Minimum Örten Ağaç Algoritmaları:

Boruvka

vs

Kruskal

BAHAR DÖNEMİ,2019

Onur Yurteri

15253070

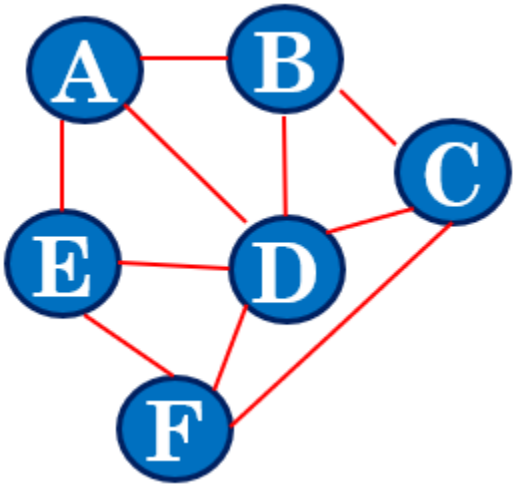
Giriş

Bu çalışma minimum örten ağaç algoritmalarından olan ‘Boruvka’ ve ‘Kruskal’ algoritmalarının seçilen problem üzerinde uygulanma yöntemini ve algoritmaların Julia dili kullanarak implementasyonunu baz alır.

Minimum Örten Ağaçlar

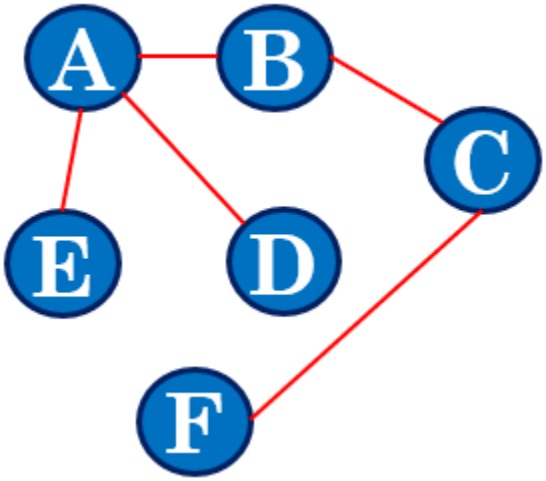
Minimum örten ağaçlar, ağırlıklı bir ağda (weighted graph, yani her düğümü birbirine bağlayan yolların maliyeti -ağırlığı- olması durumu), bütün düğümleri dolaşan en kısa yolu verir. [i]

Örneğin, ***G=(V,E)*** bağlantılı çizgesi



G=(V,E) Çizgesi

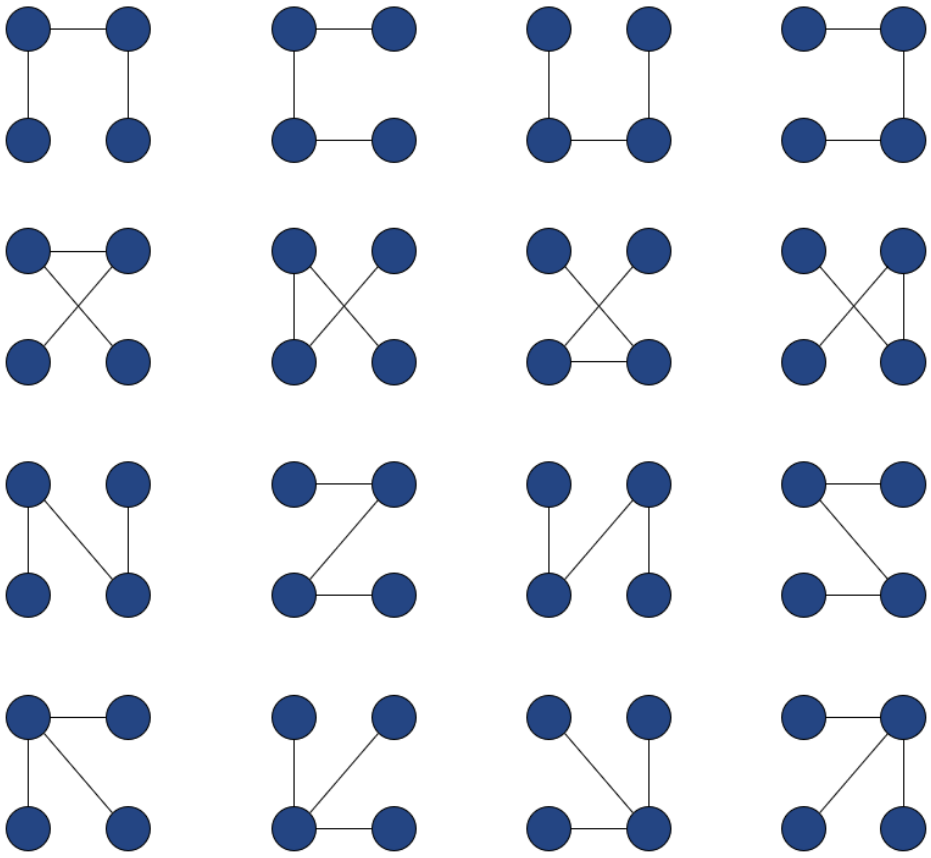
İken Örteni ise;



G için Örten Ağaç

Bir ağaçtır. ***V*** kümesinin tüm elemanları bu ağacın düğümleridir.

Örten bir ağaçta kiriş (edge) sayısı $|V|-1$'dir.
Anlaşılabileceği üzere bir çizgenin çok sayıda örten ağacı olabilir.
n elemanlı bir V kümesi için örten ağaç sayısı $n^{(n-2)}$'dir.(Cayley formülü)[ii]
Örneğin; 4 elemanlı bir V kümesinin 16 farklı örten ağacı bulunur.



4 elemanlı bir V kümesi için Örten Ağaçlar

Fark ediyoruz ki kümemizdeki eleman sayımız arttıkça problemimiz sadece 'örten ağaçları bulmak' iken bile çok büyük bir zaman problemiyle karşı karşıyayız.
Örneğin; 100 elemanlı bir V kümesinin örten ağaçlarını bulmak için işlem süremizi ***1 nanosaniye (=(10^{-9}) saniye)*** kabul edersek,

- **$100^{98} = 10^{196}$**
- **Bir yılda: $315576*10^6$ saniye**
- **$315576*10^{15} < 10^{21}$ nanosaniye**

100 köşeli çizgenin tüm örten ağaçlarını ancak **10^{175}** yıldan daha uzun sürede bulabiliriz.
Problemimize kirişler arası *ağırlıkları(weight)* eklediğimizde ve karşılaştırma operatörüyle ilerlediğimizde, brute-force yöntemlerin gerçek hayat senaryolarında çözüm getiremeyeceği aşikardır.

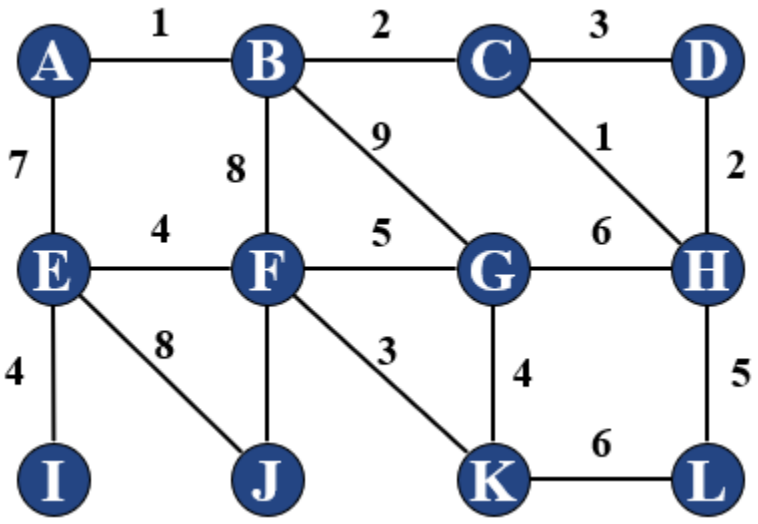
Boruvka'nın Algoritması (Sollin Algoritması)

Boruvka algoritması toplam kiriş ağırlıklarını minimize etmeye edip amacı olan minimum örten ağacı bulmaya çalışan bir algoritmadır. 1926'da geliştirildiğinde; Otakar Boruvka bir elektrik ağının en uygun maliyetle tasarlanması için kullanmıştır.[iii]

```
Input: A graph G whose edges have distinct weights
 Initialize a forest F to be a set of one-vertex trees, one for each vertex
of the graph.
 While F has more than one component:
   Find the connected components of F and label each vertex of G by its
component
   Initialize the cheapest edge for each component to "None"
   For each edge uv of G:
     If u and v have different component labels:
       If uv is cheaper than the cheapest edge for the component of u:
         Set uv as the cheapest edge for the component of u
       If uv is cheaper than the cheapest edge for the component of v:
         Set uv as the cheapest edge for the component of v
    For each component whose cheapest edge is not "None":
      Add its cheapest edge to F
  Output: F is the minimum spanning forest of G.
```

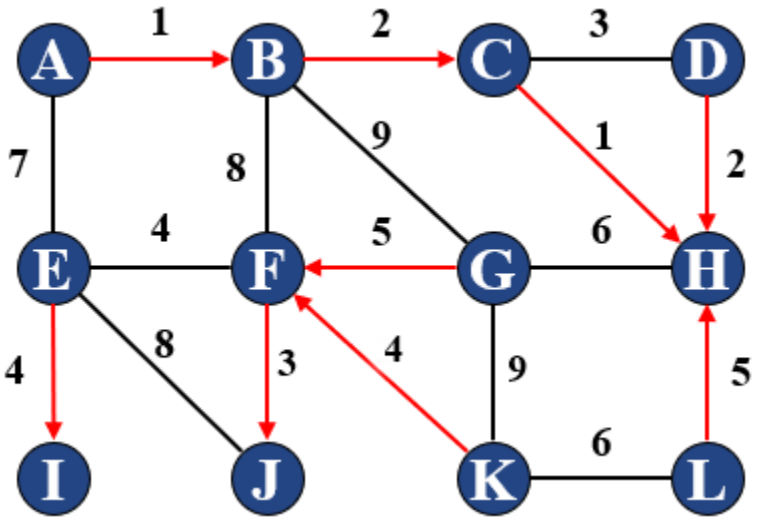
[iv]

