

**Gebze Technical University  
Computer Engineering**

**CSE 222 - 2018 Spring**

**HOMEWORK 7 REPORT**

**FERHAT ŞİRİN  
161044080**

Course Assistant: Fatma Nur Esirci

# 1 Q1

## 1.1 Problem Solution Approach

```
public class ListGraph extends AbstractGraph {  
    private List<Edge>[] edges;
```

Graph oluşturmak için ListGraph yapısı kullanılmıştır. ListGraph ile vertexlerin komşuları list arrayinde temsil edilir. Index vertexi belirtirken, listenin içindeki değerler vertexin komşularıdır.

```
public void insert(Edge edge){  
    edges[edge.getSource()].add(edge);  
    System.out.println("New edge created "+edge);  
    if(!isDirected()){  
        edges[edge.getDest()].add(new  
Edge(edge.getDest(),edge.getSource(),edge.getWeight()));  
    }  
}
```

insert methodu ile yeni edge'ler graph yapısına eklenir. Eğer graph undirected yapıda ise aynı edge ters yönden eklenerek iki vertex arası gidiş geliş yolları belirtilir. Directed bir graph için bu yol tek yön olarak kalır.

```
public Edge(int s,int d,double w){  
    source =s;  
    dest =d;  
    weight =w;  
}  
public Edge(int s,int d){  
    this(s,d,1.0);  
}
```

Edge sınıfı ile yeni bir edge oluşturulur. Edge için source ve destination değerleri belirtilir. Eğer grap weighted yapıdaysa weight değeri girilir eğer unweighted ise değer default 1.0 olarak atanır.

```
public static boolean is_acyclic_graph(Graph graph){  
    for(int i=0;i<graph.getNumV();++i) {  
        boolean[] visited =new boolean[graph.getNumV()];  
        visited[i] = true;  
        if(traverseTree(graph,i,i,visited,1)){  
            return false;  
        }  
    }  
    return true;  
}  
private static boolean traverseTree(Graph graph,int vertex,int  
current,boolean[] visited,int count){  
    visited[current] =true;  
    Iterator<Edge> iter =graph.edgeIterator(current);  
    while(iter.hasNext()){  
        int neighbor =iter.next().getDest();  
        if(2 < count && neighbor == vertex){  
            return true;  
        }
```

```

    }
    if(!visited[neighbor]){
        ++count;
        return false ||
        traverseTree(graph,vertex,neighbor,visited,count);
    }
}
return false;
}

```

is\_acyclic\_graph methodu ile graph'ın cycle içerip içermediğine bakılır. Bunun için deep first search algoritmasından yararlanılmıştır. is\_acyclic methodu her bir vertex için traverseTree methodunu çağırır. Bu methodta deep first search algoritması ile graph verilen vertex değerinden itibaren tek tek gezilir. Eğer aynı vertex değerine dönülürse bu durumda graph içinde cycle vardır. Bu durumda false değeri döndürülür. Tüm vertexler için bu duruma bakılır eğer cycle çıkmaz ise true değeri döndürülür.

```

public static boolean is_undirected(Graph graph){
    for(int i=0;i<graph.getNumV();++i){
        Iterator<Edge> iter =graph.edgeIterator(i);
        while(iter.hasNext()){
            if(graph.getEdge(iter.next().getDest(),i) == null){
                return false;
            }
        }
    }
    return true;
}

