Gebze Technical University
Computer Engineering

CSE 222 - 2018 Spring

HOMEWORK 7 REPORT

GULZADA IISAEVA
131044085

Course Assistant: Fatma Nur Esirci

1 Q1 && Q2 && Q3

Bu bölümler için kitabın source kodunu kullanarak GTUGraph sınıfı yazıldı.
Bu sınıfın objesini oluştururken directed ya da undirected olduğunu belirtince ona göre graph oluşturuyor.

Bu graph sınıfında edge leri tutmak için bir List arrayi kullanıldı. Bunun içinde source ve destinationlar vardır. Eğer undirected ise weight için random sayılar üretilir, değilse weight yoktur.

Yazılan metodlar:

plot_graph() :

```
directed ise     4 weight:26.0  ->  10 weight:26.0
                 5 weight:22.0  ->  4 weight:22.0  ->  10 weight:22.0
                 6 weight:56.0  ->  5 weight:56.0
undirected ise 4  ->  3  ->  10  ->  5
                 5  ->  2  ->  3  ->  4  ->  10  ->  6  ->  7
                 6  ->  2  ->  5  ->  7  ->  9  ->  10
```

şeklinde ekrana basar

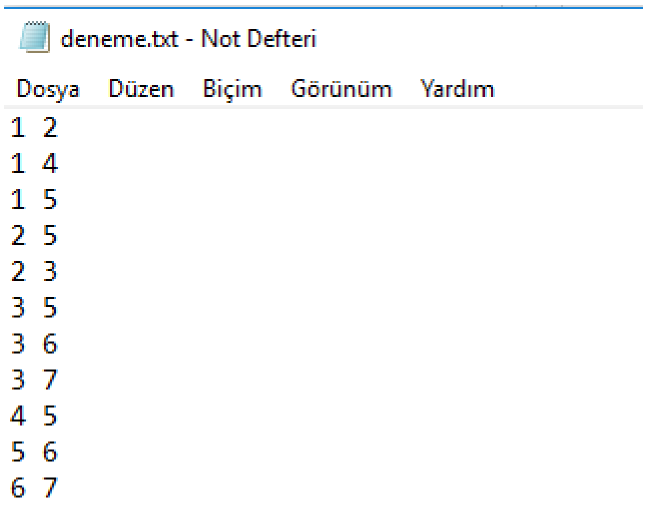
is_directed() : directed ise true, değilse false döndürür.
is_connected(s,d): s (source) – d(destination) arası bir yol varsa true, değilse false döndürür
loadEdgesFromFile()- dosyadan okuyup graph oluşturur.

Test Cases

Test için vertexleri bir dosyaya yazarak loadEdgesFromFile() metodu ile okudum ve graph oluşturdum.

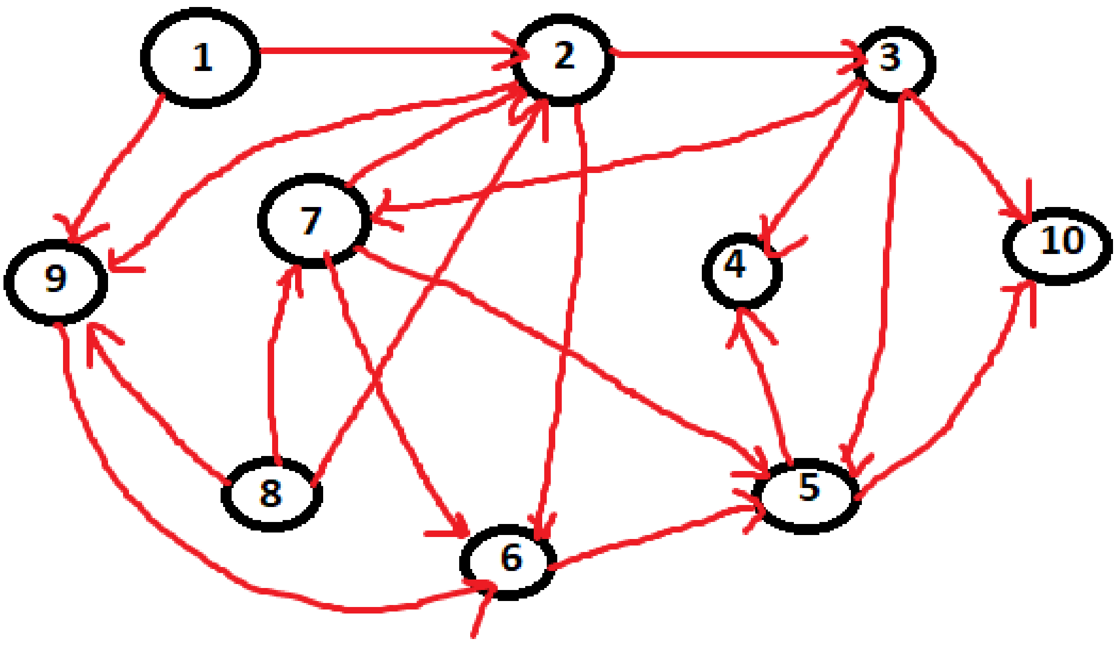
Q1 için directed olarak oluşturdum . Bu da random weight üretecektir
Q2 için undirected ve weight yoktur.