Nasıl çalıştığını aşağıdaki örnekteki 12 elemanlı ağda öğrenelim;

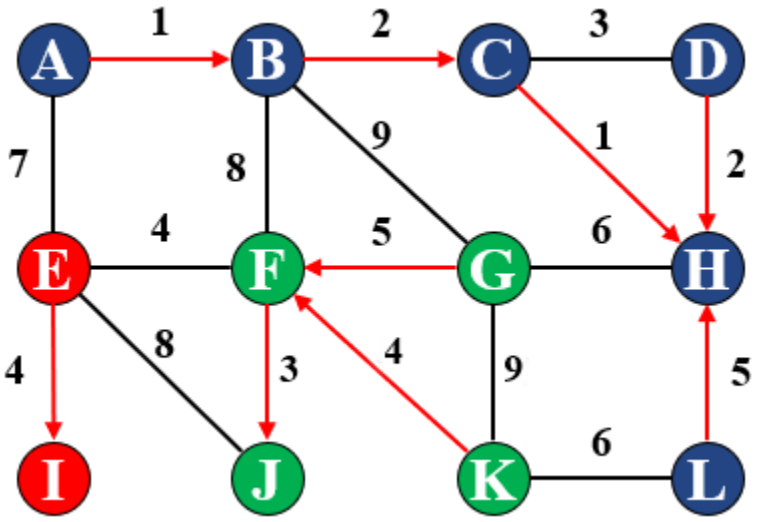


Sahip olduğumuz elemanları tek tek dolaşıp sahip oldukları en küçük ağırlıklı kirişleri;

1. Her eleman için sadece bir kiriş
2. Kiriş ağırlıkları eşitse (E elemanı ağırlığı 4 olan iki kirişe sahip), birini en düşük ağırlıklı kabul edip seçiyoruz. Tamamen keyfi olmakla birlikte, tüm süreç boyunca aynı şekilde seçiyoruz. Bu örnekte, eşit olması durumunda soldaki kirişi en düşük kabul edip seçiyoruz.
3. Her zaman en düşük ağırlıklı kirişi seçiyoruz, *daha önce seçili olsa bile*. **Kesinlikle** seçiliden bir sonraki en küçüğü seçmek gibi bir şey yapmıyoruz.

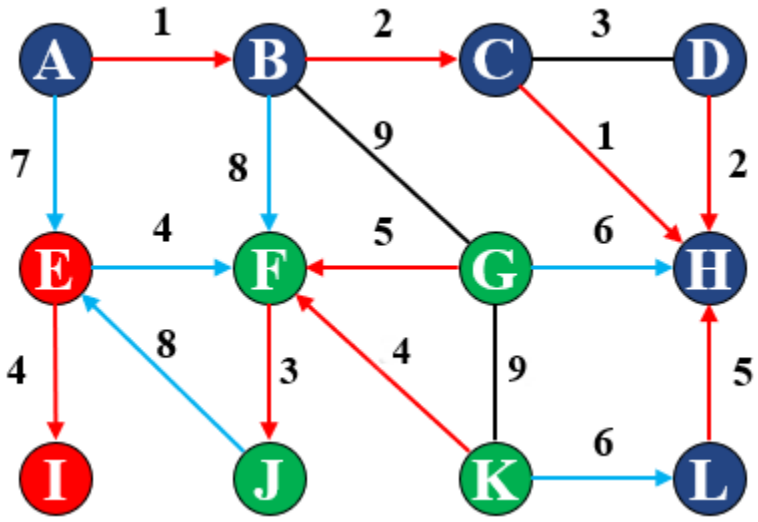


Oluşan alt ağaçları belirliyoruz. Birbiriyle bağı olan bu elemanlara *komponent(component)* diyeceğiz.

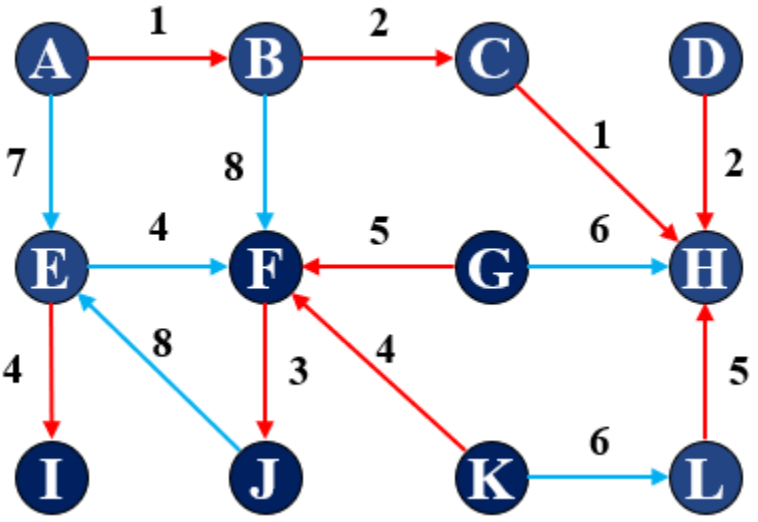


Algoritmayı her komponent için tekrarlıyoruz(Farklı renklendirilmiş ağaçlar). Bu sefer her eleman için **komponent dışına çıkan** en düşük ağırlıklı kirişi seçiyoruz. Örneğin, ABCDHL komponenti (mavi elemanlar). A elemanı için, komponent dışına çıkan en düşük ağırlıklı kiriş 7 (1 ağırlıklı kiriş komponent içinde olduğu için). Bu kirişleri maviyle işaretleyelim.

1. *Dışarıya* çıkan en düşük ağırlıklı kirişi seçtiğinizden emin olun.
2. İlk iterasyonda olduğu gibi, kiriş daha önce seçilmişse atlıyoruz.



Eğer tüm elemanlar bağlıysa ve elimizde tek komponent kaldıysa algoritmamızın sonuna geldik demektir. Bir iterasyona daha ihtiyacımız yok.
Bağlı olmayan kirişleri çıkarttığımızda son minimum örten ağacımız;



Şeklinde olacaktır.

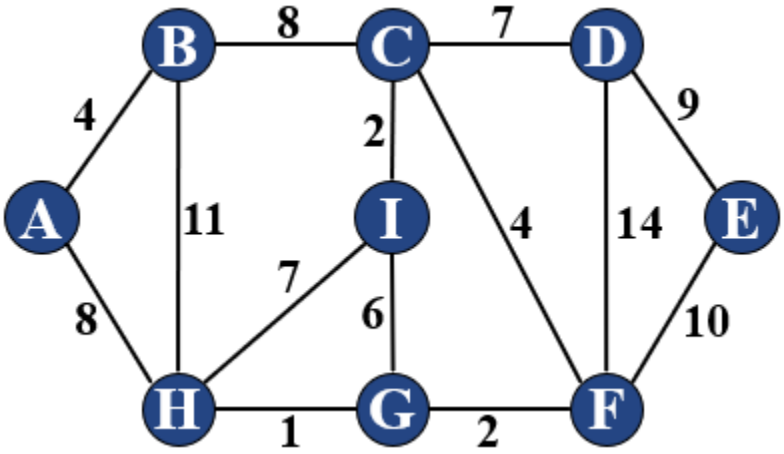
Kruskal'ın Algoritması

Kruskal ise kiriş tabanlı bir algoritmadır. Aç gözlü bir algoritma olduğundan, iterasyondaki ağaç için herhangi bir *çevrim(cycle)* yaratmayan -en düşük ağırlıklı kirişi- seçer.

```
KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3    MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5    if FIND-SET(u) ≠ FIND-SET(v):
6       A = A ∪ {(u, v)}
7       UNION(u, v)
8 return A
```

v

Nasıl çalıştığını aşağıdaki örnekteki 9 elemanlı ağda öğrenelim;

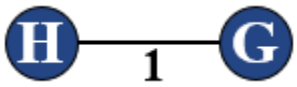


Öncelikle kirişlerimizi ağırlıklarına göre artan şekilde sıralıyoruz.

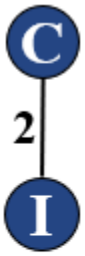
Ağırlık	Nereden	Nereye
1	H	G
2	I	C

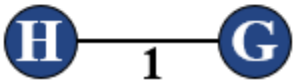
3	G	F
4	A	B
5	C	F
6	I	G
7	C	D
8	H	I
9	A	H
10	B	C
11	D	E
12	F	E
13	B	H
14	D	F

