağı ve dirençle sınırlandırılmıştır. Böylece basit bir elektrik devresini oluşturan elemanlar kullanılarak yazılım hazırlanmıştır.

Şekil 2.1'de de gösterildiği gibi yazılım üç kısımdan oluşmaktadır. Bu aşamalar bir devrenin girdi olarak alınması, verilen devrenin analizinin yapılarak devre elemanlarının akım ve gerilim değerlerinin belirlenmesi ve devre çözümünde yapılan ara işlemleri de adım adım belirten çıktı dokümanın üretilmesidir.



Şekil 2.1. Devre analiz yazılımının aşamaları

2.2. Girdi Dosyası

Elektrik devre analizi yapılabilmesi için devreyi temsil eden girdinin kullanıcıdan alınması gerekir. Burada kullanıcıdan girdi alınması, tanımlanan bir Devre Girdi Dili (DGD) ile gerçekleştirilmektedir.

Tanımlanan Devre Girdi Dili (DGD) üç bloktan oluşmaktadır. Dilin birinci bloğunda devreye bağlanacak elemanlar belirtilir. İkinci bloğunda, devreye bu elemanların bağlantısını yapan ifadeler yazılır. Son bloğunda ise devreye bağlanan elemanların değerleri verilir. DGD'nin üç bloğu Tablo 2.1'deki gibidir.

Tablo 2.1. Devre Girdi Dili blokları

```
circuit
{       elements {
                Burada devre elemanları tanımlanır.
        }
        connections {
                Elemanların bağlantıları yapılır
        }
        parameters {
                Elemanlara belirli değerler verilir.
        }
}
```

Devre girdi dilinde *"elements"* bloğunda dört çeşit eleman tanımlanır. Bunlar; direnç (resistor), bağımsız gerilim kaynağı (voltagesource), bağımsız akım kaynağı (currentsource) ve düğümlerdir (node). Tanımlanan elemanlar, eleman tipi ve değişken adı şeklinde iki kısımdan oluşur. Eleman tipleri direnç, gerilim kaynağı, akım kaynağı ve düğümler için sırasıyla *"resistor", "voltagesource", "currentsource"* ve *"node"* tur. Değişken isimleri tanımlanırken, anlaşılır bir doküman üretimi için direnç, gerilim kaynağı, akım kaynağı ve düğüm için sırasıyla *"R"*, *"V"*, *"I"* ve *"N"* harfleriyle başlaması zorunluluğu getirilmiştir. Sonrasında herhangi bir rakam veya harf gelebilir. Bu kurallar dikkate alınarak tanımlanan dilin gramerini EBNF formunda Tablo 2.2'deki gibi verilmektedir.

DGD'de *"connections"* bloğunda, tanımlanan elemanların bağlantısı yapılır. Bağlantılar her bir devre elemanının düğümlere bağlantısı yapılarak sağlanır. Eğer devreye toprak bağlantısı yapılacaksa *"gnd"* kullanılarak devrede bir düğüme bağlanır.

Devre bağlantısı yapılırken argüman olarak verilen düğümlerin sırası, analizde kullanılacak olan akım yönünü belirtmek için kullanılır. Direnç ve akım kaynağı için akım yönünün birinci düğümden ikinci düğüme ve gerilim kaynağı için *"+"* ucundan *"-"* ucuna doğru olacak şekilde ele alınır.

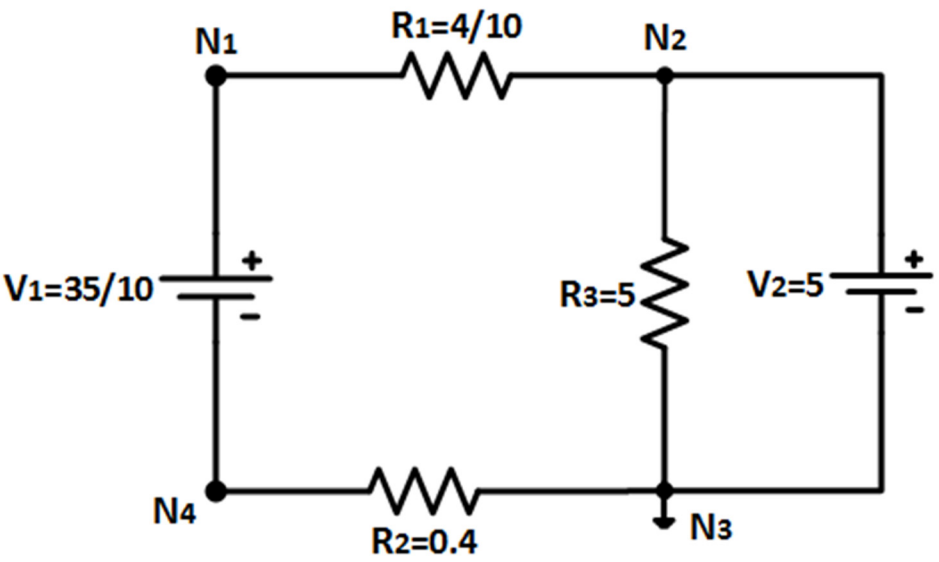
Tablo 2.2. Devre Girdi Dilinin EBNF gramerinin bir kısmı

```
RESISTOR_ID ::= "R" ((["a"-"z"])*  (["A"-"Z"])*  (["0"-"9"])*)+
VOL_SRC_ID ::= "V" ((["a"-"z"])*  (["A"-"Z"])*  (["0"-"9"])*)+
CUR_SRC_ID ::= "I" ((["a"-"z"])*  (["A"-"Z"])*  (["0"-"9"])*)+
NODE_ID ::= "N" ((["a"-"z"])*  (["A"-"Z"])*  (["0"-"9"])*)+
GND ::= ("G" | "g") ("N" | "n") ("D" | "d")
CIRCUIT ::= “circuit”
LCB ::= “{”
RCB ::= “}”
LB ::= “(”
RB ::= “)”
COMMA ::= “,”
SECOMMA ::= “;”
DIV ::= “/”
MINUS ::= “-”
EQ ::= “=”
ELEMENTS ::= “elements”
CONNECTIONS ::= “connections”
PARAMETERS ::= “parameters”
RESISTOR ::= “resistor”
VOLSOURCE ::= “voltagesource”
CURSOURCE ::= “currentsource”
NODE ::= “node”
```

Devre dilinin son bloğu olan *“parameters”* bloğunda devrenin asıl elemanları olan direnç, gerilim kaynağı ve akım kaynağının değerleri verilir. Değerler; tamsayı, gerçel sayı veya rasyonel sayı şeklinde girilebilir. Şekil 2.2’deki devre şematiğinin devre girdi dili (DGD) ile kodlanmış hali Şekil 2.3’te gösterilmektedir.

Devre girdi dilini (DGD) kelimesel, sözdizimsel ve anlamsal yönden analiz yapacak kod yazılması gerekir. Bu işlemlerin yapılması için Java programlama dili ve JavaCC otomatik ayrıştırıcı üreteci kullanılmıştır.

Devre analizi aşamasına geçmeden önce tanımlanan devre girdi dilinde bazı ön işlemler yapılmalıdır. Bu işlemler, girdi dosyasına yazılan karakter veya kelimelerin dilde yer alıp almadığı, aynı şekilde uygun bir sırada yazılıp yazılmadığı ve aynı zamanda devrede tekrarlı eleman olup olmadığı, devreye bağlanan devre elemanlarına değer verilip verilmediği gibi anlamsal kontrolünün yapılmasıdır.



Şekil 2.2. Gerilim kaynaklarından oluşan örnek bir elektrik devresi

Devre girdi dilinin bağlam bağımsız gramere uygun yazımı Tablo 2.3'teki gibidir. Gramerde hepsi büyük harf olan kelimeler terminal diğerleri ise non-terminaldir.

Tablo 2.3. Devre Girdi Dilinin gramer kuralları

Parse ::= CIRCUIT LCB ElementsBlock ConnectionsBlock ParametersBlock RCB

ElementsBlock ::= ELEMENTS LCB (ResDec | VolSrcDec | CurSrcDec | NodeDec) RCB*

ResDec ::= RESISTOR RESISTOR_ID (COMMA RESISTOR_ID) SECOMMA*

VolSrcDec ::= VOLSOURCE VOL_SRC_ID (COMMA VOL_SRC_ID) SECOMMA*

CurSrcDec ::= CURSOURCE CUR_SRC_ID (COMMA CUR_SRC_ID) SECOMMA*

NodeDec ::= NODE NODE_ID (COMMA NODE_ID) SECOMMA*

ConnectionsBlock ::= CONNECTIONS LCB (RESISTOR_ID | VOL_SRC _ID | CUR_SRC_ID) LB NODE_ID COMMA NODE_ID RB SECOMMA) (GND LB NODE_ID RB SECOMMA)*

*ParametersBlock ::= PARAMETERS LCB (RESISTOR_ID | VOL_SRC_ID | CUR_SRC_ID EQ RatNum)**

RatNum ::= Num (DIV Num)?

Num ::= (MINUS NUMBER | NUMBER)

Sözdizimsel analiz için gramer belirlendikten sonra bu yapının LL(k)'ya uygun hale getirilmesi gerekir. Çünkü JavaCC, LL(k) algoritması ile çalışır. Bu nedenle gramerin, LL(k) algoritmasına uygun olabilmesi için, soldan yinelemeli durumları elimine edilmiştir ve sol faktörleme (left factoring) yapılmıştır. Tüm bu işlemler yapıldıktan sonra gramer JavaCC ortamına aktarılmış ve devre girdi dilinin sözdizimsel analizini yapabilecek Java kodu üretilmiştir. Tablo 2.4'te, Tablo 2.3'te verilen gramer için yazılan JavaCC gramer dosyasının bir bölümü verilmektedir. Tamamı Ek-1'dedir.

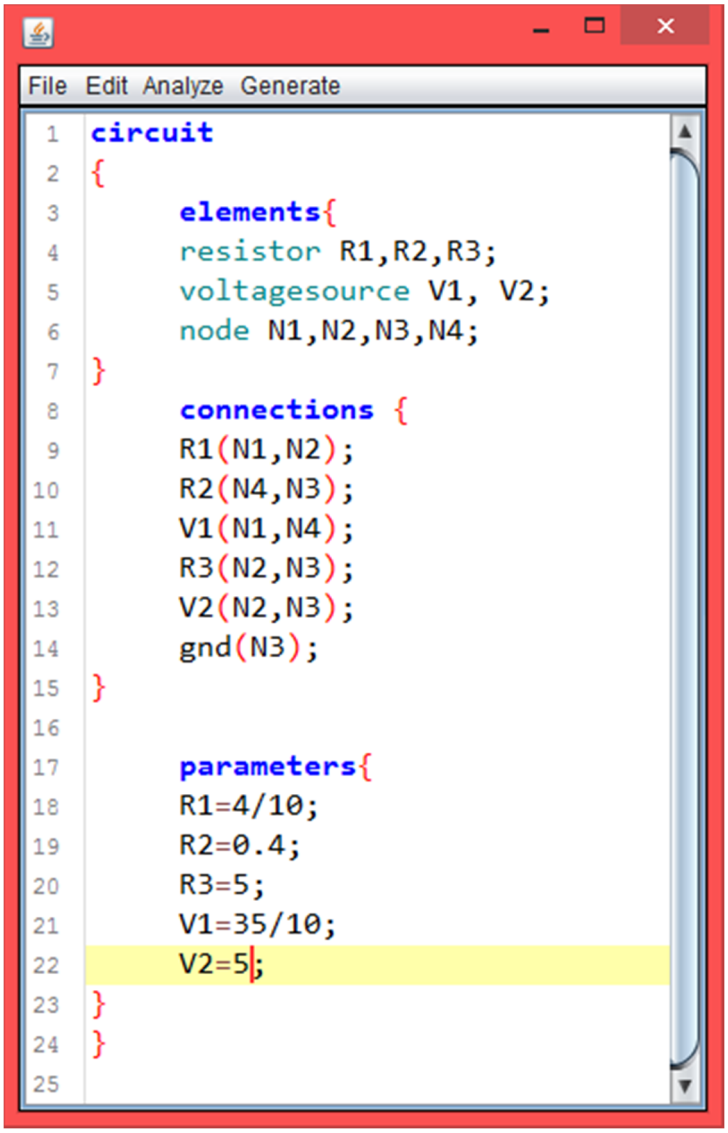
Tablo 2.4. Devre Girdi Dilinin JavaCC gramerinin bir kısmı

```
...
void parse() throws ElementAlreadyDeclaredException,
ElementNotDeclaredException : { }
{
   "circuit" "{" elementsBlock() connectionsBlock() parametersBlock() "}"
<EOF>
}
void elementsBlock() throws ElementAlreadyDeclaredException : { }
{
  "elements" "{"
   ( "resistor" resistor() ( "," resistor() )* ";"
  | "voltagesource"  voltageSource() ( "," voltageSource() )* ";"
  | "currentsource"  currentSource() ( "," currentSource() )* ";"
  | "node" node() ( "," node() )* ";"
   )*
    "}"
}
void resistor() throws ElementAlreadyDeclaredException: { Token id; }
{     id=<RESISTOR_ID> { circuit.addElement(id.image,new
Resistor(id.image)); }
}
...
```

Devre girdi dilinde sözdizimsel analizin yanı sıra, anlamsal analizi de yapacak olan Java kodu JavaCC gramer dosyasına eklenmiştir. Anlamsal analizde aşağıdaki durumlara dikkat edilmiştir.

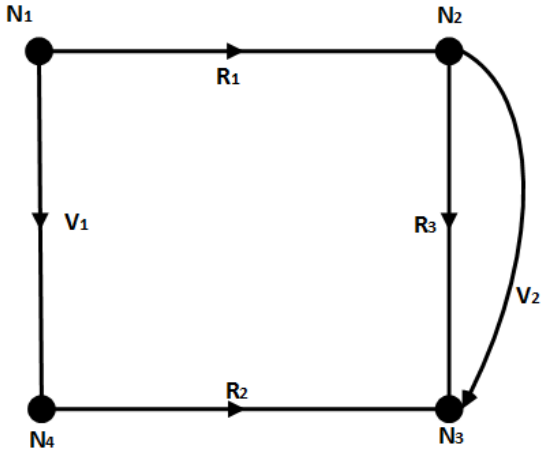
- Devre değişkenleri tanımlanırken her bir değişkenin adlandırılmasının benzersiz yapılıp yapılmadığı: Bu işlem *"elements"* bloğunda tanımlanan değişkenler bir sembol tablosuna aktarılarak kontrol edilmiştir.
- Bağlantısı yapılan değişkenlerin *"elements"* bloğunda tanımlanıp tanımlanmadığı.

- *"parameters"* bloğunda değeri atanan değişkenlerin devrede bağlantısının olup olmadığı: Bir değişken aslında *"elements"* bloğunda tanımlanıp devreye bağlantısı yapılmamış olabilir. Bu durumda o elemana değer atanması bir hata üretir.