```

is\_undirected metodu ile graph'ın direct mi undirect mi olduğuna bakılır. Her bir vertexin komşuları gezilir eğer komşu vertex içinde, baktığımız vertexe doğru başka bir edge daha varsa bu durumda graph undirect durumundadır fakat komşu vertexten, baktığımız vertexe edge yok ise grap directed bir graphdır, false değeri döndürülür.

```

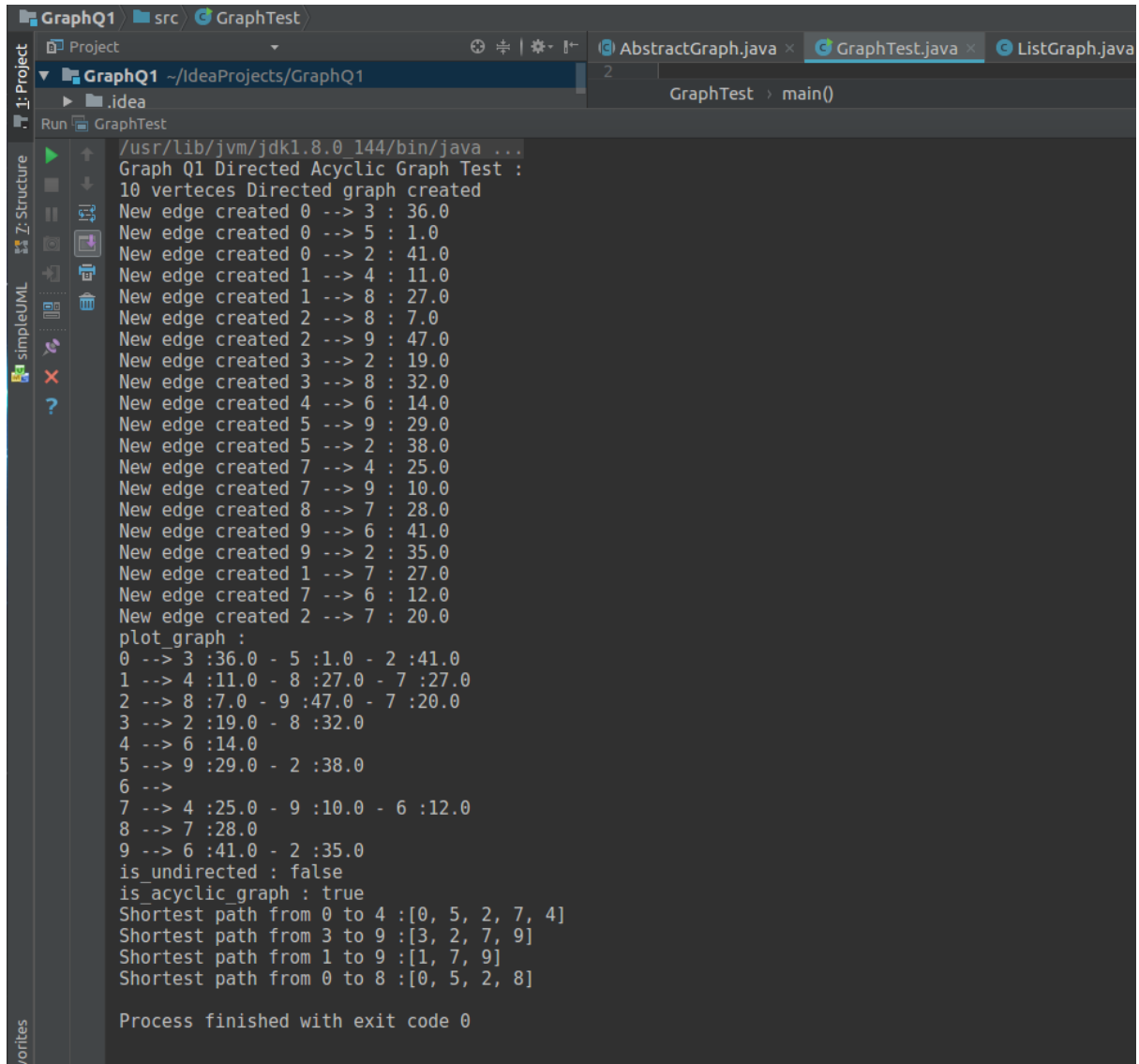
public static Vector<Integer> shortest_path(Graph graph,int
vertex1,int vertex2){
    if (vertex1 < 0 || vertex1 >= graph.getNumV() || vertex2 < 0 ||
vertex2 >= graph.getNumV()) {
        throw new IllegalArgumentException();
    }
    int[] pred =new int[graph.getNumV()];
    double[] dist =new double[graph.getNumV()];
    Vector<Integer> path =new Vector<>();
    if(!is_connected(graph,vertex1,vertex2)){
        return path;
    }
    dijkstrasAlgorithm(graph,vertex1,pred,dist);
    path.add(vertex2);
    while(vertex1 != vertex2){
        path.add(0,pred[vertex2]);
        vertex2 =pred[vertex2];
    }
}