H-G kirişine bak: çevrim oluşmuyor, ekle

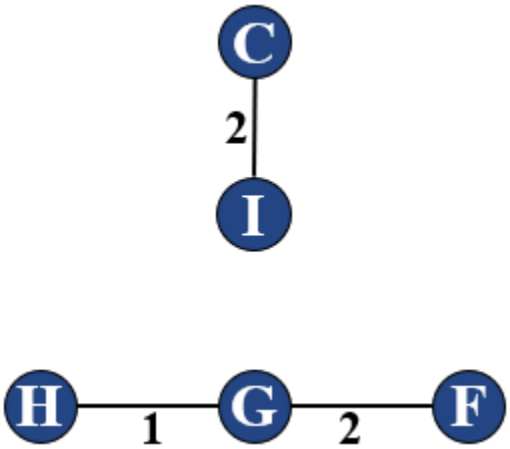


C-I kirişine bak: çevrim oluşmuyor, ekle

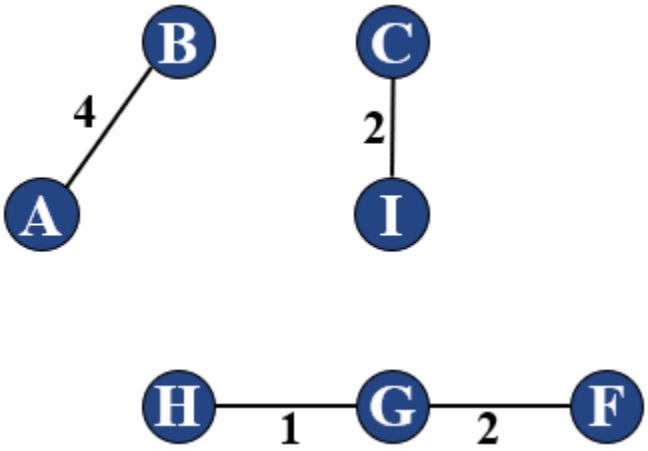




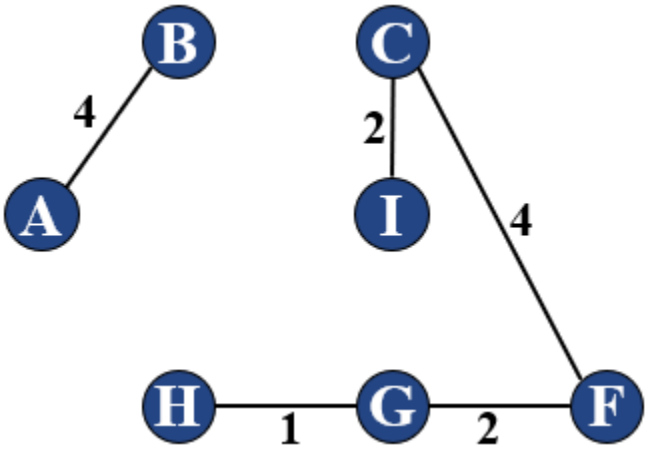
C-I kirişine bak: çevrim oluşmuyor, ekle



A-B kirişine bak: çevrim oluşmuyor, ekle

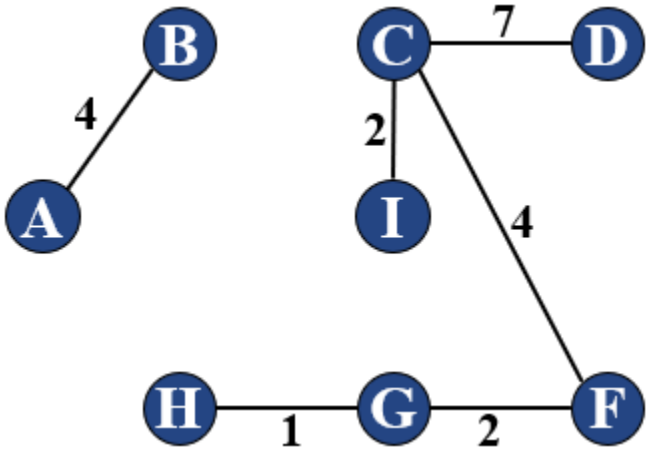


C-F kirişine bak: çevrim oluşmuyor, ekle

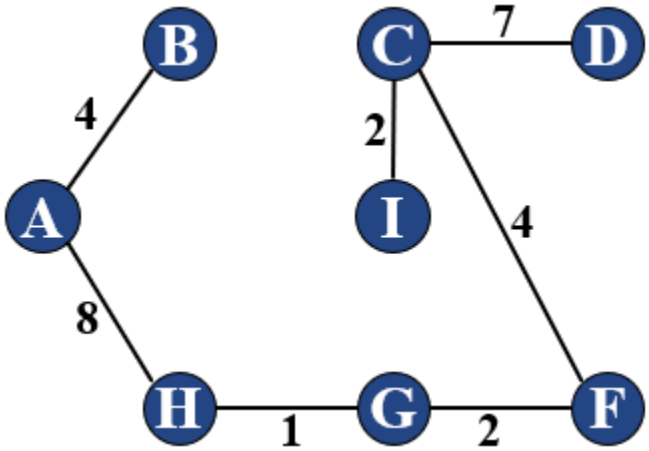


I-G kirişine bak: çevrim oluşuyor, **atla**

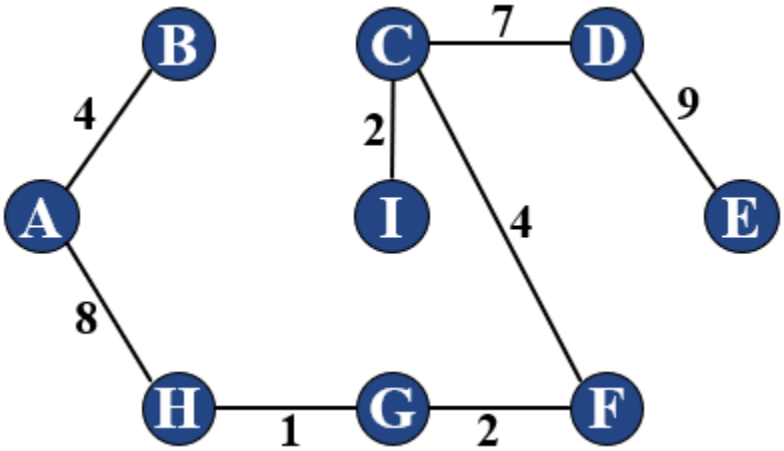
C-D kirişine bak: çevrim oluşmuyor, ekle



H-I kirişine bak: çevrim oluşuyor, **atla**
A-H kirişine bak: çevrim oluşmuyor, ekle



B-C kirişine bak: çevrim oluşuyor, **atla**
D-E kirişine bak: çevrim oluşmuyor, ekle



Eklenen kiriş sayısı ***|V|-1***'e eşit olduğu için, minimum örten ağaç tanımını sağlamış oluyoruz. Bu noktadan sonra devam etmemize gerek yok.

En son iterasyonla birlikte minimum örten ağacımızı elde ettik.

Çalışma Hakkında

Çalışmada Boruvka ve Kruskal algoritmaları Julia diline implement edilecek ve kullanıcının ağaç ve kenarları oluşturup tanımlarken kolaylık sağlaması için obje benzeri structlar oluşturulacak ve kullanıcı kolaylığı sağlanacak:

```
myEdges= Edge[]
push!(myEdges, Edge(KAYNAK,HEDEF,AĞIRLIK))
myGraph=Graph(V,E,myEdges)
```

Program verilen graph üzerinde implement edilen Boruvka ve Kruskal algoritmalarını çalıştıracak. Ve elde ettiği sonuçlarla birlikte ne kadar süre içerisinde sonuca ulaşıldığını kullanıcıya bildirecek. Projeyle birlikte örnek olarak oluşturulmuş farklı graphlar teslim edilecek.

Proje;

- Utilities.jl //Yardımcı fonksiyonları içeren Julia dosyası
- Graph.jl //Graph ve Edge tiplerini içeren Julia dosyası
- Kruskal.jl
- Boruvka.jl
- Main.jl
- Documentation.txt

Dosyalarını içerecektir.

Algoritma sonuçlanma süreleri ve bellek kullanım miktar ölçümü Julia içerisindeki *@time* fonksiyonu ile gerçekleştirilecektir.

Julia İmplementasyonu

Julia dosyaları arasında algoritmaları içeren ‘Boruvka.jl’ ve ‘Kruskal.jl’ dosyaları bulunmaktadır. Algoritmalar argüman olarak Graph tipinde ‘struct’ almakta ve bu struct yapısı ‘Graph.jl’ dosyasında tanımlanmıştır.
İki algoritmanın karşılaştırılması hedeflendiğinden, algoritmaların kendi içlerinde kullandığı ‘subset’ arama ve birleştirme fonksiyonları, aynıdır ve ‘Subset’ yapısıyla birlikte ‘Utilities.jl’ dosyasında bulunmaktadır.
‘examples’ klasörü altında karşılaştırmalar için kullanılmış örnek 5 graph bulunmaktadır. ‘Main.jl’ dosyasında sonuçların yazdırılması ve minimum örten ağacı aranan graph’ın include edilmesi gerekmektedir.

Graph.jl

Bu dosyada graph’ların oluşturmasını kolaylaştırmak ve kod’un okunabilirliğini arttırmak amacıyla tanımlanmış ‘Edge’ ve ‘Graph’ struct’ları bulunmaktadır.

'Edge': Her bir edge'in src:source (kaynak, başlangıç) ve dest:destination (hedef, varış) düğümü vardır. Düğümler unique integer değerleriyle temsil edilmektedir.

'Graph': Her bir graph için; V, graph'daki eleman/düğüm/komponent sayısını; E, graph'daki edge (kiriş/kenar) sayısını tutar. 'edges' ise graph'a ait kirişleri tutan 'Edge' tipinde bir listedir.