Şekil 2.3. Şekil 2.2'deki örnek devrenin DGD ile kodlanmış hali

Devre girdi diliyle tanımlanmış bir devrenin girdi olarak verilmesi sonucu kelimesel, sözdizimsel ve anlamsal analizden başarı ile geçen dil artık bir graf veri yapısıyla temsil edilir. Bundan sonra girdi olarak verilen bu devrenin analizi veya çözümü yapılır. Örneğin Şekil 2.2'deki elektrik devresinin yönlü bir grafla temsili ve bu yönlü grafa karşılık gelen ilişki matrisi Şekil 2.4'te verilmektedir.



$$A_c = \begin{array}{c} \\ N_1 \\ N_2 \\ N_3 \\ N_4 \end{array} \begin{array}{c} \begin{array}{ccccc} R_1 & R_2 & R_3 & V_1 & V_2 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ -1 & 0 & 1 & 0 & 1 \\ 0 & -1 & -1 & 0 & -1 \\ 0 & 1 & 0 & -1 & 0 \end{bmatrix} \end{array}$$

Şekil 2.4. Şekil 2.2'deki devreyi temsil eden yönlü graf ve onun ilişki matrisi

2.3. Devrenin Analizi

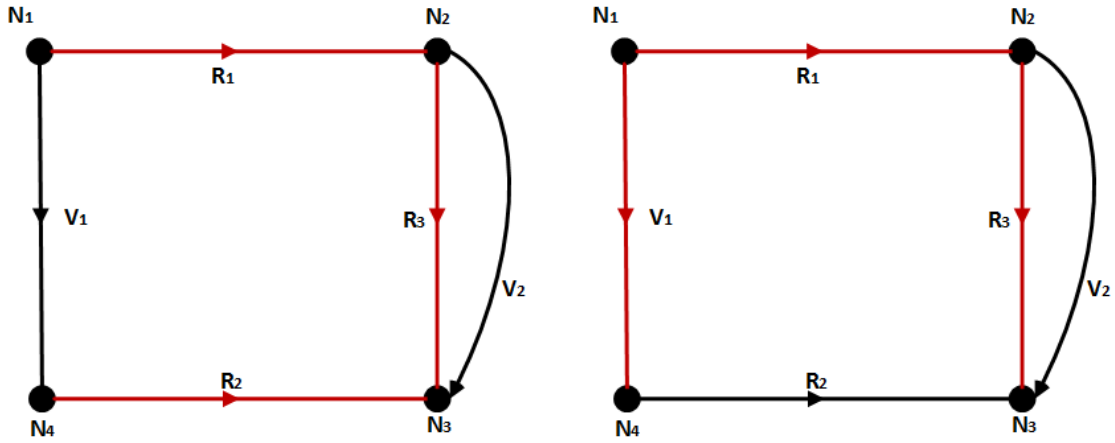
Girdi olarak verilen bir devre graf veri yapısıyla temsil edildikten sonra graf analiz yöntemiyle devre elemanlarının akım ve gerilim değerleri hesaplanır.

Devrede analiz aşamasının başlangıcında üç kontrol yapılır. Bunlar;

- Graf analiz yönteminin temel taşı olan grafın bağlantılı olup olmadığı kontrolü yapılır. Eğer bağlantılı değilse hata mesajı üretilir. Grafın bağlantılı olup olmadığı graf bağlantılılık testi algoritmasıyla rahatlıkla gerçekleştirilebilir.
- Devrede herhangi bir akım veya gerilim kaynağı olup olmadığına bakılır. Eğer yoksa hata mesajı üretilir. (Bu durum eşdeğer devre hesaplama için geçerli değildir).
- Devredeki kaynakların çeşitlerine bakılır. Eğer sadece akım kaynağı varsa graf-kesi kümesi analiz yöntemi kullanılır. Eğer sadece gerilim kaynağı varsa graf-çevre analiz yöntemi kullanılır. Eğer her iki çeşitten de kaynak varsa devre analizine devam edilmez.

Verilen devreye karşılık gelen grafta, başlangıç olarak bir referans düğümüne göre grafın bir açılım ağacı hesaplanır. Burada eğer devreye bir toprak bağlantısı yapılmışsa, referans düğüm olarak toprağın bağlı olduğu düğüm alınır. Eğer toprak bağlantısı yapılmamışsa o zaman devreye bağlantısı olan ilk düğüm alınarak açılım ağacı hesaplanır. Açılım ağacının hesaplanmasında graflarda arama algoritmalarından olan enine arama algoritması kullanılmıştır.

Şekil 2.2'deki devreyi temsil eden Şekil 2.4'teki ilişki matrisi örnek olarak ele alınırsa, açılım ağacının enine arama algoritmasıyla hesaplanmış hali Şekil 2.5(a)'daki gibidir. Şekil 2.2'deki devreye bir toprak (ground) bağlandığından, bu düğüm referans alınarak açılım ağacı hesaplanmıştır. Eğer devreye toprak bağlanmayıp referans düğüm belirtilmeseydi o zaman *"elements"* kısmında ilk tanımlanan düğüm olan *"N_1"* ele alınacak ve açılım ağacı Şekil 2.5(b)'deki gibi olacaktı.



Şekil 2.5. Şekil 2.3'teki grafın mümkün açılım ağaçları (a) Toprak bağlı ve N_3 iken açılım ağacı (b) Toprak bağlı değilken açılım ağacı

Devrenin açılım ağacı belirli bir düğüme göre hesaplandıktan sonra, açılım ağacında yer almayan kenarlar, kirişlerdir. Şekil 2.5(a)'daki açılım ağacına göre kirişler *"V_1"* ve *"V_2"* kenarlarıdır. Açılım ağacı kenarları, kesi kümesi matrisinin satırlarını ve kirişler ise çevrim matrisinin satırlarını oluşturacak şekilde düzenlenir.

B_f temel çevrim matrisinin elde edilmesinde kullanılan A_t temel ilişki matrisi, referans düğüm ilişki matrisinden kaldırılarak elde edilir. Örneğin Şekil 2.3'teki ilişki matrisinin A_t temel ilişki matrisi, referans düğüm olan *"N_3"* düğümüne karşılık gelen matris satırı çıkarılarak (2.1)'deki gibi elde edilir.

$$A_t = \begin{array}{c} \\ N_1 \\ N_2 \\ N_4 \end{array} \begin{array}{c} \begin{array}{ccccc} R_1 & R_2 & R_3 & V_1 & V_2 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ -1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & -1 & 0 \end{bmatrix} \end{array} \tag{2.1}$$

A_t temel ilişki matrisi, ilk sütunları kirişler ve diğerleri açılım ağacındaki kenarlar olacak şekilde yeniden düzenlenirse (2.2)'deki A matrisi elde edilir.

$$A = \begin{bmatrix} A_{11} & A_{12} \end{bmatrix} = \begin{array}{c} \\ N_1 \\ N_2 \\ N_4 \end{array} \begin{array}{c} \begin{array}{ccccc} V_1 & V_2 & R_3 & R_2 & R_1 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & -1 \\ -1 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array} \tag{2.2}$$

B_f temel çevrim matrisi, A matrisinin altmatrisleri olan A_{11} ve A_{12} kullanılarak (2.3)'teki gibi elde edilir. Burada verilen örnek devre için A_{11} altmatrisinin sütun sayısı 2 ve A_{12} altmatrisinin sütun sayısı ise 3'tür. B_{f11} altmatrisi birim matristir.

$$B_f = \begin{bmatrix} B_{f11} & B_{f12} \end{bmatrix} \tag{2.3}$$

Burada B_{f12} altmatrisi (2.4)'teki gibi hesaplanır.

$$B_{f12} = -A_{11}^T A_{12}^{-1} \tag{2.4}$$

(2.3) ve (2.4)'teki denklemlere göre B_f matrisi yeniden düzenlenirse (2.5)'teki sonuç matris elde edilir.

$$B_f = \begin{bmatrix} B_{f11} & B_{f12} \end{bmatrix} = \begin{array}{c} \\ V_1 \\ V_2 \end{array} \begin{array}{c} \begin{array}{ccccc} V_1 & V_2 & R_3 & R_2 & R_1 \end{array} \\ \begin{bmatrix} 1 & 0 & -1 & 1 & -1 \\ 0 & 1 & -1 & 0 & 0 \end{bmatrix} \end{array} \tag{2.5}$$

B_f matrisine göre yazılan graf çevre denklemleri (2.6) ve (2.7)'deki gibidir. Bu denklemler KVL denklemlerine karşılık gelen denklemlerdir.

$$V_1 - V_{R_3} + V_{R_2} - V_{R_1} = 0 \tag{2.6}$$

$$V_2 - V_{R_3} = 0 \tag{2.7}$$

Devreye bağlı bağımsız gerilim kaynaklarının değerlerinde oluşan E gerilim kaynağı matrisi (2.8)'deki gibi yazılır.

$$E = \begin{bmatrix} 7/2 \\ 5 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.8}$$

Z empedans matrisi ise devre sadece dirençlerden oluştuğundan Laplace dönüşümüne gerek kalmadan doğrudan (2.9)'daki gibi yazılır.

$$Z = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 2/5 & 0 \\ 0 & 0 & 0 & 0 & 2/5 \end{bmatrix} \tag{2.9}$$

Temel çevrim matrisi kullanılarak gerilim kaynağı matrisi ve empedans matrislerindeki gereksiz satır veya sütunlar elenir. Bu işlemler (2.10) ve (2.11)'de verilmektedir.

$$E_m = -B_f E = \begin{bmatrix} -7/2 \\ -5 \end{bmatrix} \tag{2.10}$$

$$Z_m = B_f Z B_f^T = \begin{bmatrix} 29/5 & 5 \\ 5 & 5 \end{bmatrix} \tag{2.11}$$

Ohm kanununa göre (2.12) denklemi yazılabildiğinden ve ayrıca E_m ve Z_m'nin de değerleri belli olduğundan I_m rahatlıkla hesaplanabilir. Z_m, E_m üzerinde çözülürse I_m (2.13)'teki gibi olur.

$$E_m = I_m Z_m \tag{2.12}$$

$$I_m = \begin{bmatrix} \frac{15}{8} \\ \frac{-23}{8} \end{bmatrix} \tag{2.13}$$

Tüm kollardaki akımların oluşturduğu I matrisi, B_f temel çevrim matrisi ve I_m kullanılarak (2.14)'teki gibi hesaplanır. I matrisindeki satırlar, B_f matrisindeki sütun sırasına göre düzenlenmiştir ve bu devre elemanları sırasıyla *"V_1, V_2, R_3, R_2, R_1"*dir.

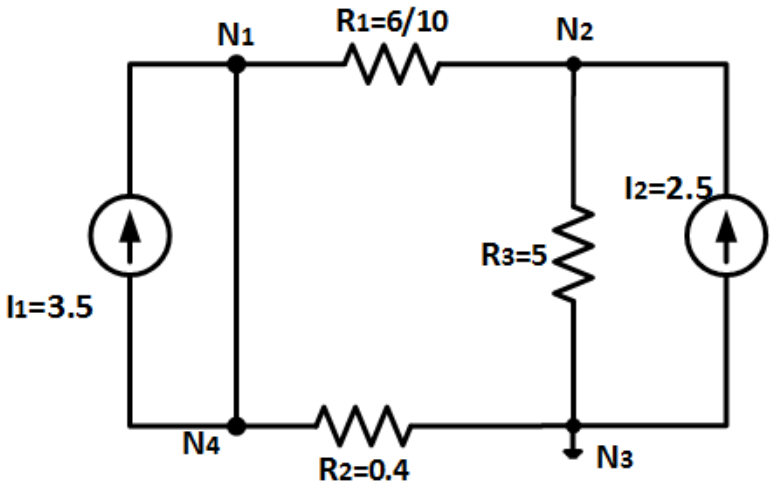
$$I = B_f^T I_m = \begin{bmatrix} 15/8 \\ -23/8 \\ 1 \\ 15/8 \\ -15/8 \end{bmatrix} \tag{2.14}$$

Z empedans matrisi ve I akım matrisinin değerleri artık belli olduğundan tüm kenarlardaki gerilimlere karşılık gelen V matrisi, Ohm kanununa göre (2.15)'teki gibi hesaplanır. V matrisindeki satırların değerleri sırasıyla "V_1, V_2, R_3, R_2, R_1" kenarlarına karşılık gelmektedir.

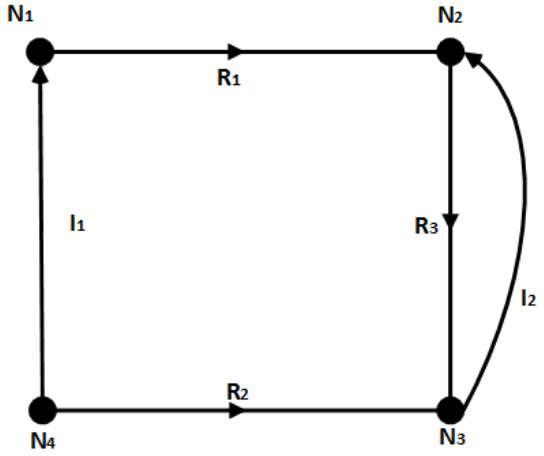
$$V = ZI = \begin{bmatrix} 0 \\ 0 \\ 5 \\ 3/4 \\ -3/4 \end{bmatrix} \tag{2.15}$$

Graf-kesi analizi yapılırken graf-çevre analizine benzer işlemler yapılır. Graf-kesi analizinde farklı olarak devrede kaynak olarak akım kaynakları bulunmalıdır. Ayrıca empedans matrisi yerine geçiri (admittance) matrisi ve temel çevrim matrisi yerine temel kesi matrisi kullanılarak denklemler oluşturulur ve bu denklem sistemleri çözülür. Bu işlemler bir örnek üzerinde adım adım açıklanmak istenirse Şekil 2.6'daki devre dikkate alınarak graf-kesi analizine göre aşağıdaki matrisler hesaplanır.

Şekil 2.6'daki devrenin yönlü graf gösterimi ve bu grafa karşılık gelen A_c ilişki matrisi Şekil 2.7'de verilmektedir. Graf-çevre analizinde olduğu gibi graf-kesi analizinde de açılım ağacı kenarları ve kirişler hesaplanır. Şekil 2.7'deki yönlü grafa göre açılım ağacı toprağın bağlı olduğu *"N_3"* düğümüne göre hesaplandığında bulunan kenarlar *"R_3, R_2, R_1"* ve kirişler ise *"I_1"* ve *"I_2"* olur. Açılım ağacı kenarları, kesi matrisinin satırlarını ve kirişler ise çevrim matrisinin satırlarını oluşturacak şekilde düzenlenir.