```

```
    return path;
}
```

shortest\_path methodu ile graph içinde verilen 2 vertex arasındaki pathlerden en kısa olanı vector içinde döndürür. Bu method en kısa yolu bulmak için dijkstrasAlgoritmasını kullanır. Bu algoritma bir S kümesi birde V-S kümesi tutar. Başlangıç olarak S kümesinde başlangıç vertexi bulunur V-S kümesinde ise diğer vertexler. Uzaklık değerleri ve parent değerleri içinde iki ayrı array tutulur. Bu arrayler ilk uzaklık değerleri ile doldurulur. Daha sonra her bir vertexin daha kısa ulaşılan değeri uzaklık arrayine yazılır ve V-S kümesinden çıkarılır. Küme boş kalana kadar bu işlem devam eder. Elde edilen parent arrayden değerler alınarak vectore eklenilir ve en kısa path döndürülür.

## 1.2 Test Cases



```
GraphQ1 src GraphTest
Project
  GraphQ1 -~/IdeaProjects/GraphQ1
    .idea
Run GraphTest
/usr/lib/jvm/jdk1.8.0_144/bin/java ...
Graph Q1 Directed Acyclic Graph Test :
10 vertices Directed graph created
New edge created 0 --> 3 : 36.0
New edge created 0 --> 5 : 1.0
New edge created 0 --> 2 : 41.0
New edge created 1 --> 4 : 11.0
New edge created 1 --> 8 : 27.0
New edge created 2 --> 8 : 7.0
New edge created 2 --> 9 : 47.0
New edge created 3 --> 2 : 19.0
New edge created 3 --> 8 : 32.0
New edge created 4 --> 6 : 14.0
New edge created 5 --> 9 : 29.0
New edge created 5 --> 2 : 38.0
New edge created 7 --> 4 : 25.0
New edge created 7 --> 9 : 10.0
New edge created 8 --> 7 : 28.0
New edge created 9 --> 6 : 41.0
New edge created 9 --> 2 : 35.0
New edge created 1 --> 7 : 27.0
New edge created 7 --> 6 : 12.0
New edge created 2 --> 7 : 20.0
plot_graph :
0 --> 3 :36.0 - 5 :1.0 - 2 :41.0
1 --> 4 :11.0 - 8 :27.0 - 7 :27.0
2 --> 8 :7.0 - 9 :47.0 - 7 :20.0
3 --> 2 :19.0 - 8 :32.0
4 --> 6 :14.0
5 --> 9 :29.0 - 2 :38.0
6 -->
7 --> 4 :25.0 - 9 :10.0 - 6 :12.0
8 --> 7 :28.0
9 --> 6 :41.0 - 2 :35.0
is_undirected : false
is_acyclic_graph : true
Shortest path from 0 to 4 :[0, 5, 2, 7, 4]
Shortest path from 3 to 9 :[3, 2, 7, 9]
Shortest path from 1 to 9 :[1, 7, 9]
Shortest path from 0 to 8 :[0, 5, 2, 8]
Process finished with exit code 0
```

plot\_graph ile graph'ın vertexlerinin tüm komşuları ve aralarındaki weight değeri gösterilmiştir. is\_undirected methodunun false dönmesi ile graph'ın direct bir graph olduğunu ve is\_acyclic\_graph methodunun true dönmesi ile graph'ın bir acyclic bir graph olduğu söylenebilir. Graphın shortest\_path methodu 4 ayrı durumda test edilmiştir.

## 2 Q2

### 2.1 Problem Solution Approach

Graph oluşturmak için part 1 de olduğu gibi ListGraph sınıfı kullanılmıştır ve aynı methodlardan yararlanılmıştır. is\_undirected ve is\_acyclic\_graph methodları part 1 ile aynıdır.