Düğümlerin teker teker tanımlanmasına ihtiyaç yoktur. Algoritmalar 'edge' struct'ının kaynak-hedef özelliklerini kullanmaktadır.

```
struct Edge
    src::Int
    dest::Int
    weight::Int
end

struct Graph
    V::Int
    E::Int
    edges::Array{Edge}
end
```

Utilities.jl

Bu dosyada, algoritmların araç olarak kullandığı subset arama ve subset birleştirme fonksiyonları bulunmaktadır. Algoritmaların karşılaştırması hedeflendiğinden, iki algoritmanın farklı subset arama ve birleştirme fonksiyonları kullanması karşılaştırmada bir algoritmanın haksız kazanç elde edebileceği anlamına geldiğinden ve bu belirsizliği ortadan kaldırmak amacıyla ortak arama ve birleştirme fonksiyonu kullanılmıştır.

```
mutable struct Subset
    parent::Int
    rank::Int
end

function findSubset(subsets::Array{Subset}, i::Int)
    if (subsets[i].parent != i)
      subsets[i].parent =findSubset(subsets, subsets[i].parent);
    end
    return subsets[i].parent;
end

function UnionSubset(subsets::Array{Subset}, x::Int, y::Int)

    xroot = findSubset(subsets, x);
    yroot = findSubset(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    elseif (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else
        subsets[yroot].parent = xroot;
        subsets[xroot].rank=subsets[xroot].rank+1;
    end
end
```

Boruvka.jl

Boruvka'nın algoritması Julia diline implement edilmiştir. Argüman olarak 'Graph' struct'ı alan, geriye MST kiriş listesini ve ağırlığını döndüren '*boruvkaMST'* fonksiyonuna sahiptir.

```
include("Graph.jl")
include("Utilities.jl")

function boruvkaMST(graph::Graph)
    V=graph.V;
    E=graph.E;
    edges=graph.edges;
    subsets=Subset[];
    cheapest=Int[];
    result=Edge[];
    for i=1:V
        push!(subsets,Subset(i,0));
        push!(cheapest, -1);
    end
    numTrees=V;
    MSTweight=0;
    while numTrees>1
        for i=1:V
            cheapest[i]=-1;
        end
        for i=1:E
            set1= findSubset(subsets,edges[i].src);
            set2= findSubset(subsets,edges[i].dest);
            if set1 != set2
                if cheapest[set1] == -1 || edges[cheapest[set1]].weight > edges[i].weight
                    cheapest[set1]=i;
                end
                if cheapest[set2] == -1 || edges[cheapest[set2]].weight > edges[i].weight
                    cheapest[set2]=i;
                end
            end
        end
        for i=1:V
            if cheapest[i] != -1
                set1=findSubset(subsets,edges[cheapest[i]].src);
                set2=findSubset(subsets,edges[cheapest[i]].dest);
                if set1 != set2
                    MSTweight= MSTweight + edges[cheapest[i]].weight;
                    push!(result,Edge(edges[cheapest[i]].src,edges[cheapest[i]].dest,edges[cheapest[i]].weight))
                    UnionSubset(subsets,set1,set2);
                    numTrees=numTrees-1;
                end
            end
        end
    end
    return result, MSTweight;
end
```

Kruskal.jl

Kruskal'ın algoritması Julia diline implement edilmiştir. Argüman olarak 'Graph' struct'ı alan, geriye MST kiriş listesini ve ağırlığını döndüren '*kruskalMST'* fonksiyonuna sahiptir.

```
include("Graph.jl")
include("Utilities.jl")

function kruskalMST(graph::Graph)
    V=graph.V;
    edges=graph.edges;
    result=Edge[];
    iter=1;
    subsets=Subset[];
    MSTweight=0;
    sort!(myEdges, by = v -> v.weight, rev=false);
    for i=1:V
        push!(subsets,Subset(i,0));
    end
    while length(result) < V-1
        nextEdge=edges[iter];
        iter+=1;
        x=findSubset(subsets, nextEdge.src);
        y=findSubset(subsets, nextEdge.dest);
        if x != y
            push!(result,nextEdge);
            MSTweight= MSTweight + nextEdge.weight;
            UnionSubset(subsets, x, y);
        end
    end
    return result, MSTweight;
end
```

Örnekler ve Main.jl

Projeyle birlikte 'examples' klasörü altında aşağıdaki 5 farklı julia dosyası ile modellenmiş graph'lar bulunmaktadır; [vi]

- CP 4.10
- CP 4.14
- K5
- Rail
- Tessellation

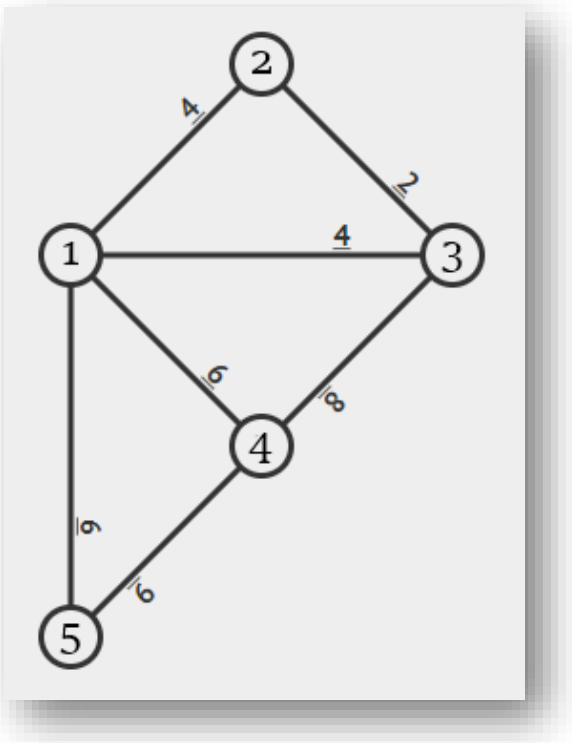
Kullanılmak istenen örneği ‘Main.jl’ dosyası içindeki yorum satırlarını kaldırarak ekleyebilir ya da istediğiniz çizgeyi, örneklerdeki formatla oluşturup çözüm alabilirsiniz.
‘Main.jl’ dosyasında ayrıca algoritmaların çağırılıp kullanıcının sonuçları görüntülemesi sağlanmıştır.

```
include("Graph.jl")
include("Boruvka.jl")
include("Kruskal.jl")

myEdges= Edge[];

#UNCOMMENT ONE OF THE EXAMPLES BELOW
include("examples/cp-4.10.jl");
#include("examples/cp-4.14.jl")
#include("examples/k5.jl")
#include("examples/rail.jl")
#include("examples/tessellation.jl")

println("Given edges: ", myEdges);

println("--");

println("Boruvka");
(boruvkaEdges, boruvkaWeight) = @time boruvkaMST(myGraph);
println(boruvkaEdges);
println("Total Weight: ", boruvkaWeight);

println("--");

println("Kruskal");
(kruskalEdges, kruskalWeight) = @time boruvkaMST(myGraph);
println(kruskalEdges);
println("Total Weight: ", kruskalWeight);
```

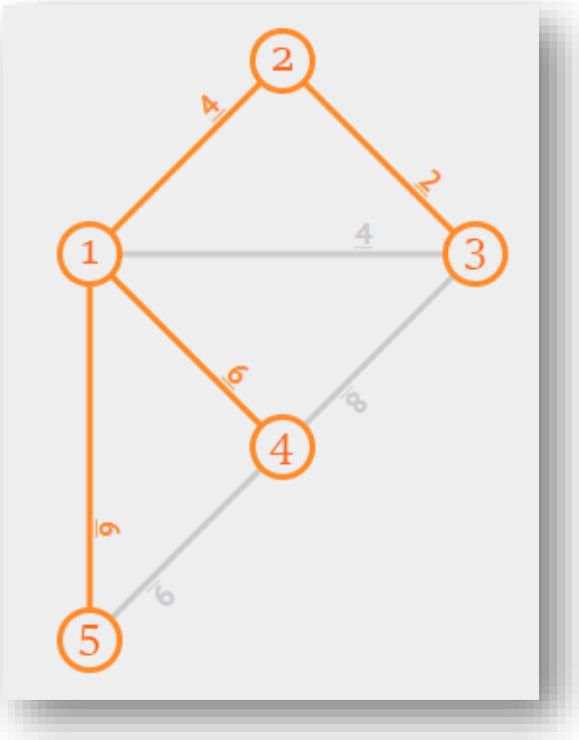
Yukarıdaki örnekte *include(“examples/cp-4.10.jl”)* satırı eklenerek programa dahil edilmiştir.

CP 4.10

5 elemanlı 7 kirişli bir çizgedir.



Ve minimum örten ağacı aşağıdaki gibidir:

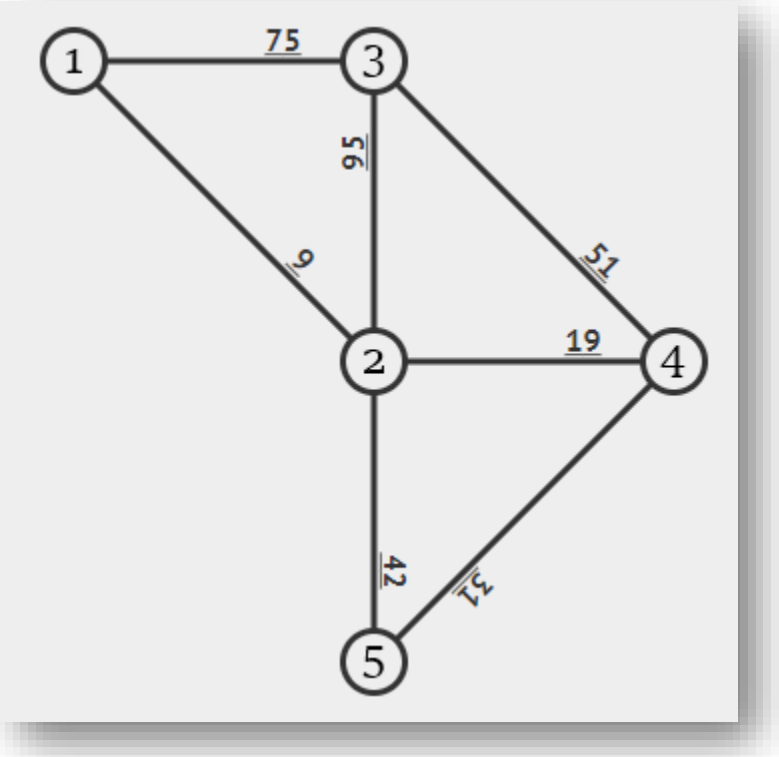


‘examples’ klasörü altında bu çizgenin modellemesini ise şu şekilde bulabilirsiniz:

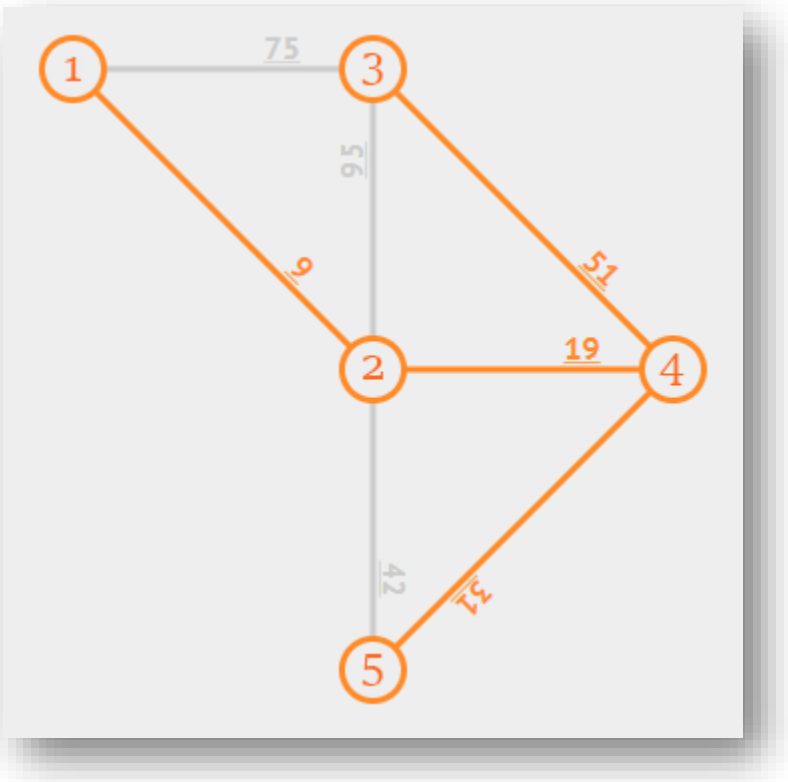
```
#CP 4.10
push!(myEdges, Edge(1,3,4))
push!(myEdges, Edge(1,2,4))
push!(myEdges, Edge(1,4,6))
push!(myEdges, Edge(1,5,6))
push!(myEdges, Edge(2,3,2))
push!(myEdges, Edge(3,4,8))
push!(myEdges, Edge(4,5,9))
myGraph=Graph(5,length(myEdges), myEdges)
```

CP 4.14

5 elemanlı 7 kirişli bir çizgedir.



Ve minimum örten ağacı aşağıdaki gibidir:

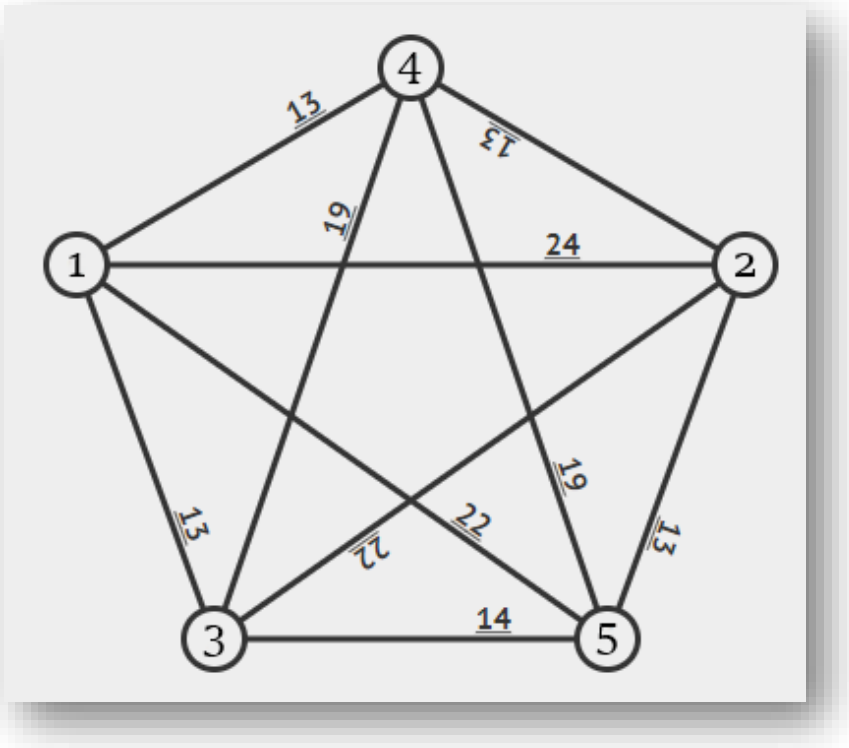


'examples' klasörü altında bu çizgenin modellemesini ise şu şekilde bulabilirsiniz:

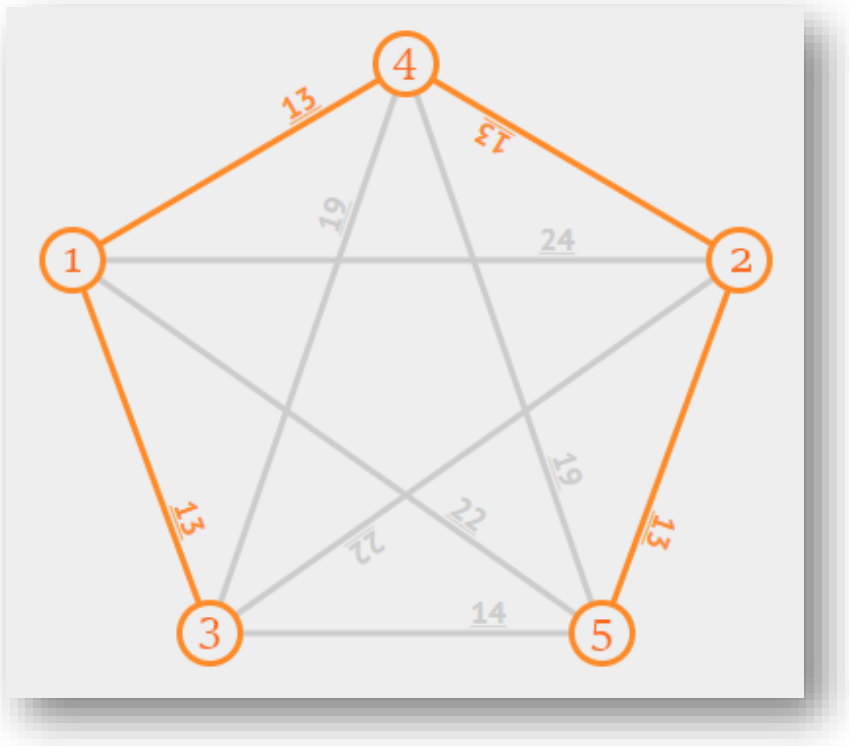
```
#CP 4.14
push!(myEdges, Edge(1,2,9))
push!(myEdges, Edge(1,3,75))
push!(myEdges, Edge(2,3,95))
push!(myEdges, Edge(2,4,19))
push!(myEdges, Edge(2,5,42))
push!(myEdges, Edge(3,4,51))
push!(myEdges, Edge(4,5,31))
myGraph=Graph(5,length(myEdges), myEdges)
```

K5

5 elemanlı 10 kirişli bir çizgedir.



Ve minimum örten ağacı aşağıdaki gibidir:

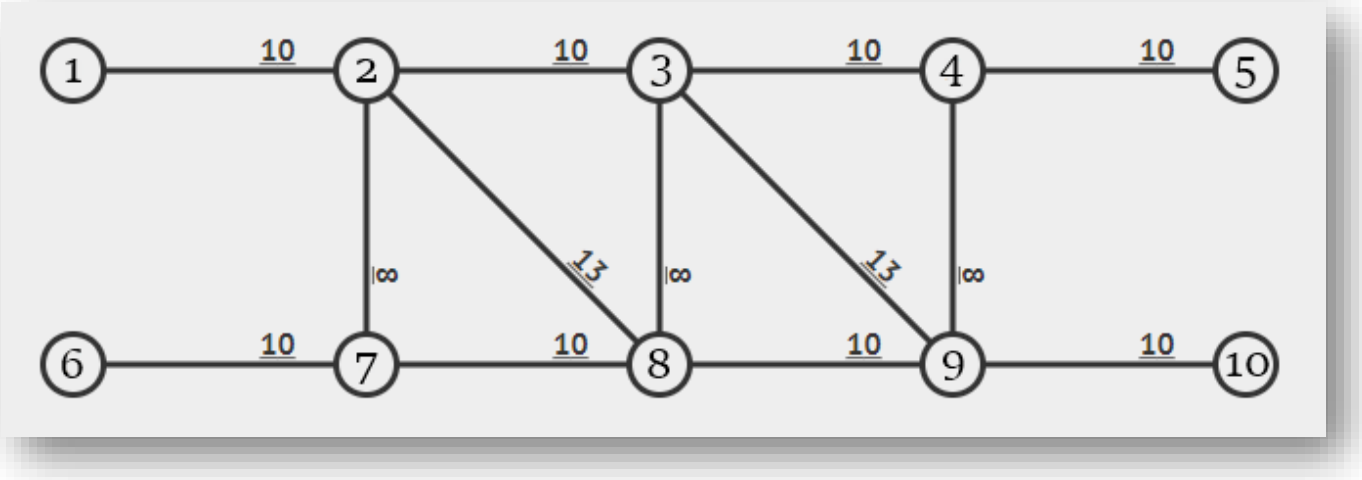


‘examples’ klasörü altında bu çizgenin modellemesini ise şu şekilde bulabilirsiniz:

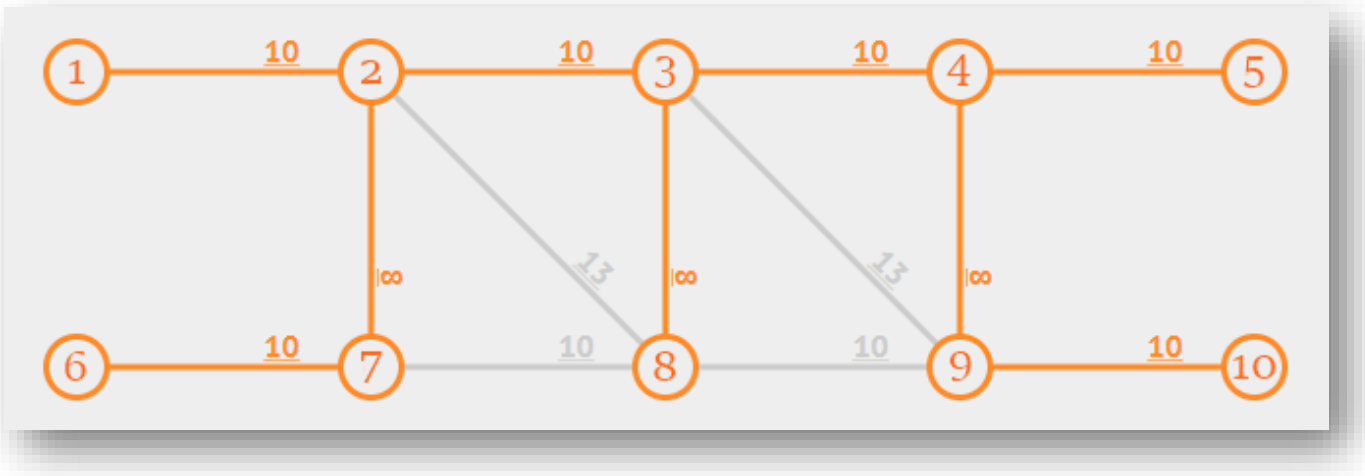
```
#K5
push!(myEdges, Edge(1,2,24))
push!(myEdges, Edge(1,3,13))
push!(myEdges, Edge(1,4,13))
push!(myEdges, Edge(1,5,22))
push!(myEdges, Edge(2,3,22))
push!(myEdges, Edge(2,4,13))
push!(myEdges, Edge(2,5,13))
push!(myEdges, Edge(3,4,19))
push!(myEdges, Edge(3,5,14))
push!(myEdges, Edge(4,5,19))
myGraph=Graph(5,length(myEdges), myEdges)
```

Rail

10 elemanlı 13 kirişli bir çizgedir.



Ve minimum örten ağacı aşağıdaki gibidir:

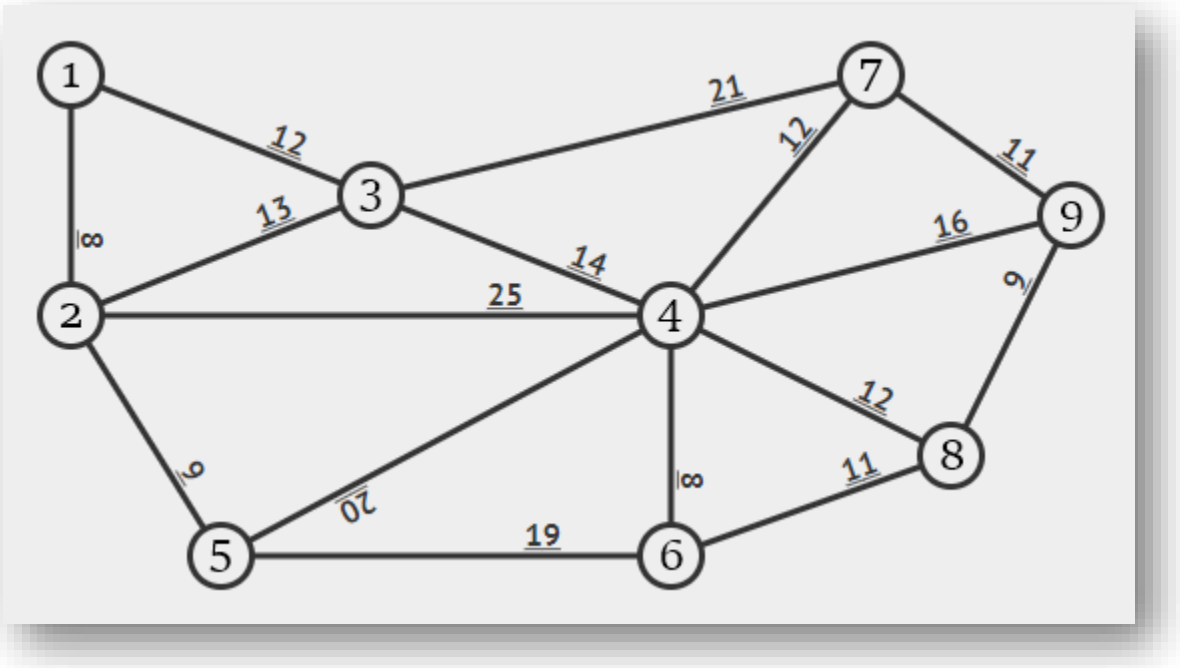


‘examples’ klasörü altında bu çizgenin modellemesini ise şu şekilde bulabilirsiniz:

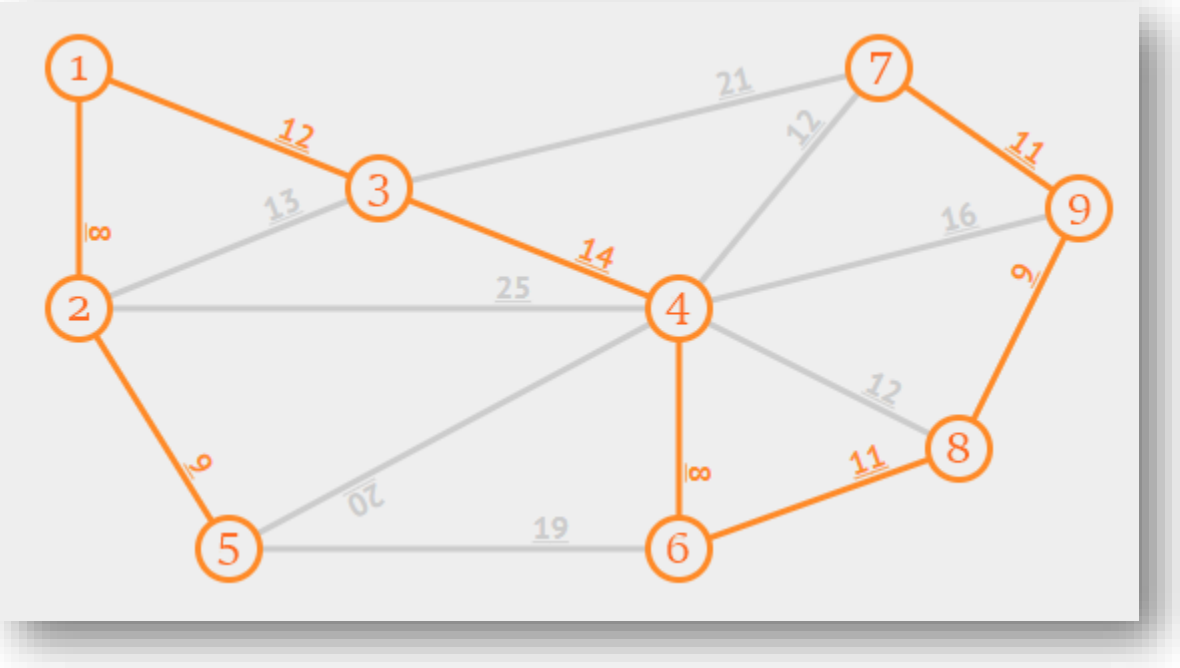
```
#Rail
push!(myEdges, Edge(1,2,10))
push!(myEdges, Edge(2,3,10))
push!(myEdges, Edge(2,7,8))
push!(myEdges, Edge(2,8,13))
push!(myEdges, Edge(3,4,10))
push!(myEdges, Edge(3,8,8))
push!(myEdges, Edge(3,9,13))
push!(myEdges, Edge(4,5,10))
push!(myEdges, Edge(4,9,8))
push!(myEdges, Edge(6,7,10))
push!(myEdges, Edge(7,8,10))
push!(myEdges, Edge(8,9,10))
push!(myEdges, Edge(9,10,10))
myGraph=Graph(10,length(myEdges), myEdges)
```

Tessellation

9 elemanlı 16 kirişli bir çizgedir.



Ve minimum örten ağacı aşağıdaki gibidir:



'examples' klasörü altında bu çizgenin modellemesini ise şu şekilde bulabilirsiniz:

```
#Tessellation
push!(myEdges, Edge(2,3,13))
push!(myEdges, Edge(1,3,12))
push!(myEdges, Edge(1,2,8))
push!(myEdges, Edge(2,4,25))
push!(myEdges, Edge(2,5,9))
push!(myEdges, Edge(3,4,14))
push!(myEdges, Edge(3,7,21))
push!(myEdges, Edge(4,5,20))
push!(myEdges, Edge(4,7,12))
push!(myEdges, Edge(4,6,8))
push!(myEdges, Edge(4,8,12))
push!(myEdges, Edge(4,9,16))
push!(myEdges, Edge(5,6,19))
push!(myEdges, Edge(6,8,11))
push!(myEdges, Edge(7,9,11))
push!(myEdges, Edge(8,9,9))
myGraph=Graph(9,length(myEdges), myEdges)
```

Örnek Program Çıktısı ve Analizler

İstediğiniz çizgeyi 'Main.jl' içinde include ettikten sonra (main.jl içerisinde *myEdges* listesi içine kiriş push'layarak da yapılabilir) program verilen kiriş listesini algoritmalara gönderir ve Julia dilindeki *@time* fonksiyonu ile sonuçlanma süresi ve memory kullanımını ölçer ve sonuçları yazdırır. Aşağıda CP 4.10 için çıktı örneği verilmiştir.

```
Given edges: Edge[Edge(1, 3, 4), Edge(1, 2, 4), Edge(1, 4, 6), Edge(1, 5, 6), Edge(2, 3, 2), Edge(3, 4, 8), Edge(4, 5, 9)]
--
Boruvka
  0.138184 seconds (107.90 k allocations: 5.434 MiB, 18.01% gc time)
Edge[Edge(1, 3, 4), Edge(2, 3, 2), Edge(1, 4, 6), Edge(1, 5, 6)]
Total Weight: 18
--
Kruskal
  0.000010 seconds (76 allocations: 2.750 KiB)
Edge[Edge(1, 3, 4), Edge(2, 3, 2), Edge(1, 4, 6), Edge(1, 5, 6)]
Total Weight: 18
[Finished in 8.406s]
```

Analizler için her bir örnek 10'ar defa çalıştırılıp sonuçlanma süreleri, atamalar ve bellek kullanımları için ortalamaları alınmıştır.