Şekil 2.6. Akım kaynaklarından oluşan örnek bir elektrik devresi



$$A_c = \begin{array}{c} \\ N_1 \\ N_2 \\ N_3 \\ N_4 \end{array} \begin{array}{c} \begin{array}{ccccc} R_1 & R_2 & R_3 & I_1 & I_2 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & -1 & 0 \\ -1 & 0 & 1 & 0 & -1 \\ 0 & -1 & -1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Şekil 2.7. Şekil 2.6'daki devreyi temsil eden yönlü graf ve onun ilişki matrisi

B_f temel çevrim matrisinin elde edilmesinde kullanılan A_t temel ilişki matrisi, referans düğüm ilişki matrisinden kaldırılarak elde edilir. Örneğin Şekil 2.3'teki ilişki matrisinin temel ilişki matrisi, referans düğüm olan *"N3"* düğümüne karşılık gelen matris satırı çıkarılarak (2.16)'deki temel ilişki matrisi elde edilir.

$$A_t = \begin{array}{c} \\ N_1 \\ N_2 \\ N_4 \end{array} \begin{array}{c} \begin{array}{ccccc} R_1 & R_2 & R_3 & I_1 & I_2 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & -1 & 0 \\ -1 & 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} \end{array} \qquad (2.16)$$

A_t temel ilişki matrisi, ilk sütunları kirişler ve diğerleri açılım ağacındaki kenarlar olacak şekilde yeniden düzenlenirse (2.17)'deki matris elde edilir.

$$A = \begin{bmatrix} A_{11} & A_{12} \end{bmatrix} = \begin{array}{c} \\ N_1 \\ N_2 \\ N_4 \end{array} \begin{array}{c} \begin{array}{ccccc} I_1 & I_2 & R_3 & R_2 & R_1 \end{array} \\ \begin{bmatrix} -1 & 0 & 0 & 0 & 1 \\ 0 & -1 & 1 & 0 & -1 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array} \tag{2.17}$$

B_f temel çevrim matrisi, A matrisinin altmatrisleri olan A_{11} ve A_{12} kullanılarak (2.18)'teki gibi elde edilir. Burada verilen örnek devre için A_{11} altmatrisinin sütun sayısı 2 ve A_{12} altmatrisinin sütun sayısı ise 3'tür. B_{f11} altmatrisi birim matristir. Q_f temel kesi matrisinin hesaplanılması ya B_f temel çevrim matrisi ya da A matrisinin altmatrisleri olan A_{11} ve A_{12} kullanılarak yapılır.

$$B_f = \begin{bmatrix} B_{f11} & B_{f12} \end{bmatrix} \tag{2.18}$$

Burada B_{f12} altmatrisi (2.19)'teki gibi hesaplanır.

$$B_{f12} = -A_{11}^T A_{12}^{-1} \tag{2.19}$$

(2.18) ve (2.19)'daki denklemlere göre B_f matrisi yeniden düzenlenirse (2.20)'deki sonuç matris elde edilir.

$$B_f = \begin{bmatrix} B_{f11} & B_{f12} \end{bmatrix} = \begin{array}{c} \\ I_1 \\ I_2 \end{array} \begin{array}{c} \begin{array}{ccccc} I_1 & I_2 & R_3 & R_2 & R_1 \end{array} \\ \begin{bmatrix} 1 & 0 & -1 & 1 & -1 \\ 0 & 1 & -1 & 0 & 0 \end{bmatrix} \end{array} \tag{2.20}$$

Q_f temel kesi matrisi, (2.21)'deki gibi yazılır. Burada Q_{f12} altmatrisi birim matristir.

$$Q_f = \begin{bmatrix} Q_{f11} & Q_{f12} \end{bmatrix} \tag{2.21}$$

Burada Q_{f11} altmatrisi, B_{f12} kullanılarak (2.22)'deki gibi hesaplanır.

$$Q_{f11} = -B_{f12}^{T} \tag{2.22}$$

(2.21) ve (2.22) denklemlerine göre Q_f matrisi hesaplanırsa (2.23)'teki matris elde edilir.

$$Q_f = \begin{bmatrix} Q_{f11} & Q_{f12} \end{bmatrix} = \begin{array}{c} \\ R_3 \\ R_2 \\ R_1 \end{array} \begin{array}{c} \begin{array}{ccccc} I_1 & I_2 & R_3 & R_2 & R_1 \end{array} \\ \begin{bmatrix} -1 & -1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \tag{2.23}$$

Q_f matrisine göre yazılan graf kesi denklemleri aşağıdaki gibidir. Bu denklemler KCL denklemlerine karşılık gelen denklemlerdir.

$$-I_1 - I_2 + I_{R_3} = 0 \tag{2.24}$$

$$I_1 + I_{R_2} = 0 \tag{2.25}$$

$$-I_1 + I_{R_1} = 0 \tag{2.26}$$

J akım kaynağı matrisi (2.27)'deki gibi yazılır.

$$J = \begin{bmatrix} 7/2 \\ 5/2 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.27}$$

Y geçiri (admittance) matrisi ise devre sadece dirençlerden oluştuğundan Laplace dönüşümüne gerek kalmadan doğrudan (2.28)'deki gibi yazılır.

$$Y = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/5 & 0 & 0 \\ 0 & 0 & 0 & 5/2 & 0 \\ 0 & 0 & 0 & 0 & 5/3 \end{bmatrix} \tag{2.28}$$

Q_f temel kesi matrisi kullanılarak I akım kaynağı matrisi ve Y geçiri matrislerindeki gereksiz satır veya sütunlar elenir. Bu işlemler (2.29) ve (2.30)'da verilmektedir.

$$J_c = -Q_f J = \begin{bmatrix} 6 \\ -7/2 \\ 7/2 \end{bmatrix} \tag{2.29}$$

$$Y_c = Q_f Y Q_f^T = \begin{bmatrix} 1/5 & 0 & 0 \\ 0 & 5/2 & 0 \\ 0 & 0 & 5/3 \end{bmatrix} \tag{2.30}$$

Ohm kanununa göre (2.31) denklemi yazılabildiğinden ve J_c ve Y_c'nin de değerleri belli olduğundan V_c rahatlıkla hesaplanabilir. Y_c , J_c üzerinde çözülürse V_c'nin sonucu (2.32)'deki gibi olur.

$$J_c = V_c Y_c \tag{2.31}$$

$$V_c = \begin{bmatrix} 30 \\ -7/5 \\ 21/10 \end{bmatrix} \tag{2.32}$$

Tüm kenarlardaki kenarlara karşılık gelen V matrisi, Q_f temel kesi matrisi ve V_c kullanılarak (2.33)'teki gibi hesaplanır. V matrisindeki satırlar, Q_f matrisindeki sütun sırasına göre düzenlenmiştir ve bu devre elemanları sırasıyla *"I_1, I_2, R_3, R_2, R_1"*dir.

$$V = Q_f^T V_c = \begin{bmatrix} -67/2 \\ -30 \\ 30 \\ -7/5 \\ 21/10 \end{bmatrix} \tag{2.33}$$

Y geçiri matrisi ve V gerilim matrisinin değerleri artık belli olduğundan I tüm kenarlardaki akımlar Ohm kanununa göre (2.34)'teki gibi hesaplanır. I matrisindeki satırların değerleri sırasıyla *"I_1, I_2, R_3, R_2, R_1"* kenarlarına karşılık gelir.

$$I = YV = \begin{bmatrix} 0 \\ 0 \\ 6 \\ -7/2 \\ 7/2 \end{bmatrix} \tag{2.34}$$

Yukarıda anlatılan gerek graf-çevre analizi gerekse graf-kesi analizi yönteminin kodu *GraphAnalysis.java* dosyasında bulunmaktadır ve kodu Ek-2'de verilmektedir.

Devre analizinde JLinAlg [23] kütüphanesi kullanılmıştır. Bu kütüphane rasyonel sayı temelli hesaplamalar yaptığından dolayı gerçel sayılardan dolayı oluşan kesme hataları oluşmaz. Ayrıca çıktı dokümanı üretilirken kullanıcıya okuma kolaylığı sağlayacak rasyonel sayılar rahatlıkla gösterilmektedir.

Ayrıca sözdizim renklendirmesi için RSyntaxArea [24] Java kütüphanesi kullanılmıştır. Bu kütüphaneye dil sözdizim kurallarına uygun bazı kod eklemeleri yapılarak renklendirme ve otomatik anahtar kelime önerme özelliği eklenmiştir.

2.4. Çıktı Dokümanının Üretilmesi

Devre analiz işlemi bittikten sonra bu devre çözümünü anlatan bir çıktı dokümanı üretilir. Bu dokümanda devre çözümünde yer alan ara adımlarla birlikte devre elemanlarının hesaplanan akım ve gerilim değerleri yer alır.

Dokümanda gerek denklemlerin değişkenleriyle birlikte gösterimi gerekse değişkenlerin yerine değerlerinin konulması sonucu oluşan denklemler de gösterilir. Devre analizi sonucu oluşan devre elemanlarının akım ve gerilim değerleri de en son adımda gösterilir.

Doküman üretiminde dikkate alınan kanunlar Ohm kanunu, Kirchhoff'un akım kanunu ve Kirchhoff'un gerilim kanunudur. Ayrıca bu kanunlara graf analiz yöntemiyle devre analizi yapılırken oluşturulan matrislerin değerleri de dikkate alınır.

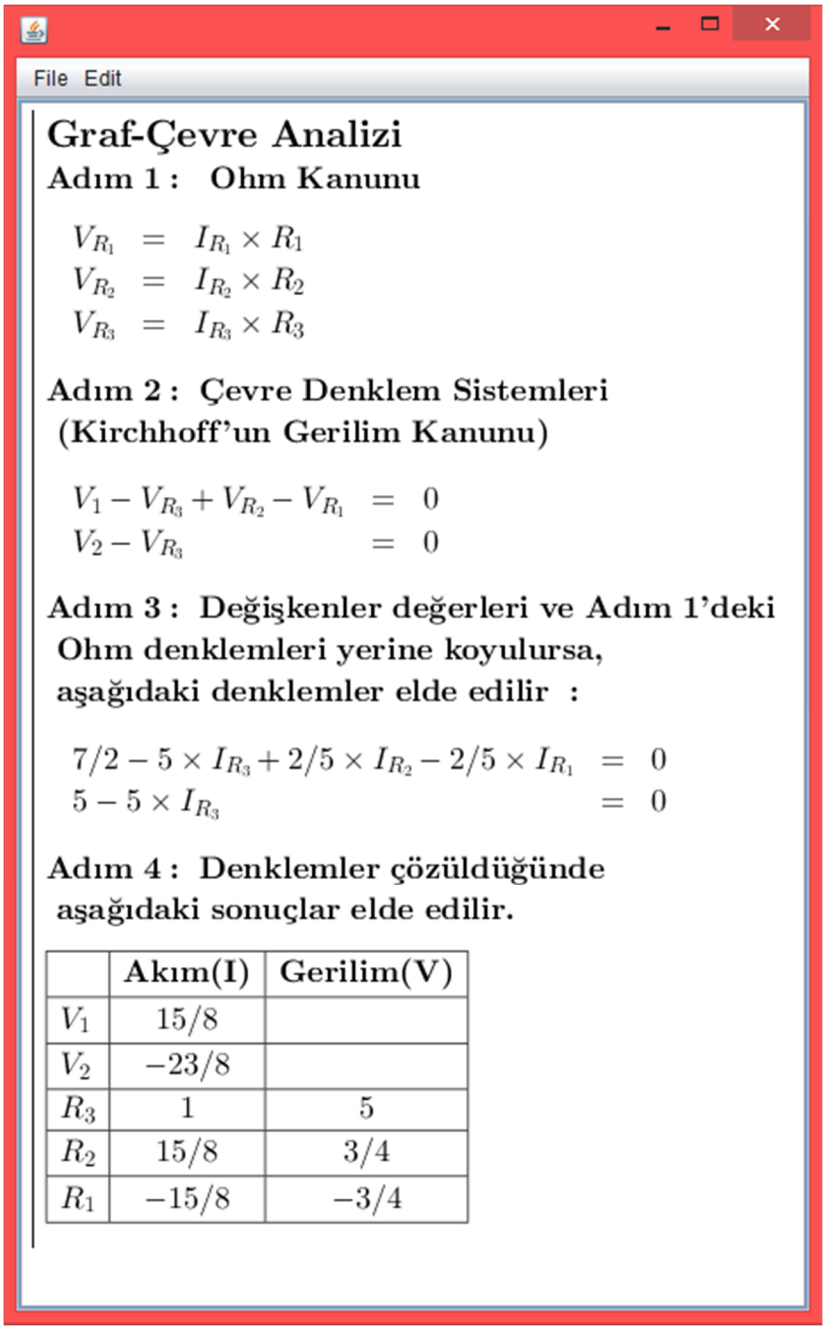
Graf-çevre analizi çıktı dokümanı incelenirse Şekil 2.2'deki devre ve onun çözüm adımları için işlem adımları şu şekilde çıktı dokümanına yansıtılır. Adım 1'de devreye bağlı olan dirençlerin Ohm kanununa göre denklemleri yazılır. Adım 2'de ise B_f temel çevrim matrisine göre oluşan denklemler yani Şekil 2.2'deki devre için bu denklemler (2.2) ve (2.3)'tür. Adım 3'te ise (2.2) ve (2.3)'teki değişkenler hem elemanları hem de Adım 1'deki

eşitliklerle yer değiştirilerek çevre denklemleri değerleriyle birlikte yazılır. Adım 4'te ise (2.14) ve (2.15)'teki hesaplanan sonuçlar olan akım ve gerilimler kullanıcılara gösterilir. Burada kullanıcılara gösterilen eşitlikler devre çözümünü anlamada yeterli olduğundan diğer ara işlemler gösterilmez.

Tablo 2.5. Şekil 2.2'deki devre çıktısının LaTeX dilindeki kaynak metni

```
\large \textbf{Graf-Çevre Analizi}\\\ textbf{Adım 1:  Ohm Kanunu}\\\\\
begin{multline} \begin{array}{lcl}
V_{R_{1}} & = & I_{R_{1}} \times R_{1}\\V_{R_{2}} & = & I_{R_{2}} \times
R_{2}
\\V_{R_{3}} & = & I_{R_{3}} \times R_{3}
\\\end{array} \end{multline}
\\\\\textbf{Adım 2: Çevre Denklem Sistemleri }
\\\textbf{ (Kirchhoff'un Gerilim Kanunu) }
\\\\\begin{multline} \begin{array}{lcl}
V_{1} - V_{R_{3}} + V_{R_{2}} - V_{R_{1}} & = & 0
\\V_{2} - V_{R_{3}} & = & 0
\\\end{array} \end{multline}
\\\\\textbf{Adım 3: Değişkenler değerleri ve Adım 1'deki}
\\\textbf{ Ohm denklemleri yerine koyulursa,}
\\\textbf{ aşağıdaki denklemler elde edilir : }
\\\\\begin{multline} \begin{array}{lcl}
7/2 - 5\times I_{R_{3}} + 2/5\times I_{R_{2}} - 2/5\times I_{R_{1}} & = & 0
\\5 - 5\times I_{R_{3}} & = & 0
\\\end{array} \end{multline}
\\\\\textbf{Adım 4: Denklemler çözüldüğünde }
\\\textbf{ aşağıdaki sonuçlar elde edilir. }
\\\\ \begin{tabular}{|l|c|c|}
 \hline & \textbf{Akım(I)} & \textbf{Gerilim(V)} \\ \hline
V_{1} & 15/8\\ \hline V_{2} & -23/8\\ \hline
R_{3} & 1 & 5\\ \hline R_{2} & 15/8 & 3/4\\
\hline R_{1} & -15/8 & -3/4\\ \hline
\end{tabular}\\
```

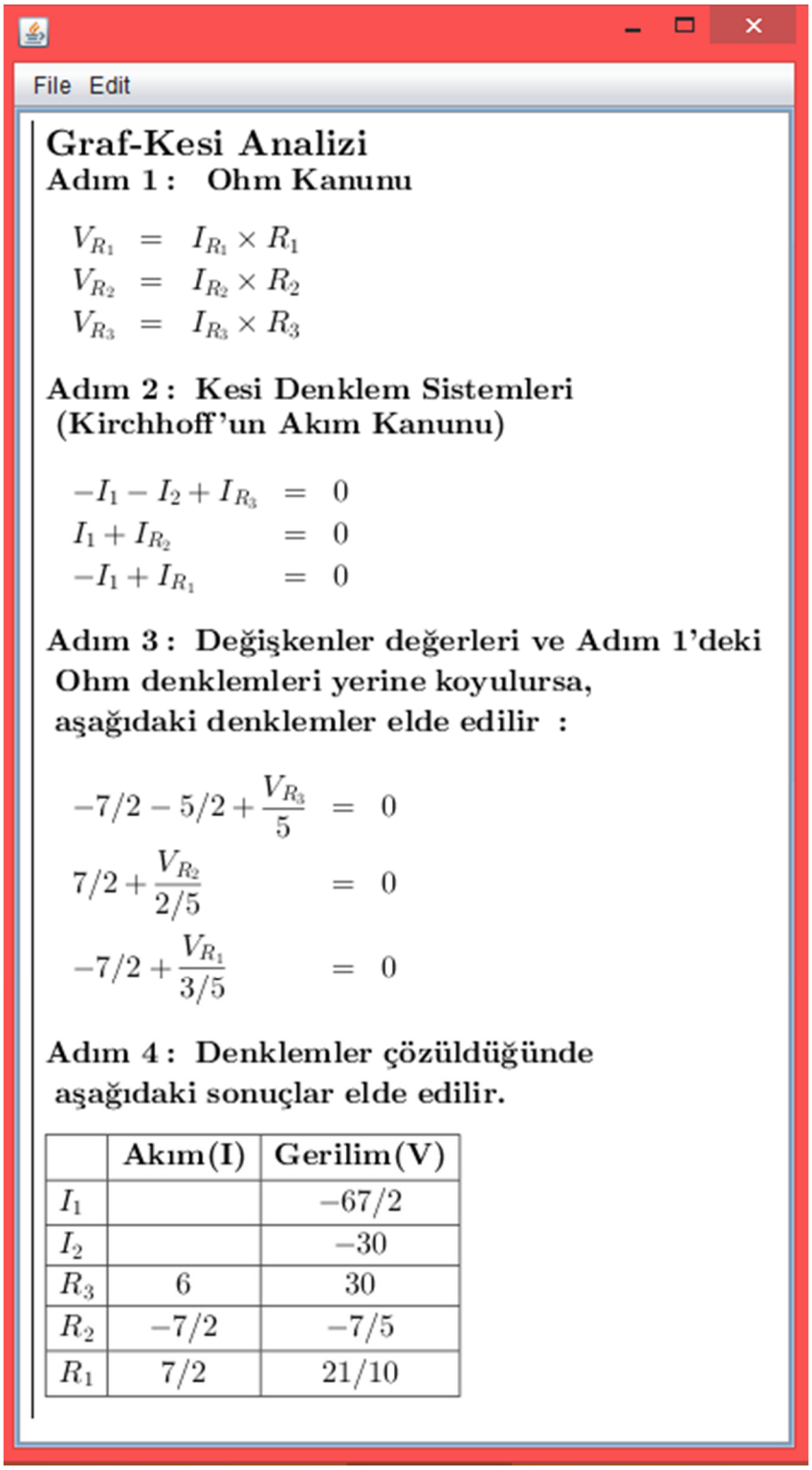
Eşitliklerin adım adım verilmesi ve anlaşılabilirlik açısından düzgün bir şekilde kullanıcılara gösterilmesi önemlidir. Özellikle matematiksel ifadeleri biçimlendirmede çok faydalı olan LaTeX dili kullanılarak eşitlikler okunabilir ve anlaşılabilir bir hale getirilir. Şekil 2.2'deki devrenin çıktı dokümanının LaTeX diliyle kodlanmış hali Tablo 2.5'te ve bu kaynak dosyasının yorumlanmış hali ise Şekil 2.8'de gösterilmektedir.