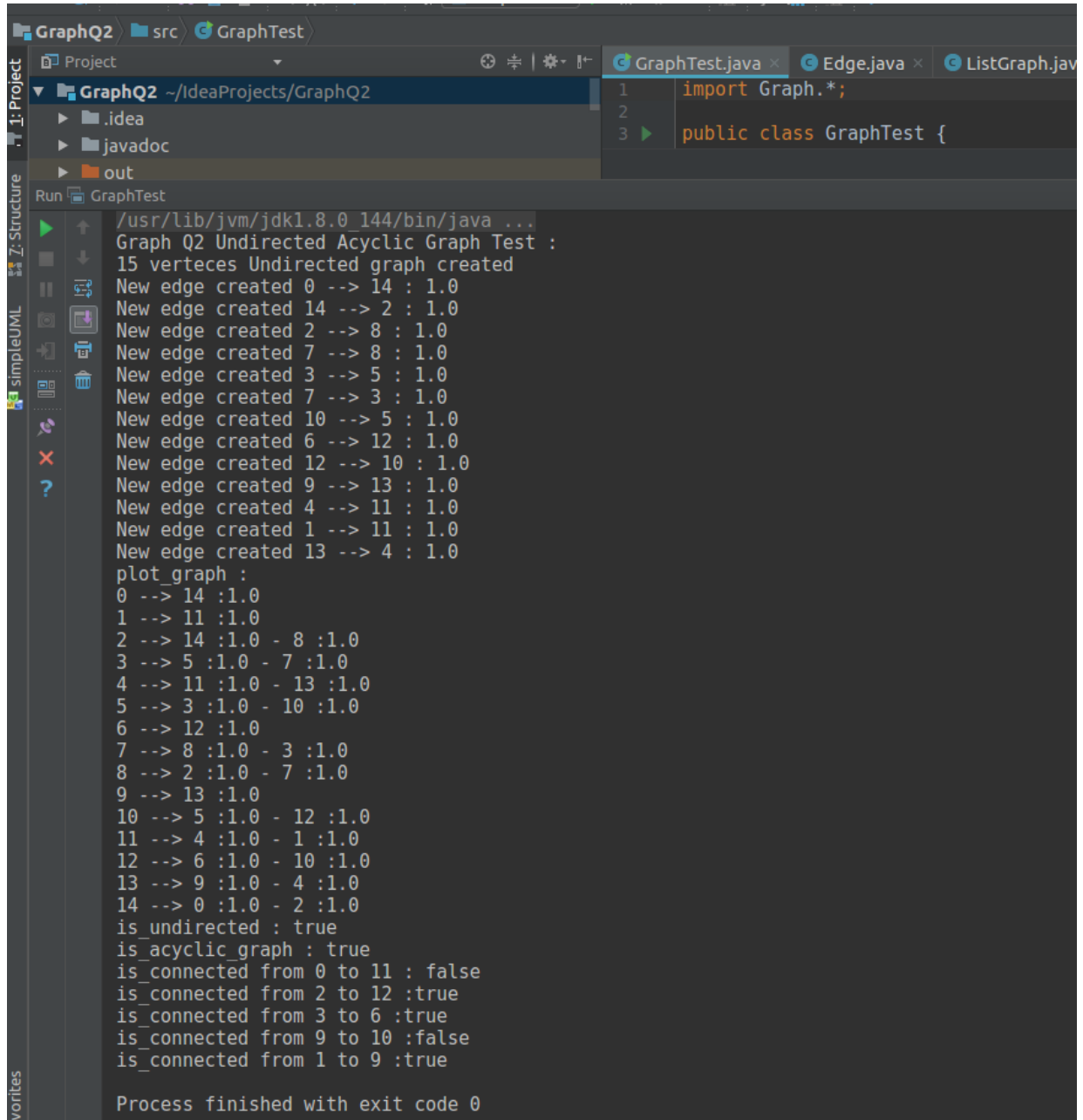
Bu methodlara ek olarak is\_connected methodu bu bölümde kullanılmıştır.

```
public static boolean is_connected(Graph graph,int vertex1,int vertex2) {
    if (vertex1 < 0 || vertex1 >= graph.getNumV() || vertex2 < 0 || vertex2 >= graph.getNumV()) {
        throw new IllegalArgumentException();
    }
    if(vertex1 == vertex2){
        return true;
    }
    Queue<Integer> queue =new LinkedList<Integer>();
    boolean[] identified =new boolean[graph.getNumV()];
    identified[vertex1] =true;
    queue.offer(vertex1);
    while(!queue.isEmpty()){
        int current =queue.remove();
        Iterator iter =graph.edgeIterator(current);
        while(iter.hasNext()){
            Edge edge = (Edge) iter.next();
            int neighbor =edge.getDest();
            if(neighbor == vertex2){
                return true;
            }
            if(!identified[neighbor]){
                identified[neighbor] =true;
                queue.offer(neighbor);
            }
        }
    }
    return false;
}
```