Aşağıda örnekler üzerindeki ölçümleri, en altta ortalama değerleriyle birlikte bulabilirsiniz

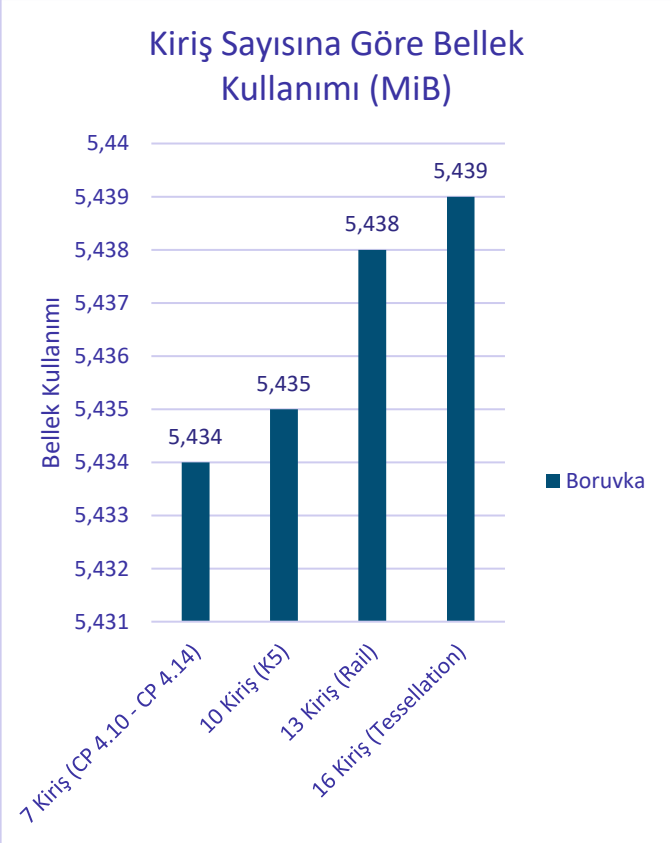
CP 4.10					
Boruvka			Kruskal		
Time	Allocations	Memory	Time	Allocations	Memory
0,098078	107900	5,434	0,000014	76	0,002686
0,101186	107900	5,434	0,000055	76	0,002686
0,116837	107900	5,434	0,00001	76	0,002686
0,111439	107900	5,434	0,00001	76	0,002686
0,108322	107900	5,434	0,000012	76	0,002686
0,106386	107900	5,434	0,000011	76	0,002686
0,104961	107900	5,434	0,000011	76	0,002686
0,103384	107900	5,434	0,00001	76	0,002686
0,107114	107900	5,434	0,000011	76	0,002686
0,097989	107900	5,434	0,000062	76	0,002686
0,1055696	**107900**	**5,434**	**0,0000206**	76	**0,002686**

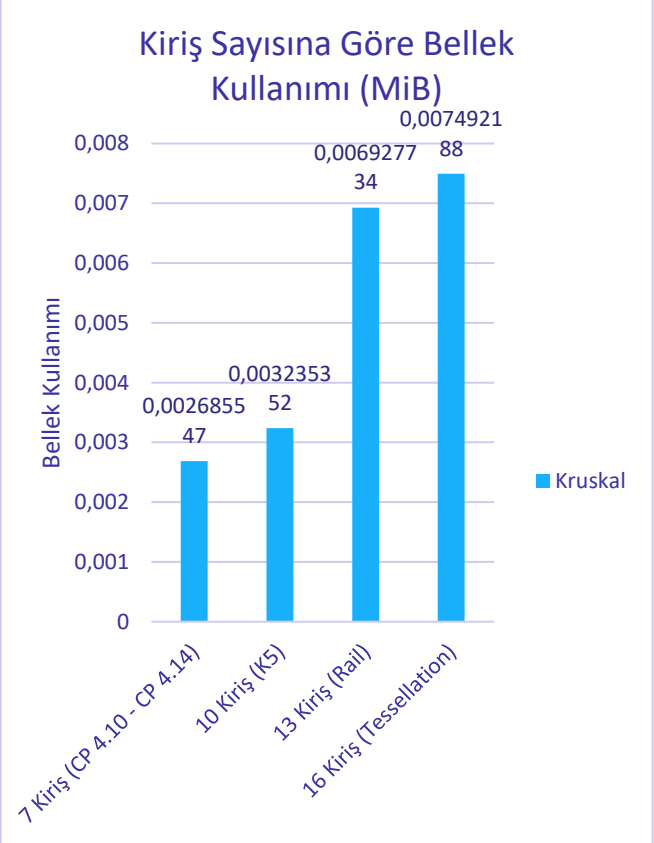
CP 4.14					
Boruvka			Kruskal		
Time	Allocations	Memory	Time	Allocations	Memory
0,096501	107900	5,434	0,00001	76	0,002686
0,102188	107900	5,434	0,000042	76	0,002686
0,104075	107900	5,434	0,00001	76	0,002686
0,119067	107900	5,434	0,000011	76	0,002686
0,102411	107900	5,434	0,000038	76	0,002686
0,099645	107900	5,434	0,00001	76	0,002686
0,097524	107900	5,434	0,00001	76	0,002686
0,099512	107900	5,434	0,000032	76	0,002686
0,103945	107900	5,434	0,00001	76	0,002686
0,101922	107900	5,434	0,000011	76	0,002686
0,102679	**107900**	**5,434**	**0,0000184**	**76**	**0,002686**

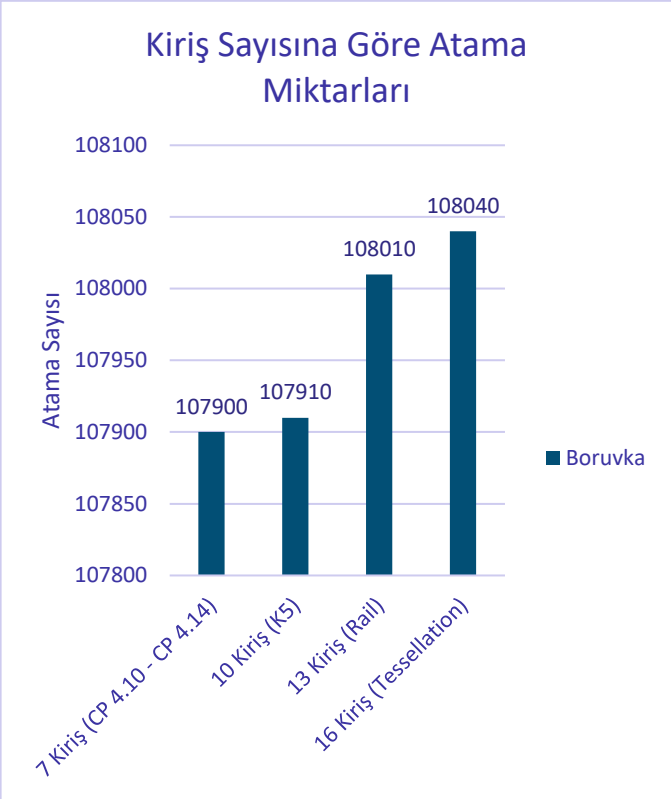
K5					
Boruvka			Kruskal		
Time	Allocations	Memory	Time	Allocations	Memory
0,104452	107910	5,435	0,000011	94	0,003235
0,101988	107910	5,435	0,000013	94	0,003235
0,104287	107910	5,435	0,00001	94	0,003235
0,107814	107910	5,435	0,000011	94	0,003235
0,105574	107910	5,435	0,000011	94	0,003235
0,098798	107910	5,435	0,000022	94	0,003235
0,106755	107910	5,435	0,000011	94	0,003235
0,097914	107910	5,435	0,000011	94	0,003235
0,103328	107910	5,435	0,000011	94	0,003235
0,099787	107910	5,435	0,000011	94	0,003235
0,1030697	**107910**	**5,435**	**0,0000122**	**94**	**0,003235**

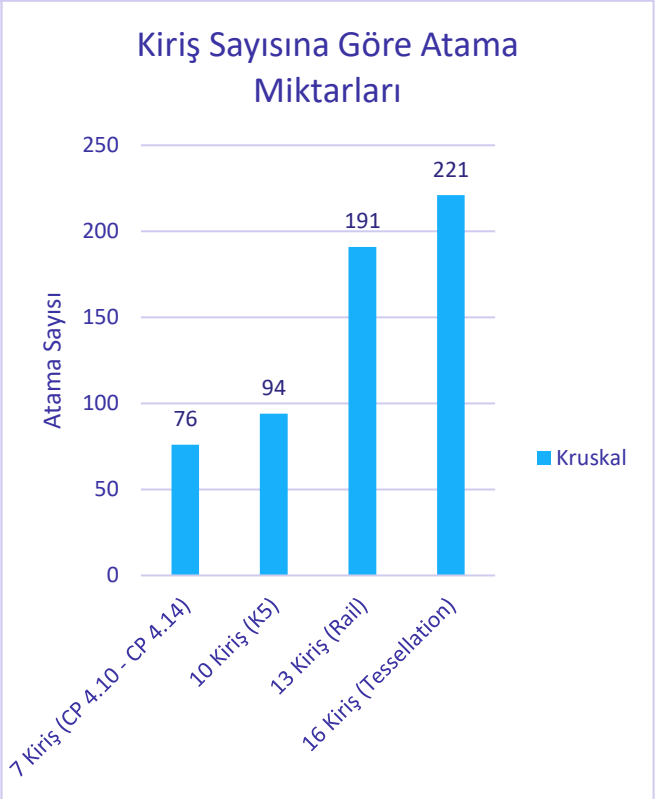
RAIL					
Boruvka			Kruskal		
Time	Allocations	Memory	Time	Allocations	Memory
0,105461	108010	5,438	0,000016	191	0,006928
0,111127	108010	5,438	0,000015	191	0,006928
0,101095	108010	5,438	0,000016	191	0,006928
0,103992	108010	5,438	0,000036	191	0,006928
0,109423	108010	5,438	0,000015	191	0,006928
0,102072	108010	5,438	0,000014	191	0,006928
0,099707	108010	5,438	0,000014	191	0,006928
0,106114	108010	5,438	0,000016	191	0,006928
0,104067	108010	5,438	0,000015	191	0,006928
0,104428	108010	5,438	0,000017	191	0,006928
0,1047486	**108010**	**5,438**	**0,0000174**	**191**	**0,006928**