Şekil 2.8. Şekil 2.2'deki devrenin çıktı dokümanı

Tablo 2.6. Şekil 2.6'daki devre çıktısının LaTeX dilindeki kaynak metni

```
\large \textbf{Graf-Kesi Analizi}
\\\textbf{Adım 1:  Ohm Kanunu}
\\\\\begin{multline}\begin{array}{lcl}
V_{R_{1}} & = & I_{R_{1}} \times R_{1}
\\V_{R_{2}} & = & I_{R_{2}} \times R_{2}
\\V_{R_{3}} & = & I_{R_{3}} \times R_{3}
\\\end{array}\end{multline}
\\\\\textbf{Adım 2: Kesi Denklem Sistemleri }
\\\textbf{ (Kirchhoff'un Akım Kanunu) }
\\\\\begin{multline}\begin{array}{lcl}
- I_{1} - I_{2} + I_{R_{3}} & = & 0
\\I_{1} + I_{R_{2}} & = & 0 \\ - I_{1} + I_{R_{1}} & = & 0
\\\end{array}\end{multline}
\\\\\textbf{Adım 3: Değişkenler değerleri ve Adım 1'deki}
\\\textbf{ Ohm denklemleri yerine koyulursa,}
\\\textbf{ aşağıdaki denklemler elde edilir : }
\\\\\begin{multline}\begin{array}{lcl}
- 7/2 - 5/2 + \frac{ V_{R_{3}}}{5} & = & 0
\\7/2 + \frac{ V_{R_{2}}}{2/5} & = & 0
\\ - 7/2 + \frac{ V_{R_{1}}}{3/5} & = & 0
\\\end{array}\end{multline}
\\\\\textbf{Adım 4: Denklemler çözüldüğünde }
\\\textbf{ aşağıdaki sonuçlar elde edilir. }
\\\\ \begin{tabular}{|l|c|c|}
\hline & \textbf{Akım(I)} & \textbf{Gerilim(V)} \\ \hline
I_{1} & & -67/2\\ \hline I_{2} & & -30\\ \hline R_{3} & 6 & 30\\
\hline R_{2} & -7/2 & -7/5\\ \hline R_{1} & 7/2 & 21/10\\ \hline
\end{tabular}\\
```



Şekil 2.9. Şekil 2.6'daki devrenin çıktı dokümanı

Graf-kesi analizi çıktı dokümanı incelenirse Şekil 2.6'daki devre ve onun çözüm adımları şu şekilde çıktı dokümanına yansıtılır. Adım 1'de graf-çevre analizinde olduğu gibi

devreye bağlı olan dirençlerin Ohm kanununa göre denklemleri yazılır. Adım 2'de, Q_f temel çevrim matrisine göre oluşan denklemler gösterilir. Şekil 2.6'daki devre için bu denklemler (2.24), (2.25) ve (2.26)'dır. Adım 3'te ise kesi denklemleri (2.24), (2.25) ve (2.26)'nın değişkenleri, hem elemanları hem de Adım 1'deki eşitliklerle yer değiştirilerek değerleriyle birlikte gösterilir. Adım 4'te ise (2.33) ve (2.34)'teki hesaplanan sonuçlar olan gerilim ve akım değerleri kullanıcılara gösterilir. Burada kullanıcılara gösterilen eşitlikler devre çözümünü anlamada yeterli olduğundan diğer ara işlemler gösterilmez. Şekil 2.6'daki devrenin çıktı dokümanının LaTeX diliyle kodlanmış hali Tablo 2.6'da ve bu kaynak dosyasının yorumlanmış hali ise Şekil 2.9'da gösterilmektedir.

Doküman oluşturma da eğer graf-çevre analizi yöntemi kullanılmışsa, çıktı dokümanında KCL denklemleri yer almaz. Eğer graf-kesi kümesi analizi yapılmışsa o zaman çıktı dokümanın da KVL denklemleri yer almaz. Burada LaTeX'in yorumlanması için JLaTeXMath [25] Java kütüphanesi kullanılmıştır. Ayrıca graf analizi çıktısını oluşturan Java dosyası *GraphAnalysisLatex.java*'dır ve kaynak kodu Ek-3'te bulunmaktadır.

2.5. Yazılımdaki Diğer Faydalı Özellikler

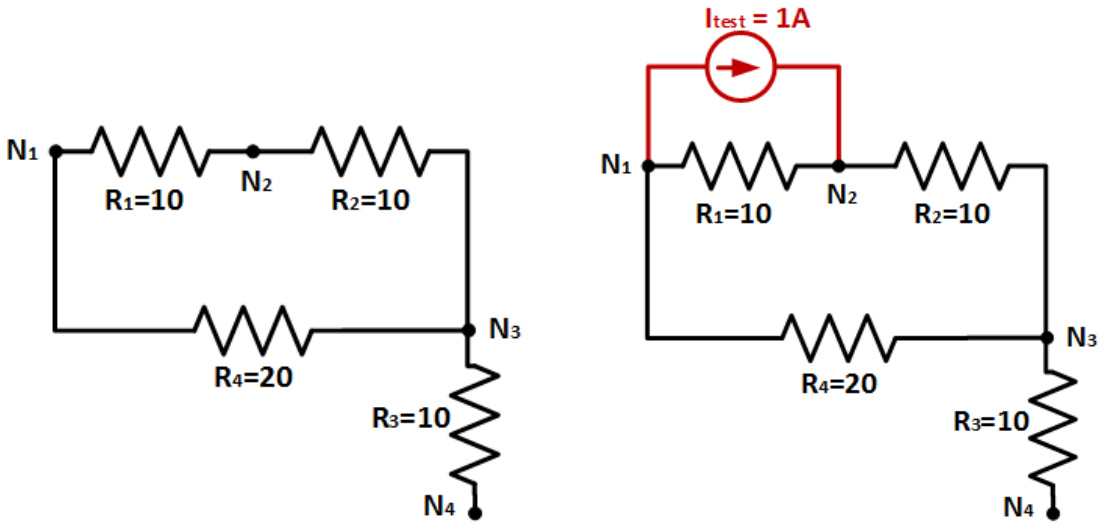
2.5.1. Eşdeğer Direnç Hesaplama

Eğer verilen devrede eşdeğer direnç hesabı yapılacaksa, öncelikle devrenin herhangi bir akım veya gerilim kaynağı içerip içermediği kontrolü yapılır. Eğer herhangi birini içeriyorsa hata mesajı üretilir. Daha sonra kullanıcıdan hangi iki düğüm arasındaki eşdeğer direncin hesaplanacağı bilgisi alınır. Devamında bu iki düğüm arasında 1 A'lık akım kaynağı bağlanarak devre analizi yapılır. Sonuç olarak bağlanan akım kaynağının iki ucu arasındaki hesaplanan gerilim değeri bize eşdeğer direnci verir. Tablo 2.7'de örnek bir devre girdisi ve bu girdinin verilen *"N1"* ve *"N2"* düğümleri arasındaki eşdeğer direncinin hesaplama çıktısı Şekil 2.11'de verilmektedir. Eşdeğer direnç hesaplayan Java kodu *EquivalentResistance.java* dosyasındadır ve Ek-4'te yer almaktadır. Ayrıca eşdeğer direnç hesaplama çıktısını üreten kaynak kod *EquivalentResistanceLatex.java* dosyasında yer almakta ve Ek-5'te verilmektedir.

Tablo 2.7. Şekil 2.6'daki devrenin DGD ile kodlanmış hali

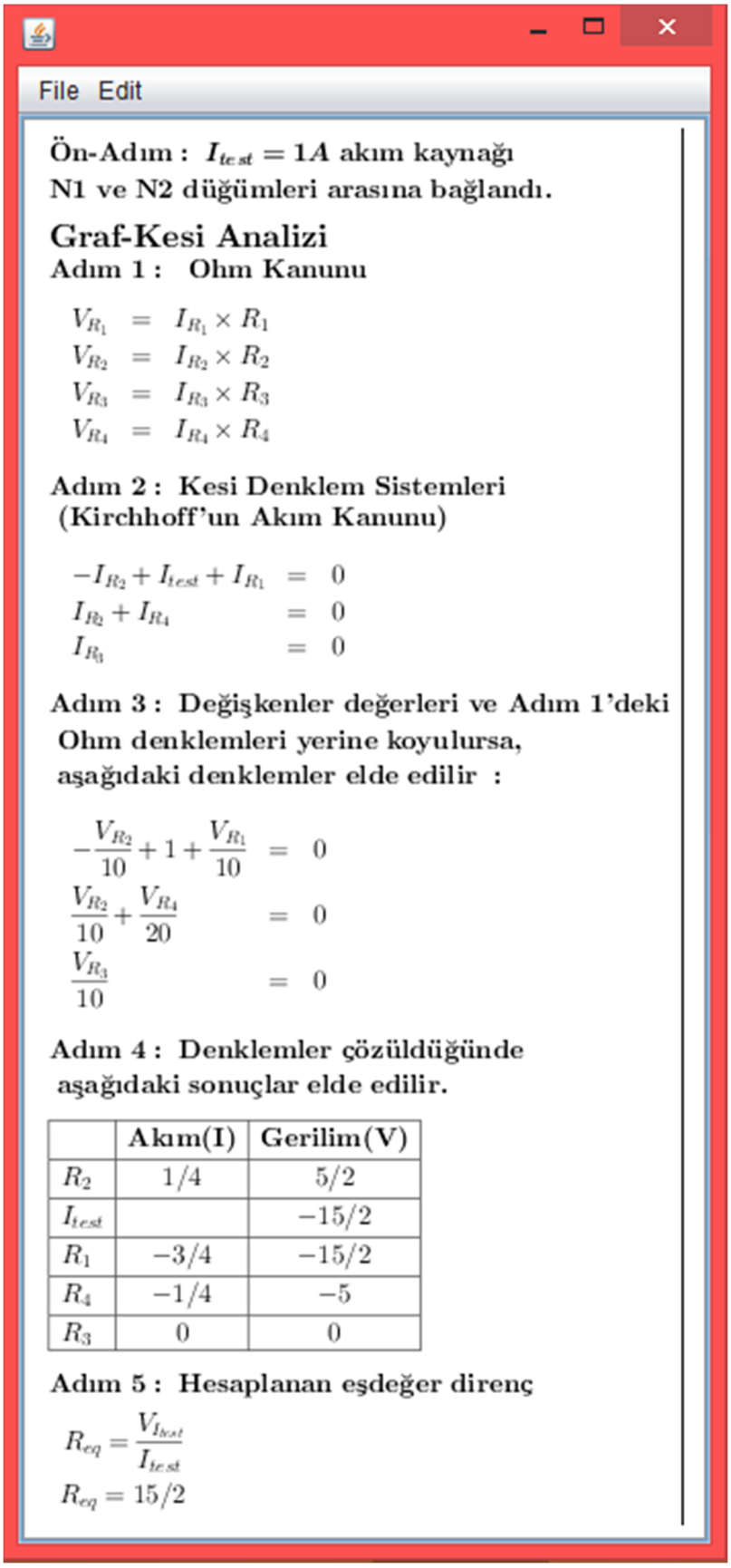
```
circuit {
        elements{
                resistor R1,R2,R3,R4;
                node N1,N2,N3,N4;
        }
        connections{
                R1(N1,N2);
                R3(N3,N4);
                R2(N2,N3);
                R4(N1,N3);
        }
        parameters{
                R1=10;
                R2=10;
                R3=10;
                R4=20;
        }
}
```

Tablo 2.7'de tanımlanan devrenin şematik gösterimi ve 1 A'lık akım kaynağı bağlanması sonucu oluşan devrenin şematik gösterimi Şekil 2.10'da gösterilmektedir.



(a) (b)

Şekil 2.10. Eşdeğer direnç hesabı için örnek bir devre (a) Eşdeğer direnç hesaplanacak devre şematiği (b) Test akım kaynağı bağlanmasıyla oluşan devre şematiği



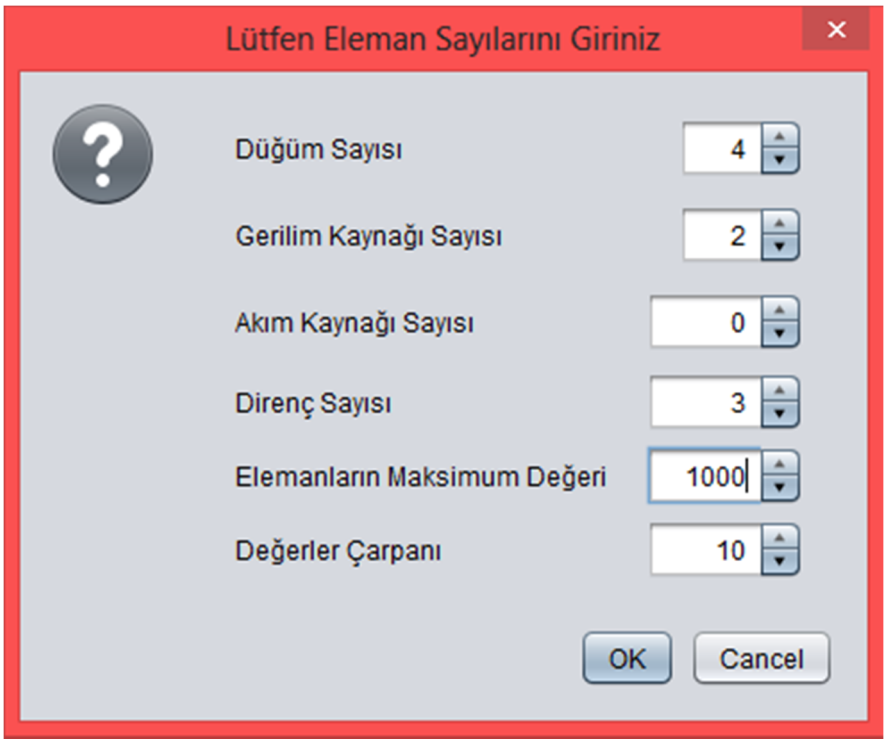
Şekil 2.11. Bir eşdeğer direnç hesaplama örnek çıktısı

2.5.2. Otomatik Devre Üretimi

Otomatik devre üretimi, devrede olabilecek eleman sayıları girdi olarak alınarak bu elemanların bağlantılarının rastgele yapılması ve ayrıca elemanlarının değerlerinin de rastgele verilmesiyle sonuçta bir elektrik devresi oluşturma işlemidir. Elektrik devreleri graf veri yapısıyla kolaylıkla temsil edilebildiğinden otomatik devre üretimi, rastgele bağlantılı graf üretimi problemi olarak görülebilir. Bu şekilde otomatik devre üretimi daha basit bir problem haline gelir. Çünkü rastgele bağlantılı graf üretimi için geliştirilmiş birçok algoritma mevcuttur.