is\_connected methodu ile verilen graph'ın iki vertexi arasında bir path olup olmadığına bakılır.

Öncelikle istisna durumlara bakılarak vertexlerin graphta olup olmadığı ve birbirine eşit olup olmadığına bakılır. Vertexler kendilerine bağlı olduğu için eşitse true değeri döndürülür. İki vertex arası pathi bulmak için breadth first search algoritmasından yararlanılmıştır. Başlangıç vertex'inden yola çıkarak tüm komşular queue içine alınır. Daha sonra bu komşulara bakılarak aynı işlem tekrar eder eğer bu sırada aranan vertex değerine ulaşırsa true değeri döndürülür fakat tüm graph yapısı gezildiği halde değer bulunamazsa false değeri döndürülür.

## 2.2 Test Cases



```
GraphQ2 src GraphTest
Project
  GraphQ2 ~/IdeaProjects/GraphQ2
    .idea
    javadoc
    out
Run GraphTest
/usr/lib/jvm/jdk1.8.0_144/bin/java ...
Graph Q2 Undirected Acyclic Graph Test :
15 vertexes Undirected graph created
New edge created 0 --> 14 : 1.0
New edge created 14 --> 2 : 1.0
New edge created 2 --> 8 : 1.0
New edge created 7 --> 8 : 1.0
New edge created 3 --> 5 : 1.0
New edge created 7 --> 3 : 1.0
New edge created 10 --> 5 : 1.0
New edge created 6 --> 12 : 1.0
New edge created 12 --> 10 : 1.0
New edge created 9 --> 13 : 1.0
New edge created 4 --> 11 : 1.0
New edge created 1 --> 11 : 1.0
New edge created 13 --> 4 : 1.0
plot_graph :
0 --> 14 :1.0
1 --> 11 :1.0
2 --> 14 :1.0 - 8 :1.0
3 --> 5 :1.0 - 7 :1.0
4 --> 11 :1.0 - 13 :1.0
5 --> 3 :1.0 - 10 :1.0
6 --> 12 :1.0
7 --> 8 :1.0 - 3 :1.0
8 --> 2 :1.0 - 7 :1.0
9 --> 13 :1.0
10 --> 5 :1.0 - 12 :1.0
11 --> 4 :1.0 - 1 :1.0
12 --> 6 :1.0 - 10 :1.0
13 --> 9 :1.0 - 4 :1.0
14 --> 0 :1.0 - 2 :1.0
is_undirected : true
is_acyclic_graph : true
is_connected from 0 to 11 : false
is_connected from 2 to 12 :true
is_connected from 3 to 6 :true
is_connected from 9 to 10 :false
is_connected from 1 to 9 :true
Process finished with exit code 0
```

plot\_graph ile graph'ın vertexlerinin tüm komşuları ve aralarındaki weight değeri gösterilmiştir.

Graph unweighted graph olduğu için weight değeri hepsi için 1.0 olarak seçildi.

is\_undirected methodun true dönmesi ile graph'ın undirect bir graph olduğunu ve is\_acyclic\_graph methodunun true dönmesi ile graph'ın bir acyclic bir graph olduğu söylenebilir. Graphın is\_connected methodu 4 ayrı durumda test edilmiştir.