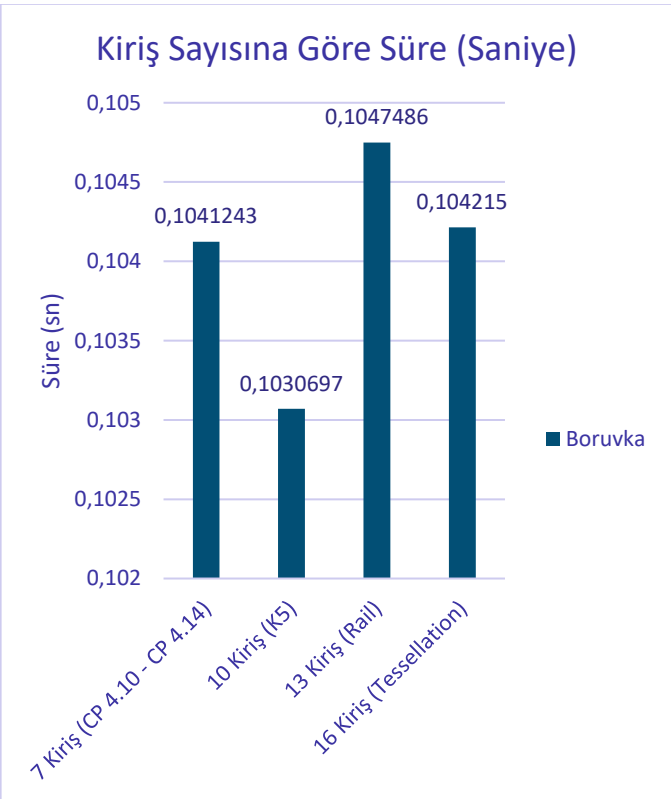
TESSELLATION					
Boruvka			Kruskal		
Time	Allocations	Memory	Time	Allocations	Memory
0,111681	108040	5,439	0,000016	221	0,007492
0,113875	108040	5,439	0,000017	221	0,007492
0,102404	108040	5,439	0,000018	221	0,007492
0,109019	108040	5,439	0,000017	221	0,007492
0,104478	108040	5,439	0,000015	221	0,007492
0,099856	108040	5,439	0,000016	221	0,007492
0,099946	108040	5,439	0,000015	221	0,007492
0,101312	108040	5,439	0,000015	221	0,007492
0,095148	108040	5,439	0,000015	221	0,007492
0,104431	108040	5,439	0,000015	221	0,007492
0,104215	**108040**	**5,439**	**0,0000159**	**221**	**0,007492**

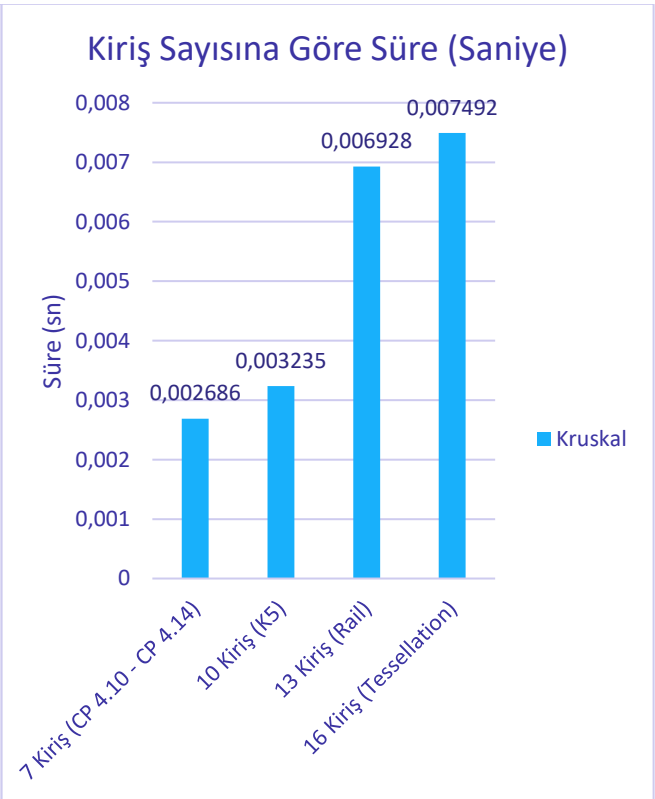





Kiriş Sayısına Göre Atama Miktarları
Atama Sayısı
108100
108050
108000
107950
107900
107850
107800
107900
107910
108010
108040
Boruvka
7 Kiriş (CP 4.10 - CP 4.14)
10 Kiriş (K5)
13 Kiriş (Rail)
16 Kiriş (Tessellation)


Kiriş Sayısına Göre Atama Miktarları
Atama Sayısı
250
200
150
100
50
0
76
94
191
221
Kruskal
7 Kiriş (CP 4.10 - CP 4.14)
10 Kiriş (K5)
13 Kiriş (Rail)
16 Kiriş (Tessellation)


Kiriş Sayısına Göre Süre (Saniye)
Süre (sn)
0,105
0,1045
0,104
0,1035
0,103
0,1025
0,102
0,1041243
0,1030697
0,1047486
0,104215
Boruvka
7 Kiriş (CP 4.10 - CP 4.14)
10 Kiriş (K5)
13 Kiriş (Rail)
16 Kiriş (Tessellation)


Kiriş Sayısına Göre Süre (Saniye)
Süre (sn)
0,008
0,007
0,006
0,005
0,004
0,003
0,002
0,001
0
0,002686
0,003235
0,006928
0,007492
Kruskal
7 Kiriş (CP 4.10 - CP 4.14)
10 Kiriş (K5)
13 Kiriş (Rail)
16 Kiriş (Tessellation)

Sonuçlar ve Çıkarımlar

Karşılaştırılacak değerler olarak kiriş sayıları seçildi. Her iki algoritmada da kiriş listesi üzerinden işlem yaptığımız için algoritmanın temel girdisini kiriş listesinin uzunluğu olarak kabul edip onun üzerinde karşılaştırmalarımızı yapıyoruz.

Algoritmalar arası atama sayıları ve çalışma süresi arasında doğrudan ikili karşılaştırma yaptığımız zaman Kruskal'ın algoritmasının hatırı sayılır şekilde de az atama ve daha kısa süre içinde sonuçlandığını görüyoruz.

Elbette algoritmaların çalıştığı an sistemdeki yükün her zaman kusursuz stabillikte olabileceğini söylemek testlerin yapıldığı kişisel bilgisayar için zor ancak bize yeterince fikir veriyor. Testler bu nedenle 3 defa yenilendi (her örnek başına 30 çalıştırma) ve tutarlı bulunan ölçümler çıkarımlar için kullanıldı.

Kiriş sayısı arttıkça iki algoritmada da atama artışı gözlendi. Artış miktarları da kendi bir önceki örnek ölçümlerine göre baktığımızda epey yaklaşık. Örneğin Boruvka için 13 kirişli *Rail* örneğiyle 16 kirişli *Tessellation* örneğinin atama sayılarına baktığımız zaman aradaki farkın 30 atama olduğunu görüyoruz. Aynı şekilde Kruskal'a baktığımızda da farkımız 30 atama. *Rail* ve *K5*'i kıyasladığımız zaman ise atama farkı 100 iken, Kruskal'da 97.

Sürelere baktığımızda ise iki algoritma için de atama miktarlarındaki gibi düzenli bir artış görebildiğimizi söylemek zor. Kruskal'ın algoritmasında kiriş sayısı arttıkça daha uzun süren bir çözüm süresiyle karşılaşıyoruz. Çözüm süresi Boruvka'nın algoritmasına kıyasla her türlü daha kısa ancak Boruvka'nın algoritması kiriş sayısı arttıkça çalışma süresi olarak da düzenli olarak artmıyor. Aksine ortalama *0,104 saniye.* En kısa çalışma süresiyle ortalama süre arasındaki fark *0,001 saniye* civarlarında. Kruskal'ın algoritması için de ortalama *0,005* saniye görüyoruz. Bu da en kısa çalışma süresiyle arasındaki farkın *0,003 saniye*, en uzun çalışma süresiyle farkın ise *0,002 saniye* olduğunu görüyoruz. Boruvka her ne kadar Kruskal'a kıyasla kullanılan örnekler çerçevesinde daha uzun süreyle çözüm bulsa da, Kruskal'a kıyasla hesaplama süresi olarak daha tutarlı ve kiriş sayısı fark etmeksizin çözüm sürelerinin daha yaklaşık olduğunu bize gösteriyor.

[i] **http://bilgisayarkavramlari.sadievrenseker.com/2007/12/24/asgari-tarama-agaci-en-kisa-orten-agac-minimum-spanning-tree/**
[ii] **https://www.geeksforgeeks.org/total-number-spanning-trees-graph/**
[iii] **https://www.statisticshowto.datasciencecentral.com/boruvkas-algorithm/**
[iv] **https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm**
[v] **https://en.wikipedia.org/wiki/Kruskal%27s_algorithm**
[vi] **https://visualgo.net/en/mst**