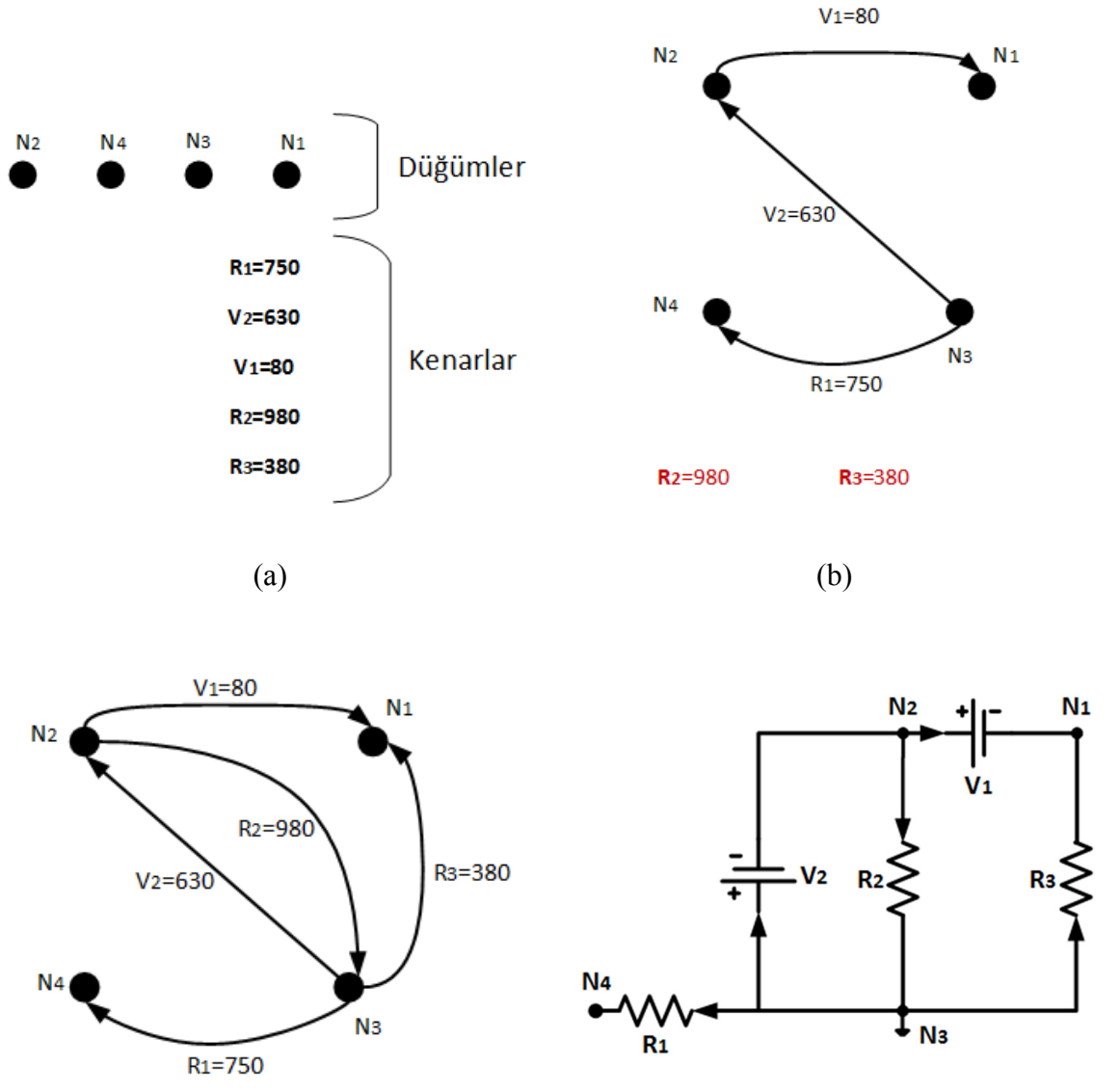
Otomatik devre üretilirken ilk önce kullanıcıdan akım kaynağı, gerilim kaynağı, direnç ve düğüm sayısı bilgisi alınır ve bir rastgele graf üretim algoritmasıyla otomatik devre üretilir. Devredeki elemanların da değerleri rastgele üretilir. Daha sonra üretilen devre, devre girdi diline dönüştürülerek analizi yapılabilecek bir devre haline getirilmiş olur.

Otomatik devre üretiminde, başlangıçta kullanıcıdan Şekil 2.12'de de gösterilen arayüzden devredeki eleman sayısı alınır.



Şekil 2.12. Otomatik devre üretimi için eleman sayısı girdi penceresi

Otomatik devre üretiminde, rastgele bağlantılı graf üretiminin gereği olarak bazı şartlar sağlanmalıdır. Devrenin düğüm sayısı *n* ve kenar sayısı *m* olarak alınırsa, *m* sayısı *(n*(n-1)/2)*'den küçük veya eşit olmalıdır. Aynı zamanda *m*, *(n-1)*'den de büyük olmalıdır. Kenar sayısı hesaplanırken gerilim kaynakları sayısı, akım kaynakları sayısı ve dirençlerin sayısı ele alınır.



Şekil 2.13. Şekil 2.12'de verilen parametrelere göre devre üretimi (a) Düğümler ve kenarların rastgele değerli üretimi (b) Açgözlü methotla rastgele açılım ağacı oluşturulması (c) Kalan diğer kenarların grafa rastgele eklenmesi (d) Otomatik üretilen devrenin şematik gösterimi

Aslında otomatik devre üretilirken yukarıda bahsedilen kurallar sağlandığında otomatik devre oluşturma işlemi için gerekli yeterlilikler sağlanmış olmaktadır. Fakat geliştirilen devre analiz programı sadece bir çeşit kaynak bağlı olduğunda çözüm işlemini yapabildiğinden, devre eleman sayıları belirlenirken sadece gerilim kaynağı sayısı veya sadece akım kaynağı sayısı belirlenerek devrenin otomatik üretiminin yapılması daha uygun olur.

Tablo 2.8. Otomatik üretilmiş bir devrenin DGD ile kodlanmış hali

```
circuit {
        elements {
                node N2;
                node N4;
                node N3;
                node N1;
                resistor R1;
                voltagesource V2;
                voltagesource V1;
                resistor R2;
                resistor R3;
        }
        connections{
                R1(N3,N4);
                V2(N3,N2);
                V1(N2,N1);
                R2(N2,N3);
                R3(N3,N1);
                gnd(N3);
        }
        parameters{
                R1=750;
                V2=630;
                V1=80;
                R2=980;
                R3=380;

        }
}
```

Rastgele devre üretimi aşamaları aşağıdaki gibi özetlenebilir.

- Kenar ve düğüm sayılarının uygun olup olmadığı kontrol edilir.

- Verilen devre elemanları sayılarına göre elemanlar oluşturulur ve bu elemanların değerleri yine verilen aralıklara göre atanır.
- Rastgele bağlantılı graf üretimine göre ilk olarak üretilen düğümler ve devre elemanları grafa eklenir. Daha sonra açgözlü yöntemle rastgele açılım ağacı oluşturulur. Dolaysıyla minimum bağlantılı graf ve devre oluşturulmuş olur.
- Açılım ağacı oluşturulduktan sonra eğer girilen eleman sayısından daha az kenar bağlanmışsa diğer elemanlar rastgele olacak seçilen düğümlere bağlanır.

Yukarıda belirtilen aşamalar örnek bir devre oluşturmak üzere uygulanırsa Şekil 2.12'de verilen girdi parametrelerine göre mümkün bir devre üretimi Şekil 2.13'teki gibi gösterilebilir.

Devre elemanlarının değerlerinin atanması tamamıyla rastgele yapılmaktadır. Fakat okunabilirlik yönünden daha uygun olması için devre elemanlarının değerlerinin 0 ile maksimum bir değer arasında üretimine izin verilmektedir. Bunun yanında bu rastgele sayıların belirli bir sayının katı olması isteniyorsa kullanıcının bu değeri de girmesi istenmektedir. Programda girdiler, maksimum değer 100 ve 10 sayısının katı olacak şekilde varsayılan olarak ayarlanmıştır.

Şekil 2.12'de verilen eleman sayılarına göre üretilen devrenin DGD'deki kaynak kodu Tablo 2.8'de verilmektedir.

Otomatik devre üretimi yapan Java kodu *CircuitUtils.java* dosyasında yer almakta ve Ek-6'da verilmektedir.

3. SONUÇLAR

Yapılan bu çalışmada, basit elektrik devreleri için bir e-öğrenme yazılımı geliştirilmiştir. Bu yazılımda analizi veya çözümü yapılacak devreyi ifade etmek için bir girdi dili tanımlanmakta, verilen girdi devresinin analizi yapılarak analiz ara adımlarını da içeren bir çıktı dokümanı üretilmektedir.

Çok yaygın kullanılan analiz ve simülasyon programı olan SPICE ve SPICE-tabanlı yazılımlarla karşılaştırıldığında, bu yazılımın eksileri ve artıları aşağıdaki gibi verilebilir. Geliştirilen yazılımın SPICE'ye göre üstünlükleri;

- SPICE, C ve Fortran dillerinde geliştirilmiştir. Bu yazılım ise çok yaygın kullanılan, nesne-tabanlı ve platform-bağımsız olan Java'da geliştirilmiştir. Dolaysıyla basit düzenlemelerde bu yazılım web-tabanlı bir uygulama haline getirilebilir.
- SPICE, analiz sonucu sadece elemanların akım ve gerilim değerlerini verir. Bu yazılım ise elemanların akım ve gerilim değerleri dahil analizde gerçekleştirilen ara adımları ve devrede kullanılan devre kanunlarının denklemlerini de çıktı olarak gösterir.
- SPICE, düğüm analizi veya modifiye düğüm analizi yöntemlerini temel alarak analiz işlemini gerçekleştirir. Bu yazılımda ise graf analiz yöntemi [15] kullanılmaktadır ve bu yöntemin gerçeklenmesi yani kodlanması ilk defa yapılmıştır.
- SPICE'de otomatik devre üretimi özelliği yer almamaktadır.
- SPICE'de matematiksel işlemler kayan noktalı sayı aritmetiğine göre yapıldığından kesme hataları oluşabilir. Bu yazılımda kesirli sayı aritmetiğine dayalı bir kütüphane kullanılarak bu hataların oluşması engellenmektedir.
- Devre elemanlarının değerlerinin kesirli formda alınabilmesi ve hesaplanan akım ve gerilim değerlerinin kesirli formda kullanıcılara gösterilmesidir.

Geliştirilen yazılımın zayıf yönleri ise;

- SPICE, elektrik devrelerindeki tüm elemanların analizini yapabilmektedir. Bu yazılımda ise sadece direnç, bağımsız akım kaynağı ve gerilim kaynağı kullanılabilmektedir.

- SPICE-tabanlı yazılımlarda kullanıcı ara yüzü yer almaktadır. Bu da kullanıcının daha hızlı devre girmesine ve dolaysıyla daha hızlı devre simülasyonu yapmasına yardımcı olmaktadır. Ayrıca devrenin şematik çizimi de interaktif arayüzlerden yapıldığından devre girdi hataları da engellenmektedir.

4. ÖNERİLER

1. Kapasite ve endüktansları da desteklenecek şekilde yazılıma eklemeler yapılabilir. Graf analiz yöntemi bu elemanları da desteklemektedir.
2. Bu program sadece bir çeşit kaynak üzerinde işlem yapmaktadır. Program genişletilerek her iki kaynak olduğunda da analiz yapılması sağlanabilir.
3. Elektrik devre eğitimlerinde genellikle öğretilen Thevenin ve Norton analiz yöntemleri eklenebilir.
4. Çıktı dokümanlarında devre şematik çizimlerinin yer alması ve bu çizimlerde akım yönlerinin ve çevrelerin gösterilmesi özelliği eklenebilir.

5. KAYNAKLAR

1. Karan, O., Design and Implementation of Interpreters, Yüksek Lisans Tezi, Haliç Üniversitesi, İstanbul, 2005.

2. Friedl, J., E., F., Mastering Regular Expressions, 2nd edition, O'Reilly Media, Cambridge, 2003.

3. Lerdo, H., G., A Translator Writing System for a Java Oriented Compilers Course, Yüksek Lisans Tezi, University of Florida, Florida, 2003.

4. Earley, J., An Efficient Context- Free Parsing Algorithm, Communication of the ACM, 13,2 (1970) 94-102.

5. URL: http://dinosaur.compilertools.net/lex/, Lex - A Lexical Analyzer Generator, 15 Aralık 2012.

6. URL: http://www.cs.princeton.edu/~appel/modern/java/JLex/, JLex: A Lexical Analyzer Generator for Java, 15 Aralık 2012.

7. URL: http://flex.sourceforge.net/, flex: The Fast Lexical Analyzer, 15 Eylül 2013.

8. URL: http://www.cs.nmsu.edu/~rth/cs/cs471/InterpretersJavaCC.html, Writing Interpreter with JavaCC, 15 Aralık 2012.

9. Kodaganallur, V., Incorporating Language Processing into Java Applications: A JavaCC Tutorial, Software, IEEE, 21 (2004) 70-77.

10. URL: https://javacc.java.net/, Java Compiler Compiler, 15 Kasım 2011.

11. Lafore, R., Data Structures & Algorithms in Java. 2nd ed. Indianapolis, Ind., Sams, 2003.

12. URL: http://en.wikipedia.org/wiki/Breadth-first_search, Breadth-first search, 15 Mart 2012.

13. Lau, H., T., A Java Library of Graph Algorithms and Optimization. Boca Raton, Chapman & Hall/CRC, 2007.

14. Irwin, J., D., Basic Engineering Circuit Analysis. 10th ed. New York, John Wiley, 2010.

15. Deo, N., Graph Theory With applications to Engineering And Computer Science, Englewood Cliffs, N.J., Prentice-Hall, 1974.

16. Chen, W., "Circuit Analysis: A Graph-Theoretic Foundation.", The Electrical Engineering Handbook. Boston, Elsevier Academic Press, 2004. 31-41.