## 3 Q3

### 3.1 Problem Solution Approach

Graph oluşturmak için part 1 de olduğu gibi ListGraph sınıfı kullanılmıştır ve aynı methodlardan yararlanılmıştır. is\_undirected ve is\_acyclic\_graph methodları part 1 ile aynıdır.

Bu methodlara ek olarak breadth first search ve depth first search methodları ödevde uygun şekilde yazılmıştır. Search sonrasında çıkan ağaçları yazdırmak için TreeNode sınıfından yararlanılmıştır.

```
public static void DepthFirstSearch(Graph graph){
    int[] parent =new int[graph.getNumV()];
    boolean[] visited =new boolean[graph.getNumV()];
    int[] discoveryOrder =new int[graph.getNumV()];
    int[] finishOrder =new int[graph.getNumV()];
    for(int i=0;i<graph.getNumV();++i){
        parent[i] =-1;
    }
    for(int i=0;i<graph.getNumV();++i){
        if(!visited[i]){
            depthFirstSearch(graph,i,parent,discoveryOrder,finishOrder,visited);
        }
    }
    TreeNode tree =new TreeNode(0,createTree(graph,parent,0));
    System.out.println("Depth First Search :");
    System.out.printf("Discovery Order : ");
    for(int i=0;i<graph.getNumV();++i){
        System.out.printf("%d ",discoveryOrder[i]);
    }
    System.out.println();
    System.out.printf("Finish Order      : ");
    for(int i=0;i<graph.getNumV();++i){
        System.out.printf("%d ",finishOrder[i]);
    }
    System.out.println();
    System.out.printf("Parent Order      : ");
    for(int i=0;i<graph.getNumV();++i){
        System.out.printf("%d ",parent[i]);
    }
    System.out.println();
    System.out.println("Depth First Search Tree :");
    tree.print();
}
```

Depth first search methodu ile gönderilen graph'ın search işlemi yapılır ve elde edilen discovery order, finish order ve parent gibi bilgiler yazdırılmıştır ve tree yapısı çizilmiştir. Depth first search ile verilen başlangıç vertexinin ilk komşusuna gidiler ve oradan hemen başka komşu vertexlere geçilir, komşu vertex kalmadığı durumda geri dönülerek diğer komşu vertexler varsa onlar gezilir ve graphın tüm vertexleri gezilmiş olur.

```

public static void BreadthFirstSearch(Graph graph){
    int[] parent =breadthFirstSearch(graph,0);
    TreeNode tree =new TreeNode(0,createTree(graph,parent,0));
    System.out.printf("Breadth First Search Order : ");
    for(int i=0;i<graph.getNumV();++i){
        System.out.printf("%d ",parent[i]);
    }
    System.out.println();
    System.out.println("Breadth First Search Tree :");
    tree.print();
}

```

Breadth first search methodu ile gönderilen graph'ın search işlemi yapılır ve elde edilen parent bilgisi yazdırılır ve tree yapısı çizilir. Breadth first search başlangıç vertexinin tüm komşularını gezer daha sonra gezecek komşusu kalmadığında, queue yapısına attığı daha önce gezdiği vertexlerin komşularına gider ve graph'ın tüm vertexlerini gezene kadar devam eder.