17. Chen, W., Graph Theory and Its Engineering Applications, Singapore, World Scientific, 1997.

18. URL: http://en.wikipedia.org/wiki/Markup_language, Markup language, 15 Mayıs 2013.

19. URL: http://www.troff.org/, The Text Processor for Typesetters, 15 Mayıs 2013.

20. URL: http://www.latex-project.org/, LaTeX – A document preparation system, 15 Mayıs 2013.

21. URL: http://www.adobe.com/products/postscript/, Adobe PostScript 3, 15 Mayıs 2013.

22. URL: http://www.ni.com/white-paper/5416/en, Basic SPICE Simulation Model Parameters, 15 Mayıs 2013.

23. URL: http://jlinalg.sourceforge.net/, JLinAlg, 10 Şubat 2013.

24. URL: http://fifesoft.com/rsyntaxtextarea/, RSyntaxTextArea, 10 Şubat 2013.

25. URL: http://forge.scilab.org/index.php/p/jlatexmath/, JLaTeXMath - A Java API to render LaTeX, 5 Nisan 2013.

6. EKLER

Ek 1. CIL.jj

```
options {
  LOOKAHEAD = 1;
  STATIC = false;
  DEBUG_PARSER = false;
  UNICODE_INPUT = true;
}
PARSER_BEGIN(CILParser)
package com.zavrak.lang;
import com.zavrak.analysis.GraphAnalysis;
import com.zavrak.lang.ast.*;
import com.zavrak.lang.ast.exp.*;
public class CILParser {
   public static void main(String args[]) { }
}
PARSER_END(CILParser)
SKIP : { " " | "\t" | "\r" | "\n" }
TOKEN : /* NUMBER TYPES */
{ < NUMBER : ((["0"-"9"])+ ("." (["0"-"9"])+)?)> }
TOKEN : /* CIRCUIT ELEMENT ID'S */
{      < RESISTOR_ID : "R" ((["a"-"z"])*  (["A"-"Z"])*  (["0"-"9"])*)+ >
   | < CAPASITOR_ID : "C" ((["a"-"z"])*  (["A"-"Z"])*  (["0"-"9"])*)+ >
   | < INDUCTOR_ID : "L" ((["a"-"z"])*  (["A"-"Z"])*  (["0"-"9"])*)+ >
   | < VOLTAGE_SOURCE_ID : "V" ((["a"-"z"])*  (["A"-"Z"])*  (["0"-"9"])*)+ >
   | < CURRENT_SOURCE_ID : "I" ((["a"-"z"])*  (["A"-"Z"])*  (["0"-"9"])*)+ >
   | < NODE_ID: "N" ((["a"-"z"])*  (["A"-"Z"])*  (["0"-"9"])*)+ >
   | < GND : ("G" | "g") ("N" | "n") ("D" | "d") >
}
```

```
void parse() throws ElementAlreadyDeclaredException,
ElementNotDeclaredException : { }
{ "circuit" "{" elementsBlock() connectionsBlock() parametersBlock() "}" <EOF>
}
void elementsBlock() throws ElementAlreadyDeclaredException : { }
{    "elements" "{"    ( "resistor" resistor() ( "," resistor() )* ";"
  | "voltagesource"  voltageSource() ( "," voltageSource() )* ";"
  | "currentsource"  currentSource() ( "," currentSource() )* ";"
  | "node" node() ( "," node() )* ";"    )*   "}"
}
void resistor() throws ElementAlreadyDeclaredException: { Token id; }
{   id=<RESISTOR_ID> { circuit.addElement(id.image,new Resistor(id.image)); }
}
void voltageSource() throws ElementAlreadyDeclaredException: { Token id; }
{    id=<VOLTAGE_SOURCE_ID> { circuit.addElement(id.image,new
VoltageSource(id.image)); }
}
void currentSource() throws ElementAlreadyDeclaredException: { Token id; }
{  id=<CURRENT_SOURCE_ID> { circuit.addElement(id.image,new
CurrentSource(id.image)); }
}
void node() throws ElementAlreadyDeclaredException: { Token id; }
{   id=<NODE_ID> { circuit.addElement(id.image,new Node(id.image)); }
}
void connectionsBlock()  throws ElementNotDeclaredException  :
{Token id, idn1, idn2; }
{    "connections" "{"  ( (id=<RESISTOR_ID> | id=<VOLTAGE_SOURCE_ID> |
id=<CURRENT_SOURCE_ID> ) "(" idn1=<NODE_ID> "," idn2=<NODE_ID>
")" ";"   { circuit.setNodes(id.image, idn1.image, idn2.image); }  )*
  (<GND> "(" idn1=<NODE_ID> ")" ";" { circuit.setGround(idn1.image); } )? "}"
}
void parametersBlock() throws ElementNotDeclaredException :
{Token id, id2; RationalNum val; }
```

```
{    "parameters" "{"  ( ( id=<RESISTOR_ID> | id=<VOLTAGE_SOURCE_ID>
 |  id=<CURRENT_SOURCE_ID> )  "="  val=rationalNumber() ";"  {
circuit.setValue(id.image, val); }  )*    "}"
}
RationalNum rationalNumber(): { Num num1, num2;}
{  num1 = number() ( "/" num2 = number() { return new RationalNum(num1,
num2); })?
     { return new RationalNum(num1); }
}
Num number(): { Token t; }
{   "-" t=<NUMBER> { return new Num((-1)*Double.parseDouble(t.image)); }
|  t=<NUMBER> { return new Num(Double.parseDouble(t.image)); }
}
```

Ek 2. GraphAnalysis.java

```
package com.zavrak.analysis;
import com.zavrak.analysis.exp.IncompatibleSourceException;
import com.zavrak.analysis.exp.NoSourceException;
import com.zavrak.analysis.exp.NotEnoughChordsException;
import com.zavrak.graph.Graph;
import com.zavrak.lang.ast.Current;
import com.zavrak.lang.ast.CurrentSource;
import com.zavrak.lang.ast.Edge;
import com.zavrak.lang.ast.Node;
import com.zavrak.lang.ast.Resistor;
import com.zavrak.lang.ast.Voltage;
import com.zavrak.lang.ast.VoltageSource;
import com.zavrak.utils.MatrixUtils;
import java.util.ArrayList;
import java.util.Arrays;
import org.jlinalg.LinAlgFactory;
import org.jlinalg.LinSysSolver;
import org.jlinalg.Matrix;
import org.jlinalg.Vector;
import org.jlinalg.rational.Rational;
/** org.jlinalg.Matrix indices start from 1. On the other hand the incidence
 * matrix indices start from 0. We have to be careful when converting from
 * incidence matrix to org.jlinalg.Matrix object.
 * @author Sultan ZAVRAK  */
public class GraphAnalysis {
   public static final int LOOP_ANALYSIS = 345351654;
   public static final int CUT_ANALYSIS = 234576545;
   protected Graph graph;
   protected Matrix<Rational> fCircuit; // Bf  -> fundamental circuit matrix
   protected Matrix<Rational> fCutset;  // Qf -> fundamental cutset matrix
   protected Matrix<Rational> E;  //  voltage source matrix
```

```
    protected Matrix<Rational> J; // current source matrix
    protected Matrix<Rational> I;
    protected Matrix<Rational> V;
    protected Matrix<Rational> impedanceMat; // branch-impedance matrix  Z
    protected Matrix<Rational> Zm;
    protected Matrix<Rational> Im;
    protected Matrix<Rational> Em;
    protected Matrix<Rational> Jc;
    protected Matrix<Rational> Yc;
    protected Matrix<Rational> admittanceMat; // cut-admittance matrix Y
    private int refNodeIndex = 0;
    int[] unorderedEdgeIndices;
    LinAlgFactory<Rational> linAlgFactory;
    protected int analysisType = LOOP_ANALYSIS;
    protected Edge[] st;
    protected Edge[] chords;
    public GraphAnalysis(Graph graph, int analysisType) throws
IncompatibleSourceException, NoSourceException {
        this.graph = graph;
        this.linAlgFactory = new LinAlgFactory<>(Rational.FACTORY);
        this.analysisType = analysisType;
        checkIfThereIsAnySource();
        checkAnalysisTypeCompatibility();
    }
    public GraphAnalysis(Graph graph, int analysisType, Node ground) throws
IncompatibleSourceException, NoSourceException {
        this(graph, analysisType);
        if (ground != null) {
            int ind = this.graph.getNodeIndex(ground);
            if (ind > -1) {
                this.refNodeIndex = ind;
            }
        }
```

```
    }

    private boolean checkIfThereIsAnySource() throws NoSourceException {
        Edge[] edges = this.graph.getEdges();
        int count = 0;
        for (Edge e : edges) {
            if (e instanceof CurrentSource || e instanceof VoltageSource) {
                ++count;
            }
        }
        if (count < 1) {
            throw new NoSourceException("The is no voltage and current source in the
circuit");
        }
        return true;
    }
    private void checkAnalysisTypeCompatibility() throws
IncompatibleSourceException {
        if (analysisType == CUT_ANALYSIS) {
            if (!allCurrentSources()) {
                throw new IncompatibleSourceException("Not all of sources are current
sources");
            }
        }
        if (analysisType == LOOP_ANALYSIS) {
            if (!allVoltageSources()) {
                throw new IncompatibleSourceException("Not all of sources are voltage
sources");
            }
        }
    }

    /** checks if there is at least one current source in the circuit
```

```
 * @return true if all sources int the circuit are voltage source.
 */
private boolean allVoltageSources() {
   Edge[] edges = this.graph.getEdges();
   boolean all = true;
   for (Edge e : edges) {
      if (e instanceof CurrentSource) {
         all = false;
         break;
      }
   }
   return all;
}
/** Checks if there is at least one voltage source in the circuit
 * @return true if all sources int the circuit are current source.
 */
private boolean allCurrentSources() {
   Edge[] edges = this.graph.getEdges();
   boolean all = true;
   for (Edge e : edges) {
      if (e instanceof VoltageSource) {
         all = false;
         break;
      }
   }
   return all;
}
/**Builds the matrices that are needed for graph analysis. The matrices that
 * will be built are fundamental basis matrix, fundamental circuit matrix
 * and fundamental cutset matrix.
 */
private void buildFundamentalMatrices() throws NotEnoughChordsException {
```

```
int[][] incMatArray = graph.getIncidenceMatrixArrayDirected();
graph.displayIncidenceMatrix();
Matrix<Rational> incMatrix = MatrixUtils.toRationalMatrix(incMatArray);
st = graph.breathFirstTraverse(refNodeIndex);
chords = graph.getChords(st);
if(analysisType == LOOP_ANALYSIS && chords.length < 1)
   throw new NotEnoughChordsException("At least one chord must be found
in the circuit.");
int[] rowIndices = getBasisIncidenceMatrixRowIndices(incMatArray,
refNodeIndex);
int[] chordsIndices = getChordsIndices(chords);
int[] stIndices = getSpanningTreeIndices(st);
// basis incidence matrix
Matrix<Rational> basisIncMat = MatrixUtils.copySubMatrix(rowIndices, 1,
incMatrix.getCols(), incMatrix);
// reference node
Matrix<Rational> temp1 = basisIncMat.transpose();
Matrix<Rational> A11 = MatrixUtils.copySubMatrix(chordsIndices, 1,
basisIncMat.getRows(), temp1).transpose();
Matrix A12 = MatrixUtils.copySubMatrix(stIndices, 1,
basisIncMat.getRows(), temp1).transpose();
setUnorderedEdgeIndices(chordsIndices, stIndices);
Matrix Bf11 = null;
if (analysisType == LOOP_ANALYSIS) {
   Bf11 = linAlgFactory.identity(chords.length);
}

Matrix Qf12 = null;
if (analysisType == CUT_ANALYSIS) {
   Qf12 = linAlgFactory.identity(st.length);
}
Matrix Bf12 = A11.transpose().multiply(Rational.FACTORY.get(-
1)).multiply(A12.transpose().inverse());
```

```
        Matrix Qf11  = Bf12.transpose().multiply(Rational.FACTORY.get(-1));
        if (analysisType == LOOP_ANALYSIS) {
            buildFundamentalCircuitMatrix(Bf11, Bf12);
        }
        if (analysisType == CUT_ANALYSIS) {
            buildFundamentalCutsetMatrix(Qf11, Qf12);
        }
    }
    private void buildImpedanceMatrix() {
        Edge[] edges = graph.getEdges();
        int m = edges.length;
        impedanceMat = linAlgFactory.zeros(m, m);
        for (int i = 0; i < edges.length; i++) {
            if (edges[unorderedEdgeIndices[i]] instanceof Resistor) {
                Rational val = ((Resistor)
edges[unorderedEdgeIndices[i]]).value.getValue();
                impedanceMat.set(i + 1, i + 1, val);
            }
        }
    }
    /**  Index of Unordered array of edges starts from zero But index of rational
     * matrices or vector starts from 1. So in this method, one is always
     * subtracted from given index parameter
     * @param index matrix or vector index
     * @return edge given index-1
     */
    protected Edge getUnorderedEdge(int index) {
        return graph.getEdges()[unorderedEdgeIndices[index - 1]];
    }
    private void buildVoltageSourceMatrix() {
        Edge[] edges = graph.getEdges();
        int m = edges.length;
```

```
        E = linAlgFactory.zeros(m, 1);
        for (int i = 0; i < edges.length; i++) {
            if (edges[unorderedEdgeIndices[i]] instanceof VoltageSource) {
                Rational val = ((VoltageSource)
edges[unorderedEdgeIndices[i]]).getValue().getValue();
                E.set(i + 1, 1, val);
            }
        }
    }
    public void solve() throws NotEnoughChordsException {
        if (analysisType == LOOP_ANALYSIS) {
            solveLoopSystemOfEquations();
        }
        if (analysisType == CUT_ANALYSIS) {
            solveCutSystemOfEquations();
        }
    }
    public void solveLoopSystemOfEquations() throws NotEnoughChordsException
{     buildFundamentalMatrices();
        buildVoltageSourceMatrix();
        buildImpedanceMatrix();
        Em = fCircuit.multiply(Rational.FACTORY.get(-1)).multiply(E);
        Zm = fCircuit.multiply(impedanceMat).multiply(fCircuit.transpose());
        Vector v1 = Em.getCol(1);
        Vector Imtemp = LinSysSolver.solve(Zm, v1);
        Im = Imtemp.toMatrix();
        I = fCircuit.transpose().multiply(Im);
        V = impedanceMat.multiply(I);
        System.out.println("\nV\n" + V);
        for (int i = 0; i < I.getRows(); i++) {
            Edge e = graph.getEdges()[unorderedEdgeIndices[i]];
            if (e instanceof Resistor) {
                ((Resistor) e).setVoltage(new Voltage(V.get(i + 1, 1).doubleValue()));
```

```
            ((Resistor) e).setCurrent(new Current(I.get(i + 1, 1).doubleValue()));
        }
        if (e instanceof CurrentSource) {
            ((CurrentSource) e).setVoltage(new Voltage(V.get(i + 1,
1).doubleValue()));
            ((CurrentSource) e).setCurrent(new Current(I.get(i + 1,
1).doubleValue()));
        }
        if (e instanceof VoltageSource) {
            ((VoltageSource) e).setVoltage(new Voltage(V.get(i + 1,
1).doubleValue()));
            ((VoltageSource) e).setCurrent(new Current(I.get(i + 1,
1).doubleValue()));
        }
      }
    }
    public void solveCutSystemOfEquations() throws NotEnoughChordsException {
      buildFundamentalMatrices();
      buildCurrentSourceMatrix();
      buildAdmittanceMatrix();
      Jc = fCutset.multiply(J).multiply(Rational.FACTORY.get(-1));
      Yc = fCutset.multiply(admittanceMat).multiply(fCutset.transpose());
      Vector Vctemp = LinSysSolver.solve(Yc, Jc.getCol(1));
      Matrix Vc = Vctemp.toMatrix();
      V = fCutset.transpose().multiply(Vc);
      I = admittanceMat.multiply(V);
      for (int i = 0; i < V.getRows(); i++) {
        Edge e = graph.getEdges()[unorderedEdgeIndices[i]];
        if (e instanceof Resistor) {
            ((Resistor) e).setVoltage(new Voltage(V.get(i + 1, 1).doubleValue()));
            ((Resistor) e).setCurrent(new Current(I.get(i + 1, 1).doubleValue()));
        }
        if (e instanceof CurrentSource) {
```

```
            ((CurrentSource) e).setVoltage(new Voltage(V.get(i + 1,
1).doubleValue()));
            ((CurrentSource) e).setCurrent(new Current(I.get(i + 1,
1).doubleValue()));
         }
         if (e instanceof VoltageSource) {
            ((VoltageSource) e).setVoltage(new Voltage(V.get(i + 1,
1).doubleValue()));
            ((VoltageSource) e).setCurrent(new Current(I.get(i + 1,
1).doubleValue()));
         }
      }
   }

   private void buildCurrentSourceMatrix() {
      Edge[] edges = graph.getEdges();
      int m = edges.length;
      J = linAlgFactory.zeros(m, 1);
      for (int i = 0; i < edges.length; i++) {
         if (edges[unorderedEdgeIndices[i]] instanceof CurrentSource) {
            Rational val = ((CurrentSource)
edges[unorderedEdgeIndices[i]]).getValue().getValue();
            J.set(i + 1, 1, val);
         }
      }
   }
   private void buildAdmittanceMatrix() {
      Edge[] edges = graph.getEdges();
      int m = edges.length;
      admittanceMat = linAlgFactory.zeros(m, m);
      for (int i = 0; i < edges.length; i++) {
         if (edges[unorderedEdgeIndices[i]] instanceof Resistor) {
```

```
            Rational val = ((Resistor)
edges[unorderedEdgeIndices[i]]).value.getValue();
            Rational one = Rational.FACTORY.one();
            admittanceMat.set(i + 1, i + 1, one.divide(val));
         }
      }
   }
   private int[] getBasisIncidenceMatrixRowIndices(int[][] incMatArray, int
referenceNodeIndex) {
      int[] rowIndices = new int[incMatArray.length - 1];
      for (int i = 0, j = 0; i < incMatArray.length - 1; i++, j++) {
         if (i != referenceNodeIndex) {
            rowIndices[i] = j;
         } else {
            rowIndices[i] = ++j;
         }
      }
      return rowIndices;
   }
   private int[] getChordsIndices(Edge[] chords) {
      int[] chordsIndices = new int[chords.length];
      for (int i = 0; i < chords.length; i++) {
         chordsIndices[i] = graph.getEdgeIndex(chords[i]);
      }
      return chordsIndices;
   }
   private int[] getSpanningTreeIndices(Edge[] st) {
      int[] spanningTreeIndices = new int[st.length];
      for (int i = 0; i < st.length; i++) {
         spanningTreeIndices[i] = graph.getEdgeIndex(st[i]);
      }
      return spanningTreeIndices;
   }
```

```
   private void setUnorderedEdgeIndices(int[] chordsIndices, int[]
spanTreeIndices) {
      // kenar sutun sırası tutuluyor
      unorderedEdgeIndices = new int[chordsIndices.length +
spanTreeIndices.length];
      System.arraycopy(chordsIndices, 0, unorderedEdgeIndices, 0,
chordsIndices.length);
      System.arraycopy(spanTreeIndices, 0, unorderedEdgeIndices,
            chordsIndices.length, spanTreeIndices.length);
   }
   /**@return the referenceNodeIndex */
   protected int getReferenceNodeIndex() {
      return refNodeIndex;
   }
   /**  @param referenceNodeIndex the referenceNodeIndex to set  */
   public void setReferenceNodeIndex(int referenceNodeIndex) {
      this.refNodeIndex = referenceNodeIndex;
   }
   private void buildFundamentalCutsetMatrix(Matrix Qf11, Matrix Qf12) {
      // fundamental cutset matrix
      fCutset = linAlgFactory.zeros(Qf11.getRows(), Qf11.getCols() +
Qf12.getCols());
      MatrixUtils.setSubMatrix(fCutset, 1, Qf11.getRows(), 1, Qf11.getCols(),
Qf11);
      MatrixUtils.setSubMatrix(fCutset, 1, Qf12.getRows(), Qf11.getCols() + 1,
Qf11.getCols() + Qf12.getCols(), Qf12);
   }
   private void buildFundamentalCircuitMatrix(Matrix Bf11, Matrix Bf12) {
      // fundamental circuit matrix
      fCircuit = linAlgFactory.zeros(Bf11.getRows(), Bf11.getCols() +
Bf12.getCols());
```

```
        MatrixUtils.setSubMatrix(fCircuit, 1, Bf11.getRows(), 1, Bf11.getCols(),
Bf11);
        MatrixUtils.setSubMatrix(fCircuit, 1, Bf12.getRows(), Bf11.getCols() + 1,
Bf11.getCols() + Bf12.getCols(), Bf12);
    }

    protected int getUnorderedEdgeIndex(int index) {
        return unorderedEdgeIndices[index - 1];
    }
}
```

Ek 3. GraphAnalysisLatex.java

```
package com.zavrak.analysis;
import com.zavrak.graph.Graph;
import com.zavrak.lang.ast.CurrentSource;
import com.zavrak.lang.ast.Edge;
import com.zavrak.lang.ast.Resistor;
import com.zavrak.lang.ast.VoltageSource;
import org.jlinalg.Matrix;
import org.jlinalg.rational.Rational;
/**  @author Sultan ZAVRAK  */
public class GraphAnalysisLatex implements ILatexBuilder {
   GraphAnalysis ga;
   public GraphAnalysisLatex(GraphAnalysis ga) {
      this.ga = ga;
   }
   @Override
   public String buildLatex() {
      StringBuilder laBuffer = new StringBuilder();
      if (ga.analysisType == GraphAnalysis.LOOP_ANALYSIS) {
         laBuffer.append("\\large \\textbf{Graf-Çevre Analizi}\\\\");
      }
      if (ga.analysisType == GraphAnalysis.CUT_ANALYSIS) {
         laBuffer.append("\\large \\textbf{Graf-Kesi Analizi}\\\\");
      }
      laBuffer.append("\\textbf{Adım 1:  Ohm Kanunu}\\\\\\\\");
      laBuffer.append(ohmsEquations(ga.graph));
      if (ga.analysisType == GraphAnalysis.LOOP_ANALYSIS) {
         laBuffer.append("\\\\\\textbf{Adım 2: Çevre Denklem Sistemleri }\\\\");
         laBuffer.append("\\textbf{ (Kirchhoff'un Gerilim Kanunu) }\\\\\\\\");
         laBuffer.append(loopSystemOfEquations(ga));
      }
      if (ga.analysisType == GraphAnalysis.CUT_ANALYSIS) {
```

```
            laBuffer.append("\\\\\\textbf{Adım 2: Kesi Denklem Sistemleri }\\\\");
            laBuffer.append("\\textbf{ (Kirchhoff'un Akım Kanunu) }\\\\\\\\");
            laBuffer.append(cutSystemOfEquations(ga));
        }
        laBuffer.append("\\\\\\textbf{Adım 3: Değişkenler değerleri ve Adım
1'deki}\\\\");
        laBuffer.append("\\textbf{ Ohm denklemleri yerine koyulursa,}\\\\");
        laBuffer.append("\\textbf{ aşağıdaki denklemler elde edilir : }\\\\\\\\");
        if (ga.analysisType == GraphAnalysis.LOOP_ANALYSIS) {
            laBuffer.append(loopSystemOfEquationsWithValues(ga));
        }
        if (ga.analysisType == GraphAnalysis.CUT_ANALYSIS) {
            laBuffer.append(cutSystemOfEquationsWithValues(ga));
        }
        laBuffer.append("\\\\\\textbf{Adım 4: Denklemler çözüldüğünde }\\\\");
        laBuffer.append("\\textbf{ aşağıdaki sonuçlar elde edilir. }\\\\\\\\");
        laBuffer.append(results(ga));
        return laBuffer.toString();
    }
    /** You should first solve equations to get results.
     * @returnlatex analysis results of given circuit as latex.
     */
    public String results(GraphAnalysis ga) {
        Matrix<Rational> I = ga.I;
        Matrix<Rational> V = ga.V;
        StringBuilder laBuffer = new StringBuilder();
        laBuffer.append(" \\begin{tabular}{|l|c|c|} ");
        laBuffer.append(" \\hline & \\textbf{Akım(I)} & \\textbf{Gerilim(V)} \\\\ \\hline
");
        for (int i = 1; i <= I.getRows(); i++) {
            Edge e = ga.getUnorderedEdge(i);

            if (e instanceof Resistor) {
```

```
            laBuffer.append("R_{").append(e.elmName.substring(1)).append("} & ")
                    .append(I.get(i, 1)).append(" & ");
            }
            if (e instanceof VoltageSource) {
               laBuffer.append("V_{").append(e.elmName.substring(1)).append("} & ")
                    .append(I.get(i, 1)). append("\\\\ \\hline ");
            }
            if (e instanceof CurrentSource) {
               laBuffer.append("I_{").append(e.elmName.substring(1)).append("} & &
").append(V.get(i, 1)).append("\\\\ \\hline ");
            }
         }
         laBuffer.append("\\end{tabular}\\\\");
         return laBuffer.toString();
      }
      public String loopSystemOfEquations(GraphAnalysis ga) {
         Matrix<Rational> fCircuit = ga.fCircuit;
         StringBuilder laBuffer = new StringBuilder();
         laBuffer.append("\\begin{multline}");
         laBuffer.append("\\begin{array}{lcl}");
         for (int i = 1; i <= fCircuit.getRows(); i++) {
            StringBuilder temp = new StringBuilder();
            for (int j = 1; j <= fCircuit.getCols(); j++) {
               Edge e = ga.getUnorderedEdge(j);
               if (e instanceof Resistor) {
                  if (fCircuit.get(i, j).doubleValue() > 0) {
                     if (temp.toString().isEmpty()) {
temp.append("V_{R_{").append(e.elmName.substring(1)).append("}}");
                     } else {
                        temp.append(" +
V_{R_{").append(e.elmName.substring(1)).append("}}");
                     }
```

```
                } else if (fCircuit.get(i, j).doubleValue() < 0) {
                    temp.append(" -
V_{R_{").append(e.elmName.substring(1)).append("}}");
                }
            } else if (e instanceof VoltageSource) {
                if (fCircuit.get(i, j).doubleValue() > 0) {
                    if (temp.toString().isEmpty()) {
temp.append("V_{").append(e.elmName.substring(1)).append("}");
                    } else {   temp.append(" +
V_{").append(e.elmName.substring(1)).append("}");
                    }
                } else if (fCircuit.get(i, j).doubleValue() < 0) {
                    temp.append(" -
V_{").append(e.elmName.substring(1)).append("}");
                }
            } else if (e instanceof CurrentSource) {
                if (fCircuit.get(i, j).doubleValue() > 0) {
                    if (temp.toString().isEmpty()) {
temp.append("V_{I_{").append(e.elmName.substring(1)).append("}}");
                    } else {
                        temp.append(" +
V_{I_{").append(e.elmName.substring(1)).append("}}");
                    }
                } else if (fCircuit.get(i, j).doubleValue() < 0) {
                    temp.append(" -
V_{I_{").append(e.elmName.substring(1)).append("}}");
                }
            }
        }
        temp.append(" & = & 0 \\\\");
        laBuffer.append(temp);
    }
    laBuffer.append("\\end{array}");
```

```
        laBuffer.append("\\end{multline}\\\\");
        return laBuffer.toString();
    }
    public String loopSystemOfEquationsWithValues(GraphAnalysis ga) {
        Matrix<Rational> fCircuit = ga.fCircuit;
        StringBuilder laBuffer = new StringBuilder();
        laBuffer.append("\\begin{multline}");
        laBuffer.append("\\begin{array}{lcl}");
        for (int i = 1; i <= fCircuit.getRows(); i++) {
            StringBuffer temp = new StringBuffer("");
            for (int j = 1; j <= fCircuit.getCols(); j++) {
                Edge e = ga.getUnorderedEdge(j);
                if (e instanceof Resistor) {
                    if (fCircuit.get(i, j).doubleValue() > 0) {
                        if (temp.toString().isEmpty()) {
                            Rational val = ((Resistor) e).value.getValue();

                            temp.append(val).append("\\times
I_{R_{").append(e.elmName.substring(1)).append("}}");
                        } else {
                            temp.append(" + ");
                            Rational val = ((Resistor) e).value.getValue();
                            temp.append(val).append("\\times
I_{R_{").append(e.elmName.substring(1)).append("}}");
                        }
                    } else if (fCircuit.get(i, j).doubleValue() < 0) {
                        temp.append(" - ");
                        Rational val = ((Resistor) e).value.getValue();
                        temp.append(val).append("\\times I_{" +
"R_{").append(e.elmName.substring(1)).append("}}");
                    }
                } else if (e instanceof VoltageSource) {
```

```
            if (fCircuit.get(i, j).doubleValue() > 0) {
               if (temp.toString().isEmpty()) {
                  temp.append(((VoltageSource) e).getValue().getValue());
               } else {
                  temp.append(" + " + ((VoltageSource) e).getValue().getValue());
               }

            } else if (fCircuit.get(i, j).doubleValue() < 0) {
               temp.append(" - " + ((VoltageSource) e).getValue().getValue());
            }
         }
      }
      temp.append(" & = & 0 \\\\");
      laBuffer.append(temp);
   }
   laBuffer.append("\\end{array}");
   laBuffer.append("\\end{multline}\\\\");
   return laBuffer.toString();
}

public String cutSystemOfEquationsWithValues(GraphAnalysis ga) {
   Matrix<Rational> fCutset = ga.fCutset;
   StringBuilder laBuffer = new StringBuilder();
   laBuffer.append("\\begin{multline}");
   laBuffer.append("\\begin{array}{lcl}");
   for (int i = 1; i <= fCutset.getRows(); i++) {
      StringBuffer temp = new StringBuffer("");
      for (int j = 1; j <= fCutset.getCols(); j++) {
         Edge e = ga.getUnorderedEdge(j);
         if (e instanceof Resistor) {
            if (fCutset.get(i, j).doubleValue() > 0) {
               if (temp.toString().isEmpty()) {
```

```
temp.append("\\frac{V_{R_{").append(e.elmName.substring(1)).append("}}}{").ap
pend(((Resistor) e).value).append("}");
                    } else {
                        temp.append(" + ");//.append(((Resistor) e).value);
                        temp.append("\\frac{
V_{R_{").append(e.elmName.substring(1)).append("}}}{").append(((Resistor)
e).value).append("}");
                    }
                } else if (fCutset.get(i, j).doubleValue() < 0) {
                    temp.append(" - ");
                    temp.append("\\frac{
V_{R_{").append(e.elmName.substring(1)).append("}}}{").append(((Resistor)
e).value).append("}");
                }
            } else if (e instanceof CurrentSource) {
                if (fCutset.get(i, j).doubleValue() > 0) {
                    if (temp.toString().isEmpty()) {
                        temp.append(((CurrentSource) e).getValue().getValue());
                    } else {
                        temp.append(" + " + ((CurrentSource) e).getValue().getValue());
                    }
                } else if (fCutset.get(i, j).doubleValue() < 0) {
                    temp.append(" - " + ((CurrentSource) e).getValue().getValue());
                }
            }
        }
        temp.append(" & = & 0 \\\\");
        laBuffer.append(temp);
    }
    laBuffer.append("\\end{array}");
    laBuffer.append("\\end{multline}\\\\");
    return laBuffer.toString();
  }
```

```
public String rearrangedLoopSystemOfEquations(GraphAnalysis ga) {
    Matrix<Rational> Zm = ga.Zm;
    Matrix<Rational> Em = ga.Em;
    StringBuilder laBuffer = new StringBuilder();
    laBuffer.append("\\begin{multline}");
    laBuffer.append("\\begin{array}{lcl}");
    for (int i = 1; i <= Zm.getRows(); i++) {
        StringBuffer temp = new StringBuffer("");
        for (int j = 1; j <= Zm.getCols(); j++) {
            Edge e = ga.getUnorderedEdge(j);
            if (e instanceof Resistor) {
                if (!Zm.get(i, j).isZero()) {
                    if (temp.toString().isEmpty()) {
                        temp.append(Zm.get(i, j)).append("\\times
I_{R_{").append(e.elmName.substring(1)).append("}}");
                    } else {
                        temp.append(" + ").append(Zm.get(i, j)).append("\\times
I_{R_{").append(e.elmName.substring(1)).append("}}");
                    }
                }
            } else if (e instanceof VoltageSource) {
                if (!Zm.get(i, j).isZero()) {
                    if (temp.toString().isEmpty()) {
                        temp.append(Zm.get(i, j)).append("\\times
I_{V_{").append(e.elmName.substring(1)).append("}}");
                    } else {
                        temp.append(" + ").append(Zm.get(i, j)).append("\\times
I_{V_{").append(e.elmName.substring(1)).append("}}");
                    }
                }
            }
        }
```

```
            temp.append(" & = & ").append(Em.get(i, 1)).append("\\\\");
            laBuffer.append(temp);
        }
        laBuffer.append("\\end{array}");
        laBuffer.append("\\end{multline}\\\\");
        return laBuffer.toString();
    }
    public String rearrangedCutSystemOfEquations(GraphAnalysis ga) {
        Matrix<Rational> Yc = ga.Yc;
        Matrix<Rational> Jc = ga.Jc;
        StringBuilder laBuffer = new StringBuilder();
        laBuffer.append("\\begin{multline}");
        laBuffer.append("\\begin{array}{lcl}");
        for (int i = 1; i <= Yc.getRows(); i++) {
            StringBuffer temp = new StringBuffer("");
            for (int j = 1; j <= Yc.getCols(); j++) {
                Edge e = ga.getUnorderedEdge(j);
                if (e instanceof Resistor) {
                    if (!Yc.get(i, j).isZero()) {
                        if (temp.toString().isEmpty()) {
                            temp.append(Yc.get(i, j)).append("\\times
V_{R_{").append(e.elmName.substring(1)).append("}}");
                        } else {
                            temp.append(" + ").append(Yc.get(i, j)).append("\\times
V_{R_{").append(e.elmName.substring(1)).append("}}");
                        }
                    }
                } else if (e instanceof CurrentSource) {
                    if (!Yc.get(i, j).isZero()) {
                        if (temp.toString().isEmpty()) {
                            temp.append(Yc.get(i, j)).append("\\times
V_{I_{").append(e.elmName.substring(1)).append("}}");
                        } else {
```

```
                    temp.append(" + ").append(Yc.get(i, j)).append("\\times
V_{I_{").append(e.elmName.substring(1)).append("}}");
                }
            }
        }
    }
    temp.append(" & = & ").append(Jc.get(i, 1)).append("\\\\");
    laBuffer.append(temp);
  }
  laBuffer.append("\\end{array}");
  laBuffer.append("\\end{multline}\\\\");
  return laBuffer.toString();
 }
 public String ohmsEquations(Graph graph) {
  StringBuilder laBuffer = new StringBuilder();
  laBuffer.append("\\begin{multline}");
  laBuffer.append("\\begin{array}{lcl}");
  for (Edge e : graph.getEdges()) {
   if (e instanceof Resistor) {
    laBuffer.append("V_{R_{").append(e.elmName.substring(1)).append("}}
& = & ");
    laBuffer.append("I_{R_{").append(e.elmName.substring(1)).append("}}
\\times ");
    laBuffer.append("R_{").append(e.elmName.substring(1)).append("}\\\\");
   }
  }
  laBuffer.append("\\end{array}");
  laBuffer.append("\\end{multline}\\\\");
  return laBuffer.toString();
 }
 public String cutSystemOfEquations(GraphAnalysis ga) {
  Matrix<Rational> fCutset = ga.fCutset;
```

```
StringBuilder laBuffer = new StringBuilder();
laBuffer.append("\\begin{multline}");
laBuffer.append("\\begin{array}{lcl}");
for (int i = 1; i <= fCutset.getRows(); i++) {
   StringBuffer temp = new StringBuffer("");
   for (int j = 1; j <= fCutset.getCols(); j++) {
      Edge e = ga.getUnorderedEdge(j);
      if (e instanceof Resistor) {
         if (fCutset.get(i, j).doubleValue() > 0) {
            if (temp.toString().isEmpty()) {
temp.append("I_{R_{").append(e.elmName.substring(1)).append("}}");
            } else {
               temp.append(" +
I_{R_{").append(e.elmName.substring(1)).append("}}");
            }
         } else if (fCutset.get(i, j).doubleValue() < 0) {
            temp.append(" -
I_{R_{").append(e.elmName.substring(1)).append("}}");
         }
      }
      if (e instanceof VoltageSource) {
         if (fCutset.get(i, j).doubleValue() > 0) {
            if (temp.toString().isEmpty()) {
temp.append("I_{V_{").append(e.elmName.substring(1)).append("}}");
            } else {
               temp.append(" +
I_{V_{").append(e.elmName.substring(1)).append("}}");
            }
         } else if (fCutset.get(i, j).doubleValue() < 0) {
            temp.append(" -
I_{V_{").append(e.elmName.substring(1)).append("}}");
         }
      }
```

```
            if (e instanceof CurrentSource) {
                if (fCutset.get(i, j).doubleValue() > 0) {
                    if (temp.toString().isEmpty()) {
temp.append("I_{").append(e.elmName.substring(1)).append("}");
                    } else {
                        temp.append(" +
I_{").append(e.elmName.substring(1)).append("}");
                    }
                } else if (fCutset.get(i, j).doubleValue() < 0) {
                    temp.append(" - I_{").append(e.elmName.substring(1)).append("}");
                }
            }
        }
        temp.append(" & = & 0 \\\\");
        laBuffer.append(temp);
    }
    laBuffer.append("\\end{array}");
    laBuffer.append("\\end{multline}\\\\");
    return laBuffer.toString();
  }
}
```