## 3.2 Test Cases

```

new edge created 1 1.0 9 1.0
plot_graph :
0 --> 4 :1.0 - 9 :1.0 - 8 :1.0 - 1 :1.0 - 7 :1.0
1 --> 5 :1.0 - 3 :1.0 - 8 :1.0 - 0 :1.0 - 9 :1.0
2 --> 7 :1.0 - 8 :1.0 - 6 :1.0 - 3 :1.0
3 --> 1 :1.0 - 8 :1.0 - 5 :1.0 - 2 :1.0 - 6 :1.0
4 --> 0 :1.0 - 7 :1.0 - 5 :1.0
5 --> 4 :1.0 - 1 :1.0 - 3 :1.0 - 7 :1.0
6 --> 2 :1.0 - 9 :1.0 - 3 :1.0
7 --> 4 :1.0 - 2 :1.0 - 5 :1.0 - 0 :1.0
8 --> 3 :1.0 - 2 :1.0 - 1 :1.0 - 0 :1.0
9 --> 0 :1.0 - 6 :1.0 - 1 :1.0
is_undirected : true
is_acyclic_graph : false
Depth First Search :
Discovery Order : 0 4 7 2 8 3 1 5 9 6
Finish Order : 5 6 9 1 3 8 2 7 4 0
Parent Order : -1 3 7 8 0 1 9 4 2 1
Depth First Search Tree :
0
├── 4
│   ├── 7
│   │   ├── 2
│   │   │   ├── 8
│   │   │   └── 3
│   │       ├── 1
│   │       └── 5
│           ├── 9
│           └── 6
Breadth First Search Order : -1 0 8 8 0 4 9 0 0 0
Breadth First Search Tree :
0
├── 1
├── 4
├── 5
├── 7
├── 8
│   ├── 2
│   └── 3
├── 9
└── 6
Process finished with exit code 0

```

plot\_graph ile graph'ın vertexlerinin tüm komşuları ve aralarındaki weight değeri gösterilmiştir.

Graph unweighted graph olduğu için weight değeri hepsi için 1.0 olarak seçildi.

is\_undirected methodu true döndürdüğü için graph'ın undirect bir graph olduğunu ve

is\_acyclic\_graph false döndürdüğü için graph'ın bir cyclic bir graph olduğu söylenebilir. Graph

breadth first search ve depth first search methodlarına gönderilerek elde edilen bilgiler yazdırılmış ve

agaç yapıları çizdirilmiştir.

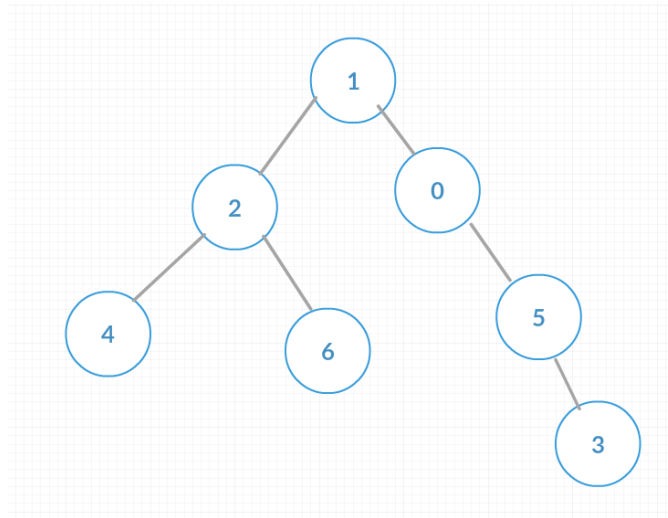


## 4 Q4

### Breadth First Search and Depth First Search

Her iki algoritmada graph ve graph'ın özel bir türü olan treelerde search ve traverse işlemlerinde kullanılır. Breadth first searchte seçilen ilk node tan tüm komşulara yayılarak devam edilirken depth first searchte ilk node sonra hemen başka bir komşu node bulunarak arama yapılır. Bu iki algoritma kullanım alanları genellikle farklılık gösterir. Deep first search ile iki vertex arasında path rahatlıkla bulunabilirken breadth first search genişleyerek traverse yaptığı için en kısa path bulmada daha kolaylık sağlar. Breadth first search ile bir graphın bipartite (iki parçalı) olup olmadığı test edilebilir. Depth first search ile labirent problemlerini daha hızlı çözebilir ve directed acyclic graphlar için topological sorting işleminde kullanılır. Breadth first search yayılarak ilerlediği için büyük graphlarda daha fazla yer tutma ihtiyacı duyarken depth first search graphta daha hızlı bir ilerleme gerçekleştirir. Depth first search bir path üzerinde sonuna kadar ilerlediği için aranan cevap yakında olsa bile cevabı bulmak breadth first searche göre daha fazla zaman alır.

#### Breadth First Search Tree



#### Depth First Search Tree