Ek 4. EquivalentResistance.java

```
package com.zavrak.analysis;
import com.zavrak.analysis.exp.IncompatibleSourceException;
import com.zavrak.analysis.exp.NoSourceException;
import com.zavrak.analysis.exp.NotEnoughChordsException;
import com.zavrak.analysis.exp.NotOnlyResistorsException;
import com.zavrak.graph.Graph;
import com.zavrak.lang.ast.CurrentSource;
import com.zavrak.lang.ast.Edge;
import com.zavrak.lang.ast.Node;
import com.zavrak.lang.ast.RationalNum;
import com.zavrak.lang.ast.Resistor;
/**  @author Sultan ZAVRAK  */
public class EquivalentResistance {
   Graph graph;
   protected Node startNode;
   protected Node endNode;
   Edge edge;
   public GraphAnalysis ga;
   public EquivalentResistance(Graph graph, Node node1, Node node2) {
      this.graph = graph;
      this.startNode = node1;
      this.endNode = node2;
   }
   private void checkAreThereSourceElements() throws NotOnlyResistorsException
   {    Edge[] edges = this.graph.getEdges();
       for(Edge e : edges)
       {
          if(!(e instanceof Resistor)) {
             throw new NotOnlyResistorsException("The circuit must contain only
resistors.");
          }
```

```
        }
    }

    public void calculate(String currentSourceName) throws
NotOnlyResistorsException,
                    IncompatibleSourceException, NoSourceException,
NotEnoughChordsException{
        checkAreThereSourceElements();
         CurrentSource cs = new CurrentSource(currentSourceName);
        cs.setValue(new RationalNum(1));
        cs.start = getStartNode();
        cs.end = getEndNode();
        graph.addEdge(cs);
        ga = new GraphAnalysis(graph, GraphAnalysis.CUT_ANALYSIS);
       ga.solve();
    }
    public Node getStartNode() {
        return startNode;
    }
    public Node getEndNode() {
        return endNode;
    }
}
```

Ek 5. EquivalentResistanceLatex.java

```
package com.zavrak.analysis;
import com.zavrak.graph.Graph;
import com.zavrak.lang.ast.CurrentSource;
import org.jlinalg.rational.Rational;
/**  @author Sultan ZAVRAK  */
public class EquivalentResistanceLatex implements ILatexBuilder{
   GraphAnalysis ga;
   String testSrcName;
   EquivalentResistance er;
   public EquivalentResistanceLatex(EquivalentResistance er, GraphAnalysis ga,
String testSrcName) {
      this.er = er;
      this.ga = ga;
      this.testSrcName = testSrcName;
   }
   @Override
   public String buildLatex() {
      StringBuilder laBuilder = new StringBuilder();
      GraphAnalysisLatex gal = new GraphAnalysisLatex(ga);
      laBuilder.append("\\textbf{Ön-Adım: $$I_{test} = 1 A$$ akım kaynağı} \\\\");
      laBuilder.append("\\textbf{").append(er.getStartNode()).append(" ve
").append(er.getEndNode());
      laBuilder.append(" düğümleri arasına bağlandı. }\\\\\\\\");
      //laBuilder.append("\\textbf{Devre, Graf-Kesi yöntemiyle analiz
ediliyor.}\\\\");
      //laBuilder.append(gal.buildLatex());
      laBuilder.append("\\textbf{Adım 5: Hesaplanan eşdeğer direnç}\\\\");
      laBuilder.append(equivalentResistance(testSrcName, ga.graph));
      return laBuilder.toString();

   }
```

```
    public String equivalentResistance(String currentSourceName, Graph graph) {
        StringBuilder laBuffer = new StringBuilder();
        laBuffer.append("\\begin{multline}");
        laBuffer.append("R_{eq}");
        laBuffer.append(" = \\frac{V_{I_{test}}}{I_{test}}\\\\");
        laBuffer.append("R_{eq} = ");
        Rational num = ((CurrentSource)
graph.getEdgeByName(currentSourceName)).getVoltage().getValue().getValue();
        Rational denum = ((CurrentSource)
graph.getEdgeByName(currentSourceName)).getValue().getValue();
        laBuffer.append(num.divide(denum).abs());
        laBuffer.append("\\end{multline}");
        return laBuffer.toString();
    }
}
```

Ek 6. CircuitUtils.java

```
package com.zavrak.lang.ast;
import com.zavrak.graph.Graph;
import com.zavrak.lang.ast.exp.ElementNotDeclaredException;
import java.util.ArrayList;
import java.util.Random;
/**  @author Sultan Zavrak  */
public class CircuitUtils {
   public static Circuit generateRandomCircuit(int nodeCount, int volSrcCount, int
curSrcCount, int resCount, int maxValue, int factor) throws
ElementNotDeclaredException {
      int edgeCount = volSrcCount + curSrcCount + resCount;
      if (edgeCount < nodeCount - 1) {
         System.out.println("EdgeCount >= nodecount-1 olmalı. edgecount = n-1
yapılıyor");
         edgeCount = nodeCount + 1;
      }
      if (edgeCount > nodeCount * (nodeCount - 1) / 2) {
         System.out.println("EdgeCount edgeCount <= nodeCount*(nodeCount-1) /
2 olmalı. edgecount =  nodeCount*(nodeCount-1) yapılıyor");
         edgeCount = nodeCount * (nodeCount - 1) / 2;
      }
      Edge[] edges = new Edge[edgeCount];
      Node[] nodes = new Node[nodeCount];
      Random rand = new Random();
      // random resistors
      for (int i = 0; i < resCount; i++) {
         Resistor res = new Resistor("R" + (i + 1));
         res.setValue(new RationalNum((rand.nextInt(maxValue) / factor) * factor));
         edges[i] = res;
      }
```

```
        // random voltage source
        for (int i = resCount; i < (resCount + volSrcCount); i++) {
            VoltageSource vsrc = new VoltageSource("V" + (i - resCount + 1));
            vsrc.setValue(new RationalNum((rand.nextInt(maxValue) / factor) *
factor));
            edges[i] = vsrc;
        }
        // random current source
        for (int i = resCount + volSrcCount; i < (resCount + volSrcCount +
curSrcCount); i++) {
            CurrentSource csrc = new CurrentSource("I" + (i - resCount - volSrcCount
+ 1));
            csrc.setValue(new RationalNum((rand.nextInt(maxValue) / factor) *
factor));
            edges[i] = csrc;
        }
        for (int i = 0; i < nodeCount; i++) {
            nodes[i] = new Node("N" + (i + 1));
        }
        // swap the nodes randomly
        for (int i = 0; i < nodeCount; i++) {
            int f = rand.nextInt(nodeCount);
            int s = rand.nextInt(nodeCount);
            Node temp = nodes[f];
            nodes[f] = nodes[s];
            nodes[s] = temp;
        }
        // swap the edges randomly
        for (int i = 0; i < edgeCount; i++) {
            int f = rand.nextInt(edgeCount);
            int s = rand.nextInt(edgeCount);
            Edge temp = edges[f];
            edges[f] = edges[s];
```

```
            edges[s] = temp;
        }
        ArrayList<Edge> addedEdges = new ArrayList<Edge>();
        Circuit circuit = new Circuit();
        Graph graph = new Graph();
        // add nodes to the graph
        for (int i = 0; i < nodeCount; i++) {
            graph.addNode(nodes[i]);
        }
        // add edges to the graph
        for (int i = 0; i < edgeCount; i++) {
            graph.addEdge(edges[i]);
        }
        // randomly create spanning tree  by adding the edges
        for (int i = 0; i < nodes.length; i++) {
            // first select random edge
            int choice = rand.nextInt(edgeCount);
            // eğer rastgele seçim eklenmişse bir sonrakine bak
            while (true) {
                Edge e = edges[choice];
                if (addedEdges.contains(e)) {
                    choice = (choice + rand.nextInt(edgeCount)) % edgeCount;
                    continue;
                } else {
                    //graph.connectNodes(nodes[i], nodes[(i + 1) % nodeCount]);
                    graph.getEdgeByName(e.elmName).start =
graph.getNodeByName(nodes[i].label);
                    graph.getEdgeByName(e.elmName).end =
graph.getNodeByName(nodes[(i + 1) % nodeCount].label);
                    break;
                }
            }
        }
```

```
 // add the remaining edges to the graph randomly
      for (int i = 0; i < edgeCount; i++) {
         if (!addedEdges.contains(edges[i])) {
            int s = rand.nextInt(nodeCount);
            int e = rand.nextInt(nodeCount);
            while (s == e) {
               e = (e + rand.nextInt(nodeCount)) % nodeCount;
            }
            graph.getEdgeByName(edges[i].elmName).start =
graph.getNodeByName(nodes[s].label);
            graph.getEdgeByName(edges[i].elmName).end =
graph.getNodeByName(nodes[e].label);
         }
      }
      circuit.setGraph(graph);
      circuit.ground =
graph.getNodeByName(nodes[rand.nextInt(nodeCount)].label);
      return circuit;
   }
}
```

ÖZGEÇMİŞ

1987 yılında Trabzon'nun Akçaabat ilçesinde doğdu. İlköğrenimini Mehmet Akif Ersoy İlköğretim Okulu'nda, orta öğrenimini ise Trabzon Fatih Lisesi'nde tamamladı. 2004 yılında Karadeniz Teknik Üniversitesi Mühendislik Fakültesi Bilgisayar Mühendisliği Bölümü'nde lisans programına başladı ve 2010 yılında bu bölümden mezun oldu.

Kısa bir süre özel sektörde çalıştıktan sonra, Karadeniz Teknik Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı'nda yüksek lisans programına ve ayrıca Düzce Üniversitesi Mühendislik Fakültesi Bilgisayar Mühendisliği Bölümünde Araştırma Görevlisi olarak çalışmaya başladı. Halen bu görevini sürdürmektedir. Yabancı dil olarak İngilizce bilmektedir.