Kitabın Depth First Search ile Breadth First Search kodlari test edildi

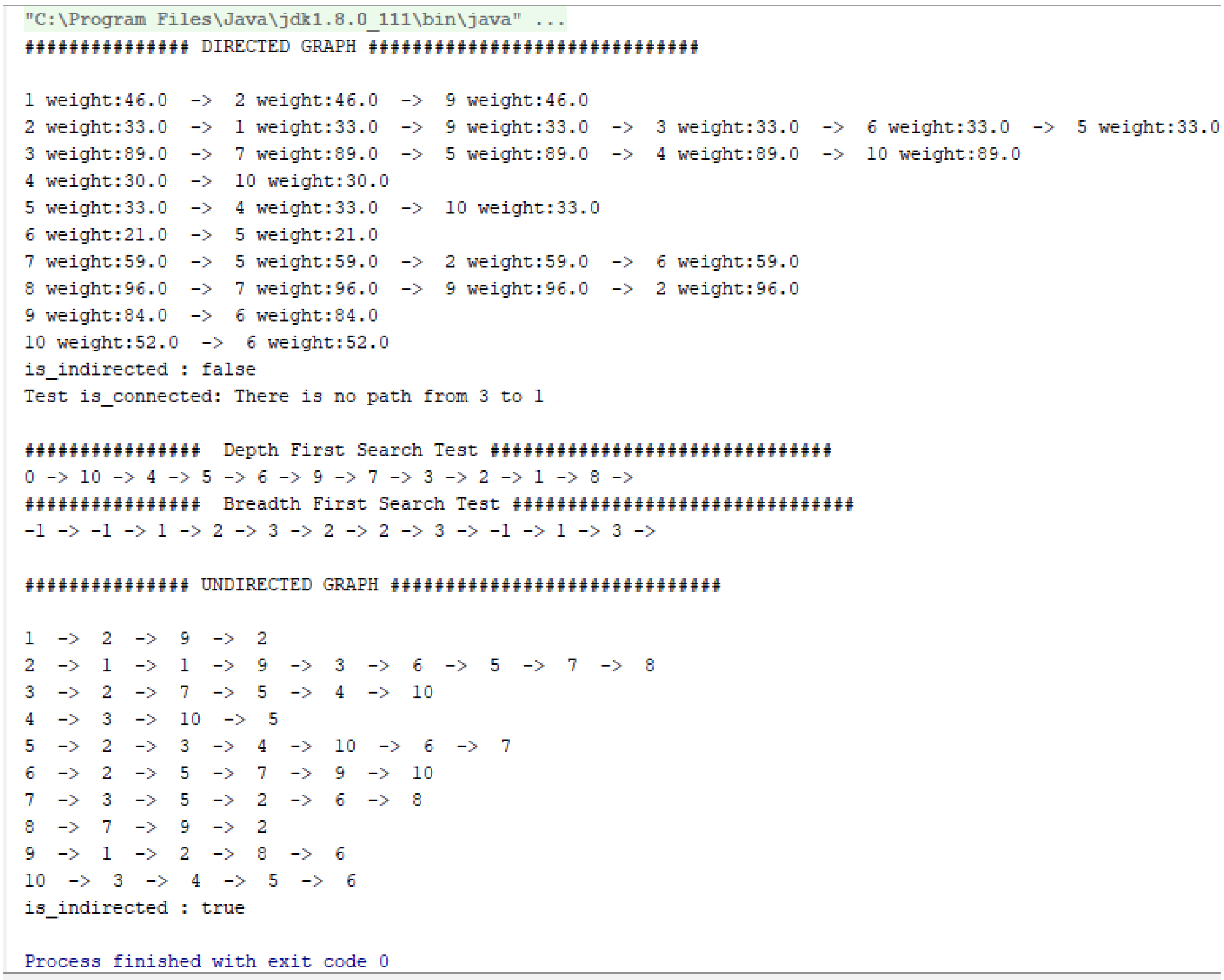


```
1 2
1 4
1 5
2 5
2 3
3 5
3 6
3 7
4 5
5 6
6 7
```

Şekil 1: Vertexlerin dosyaya yazılması



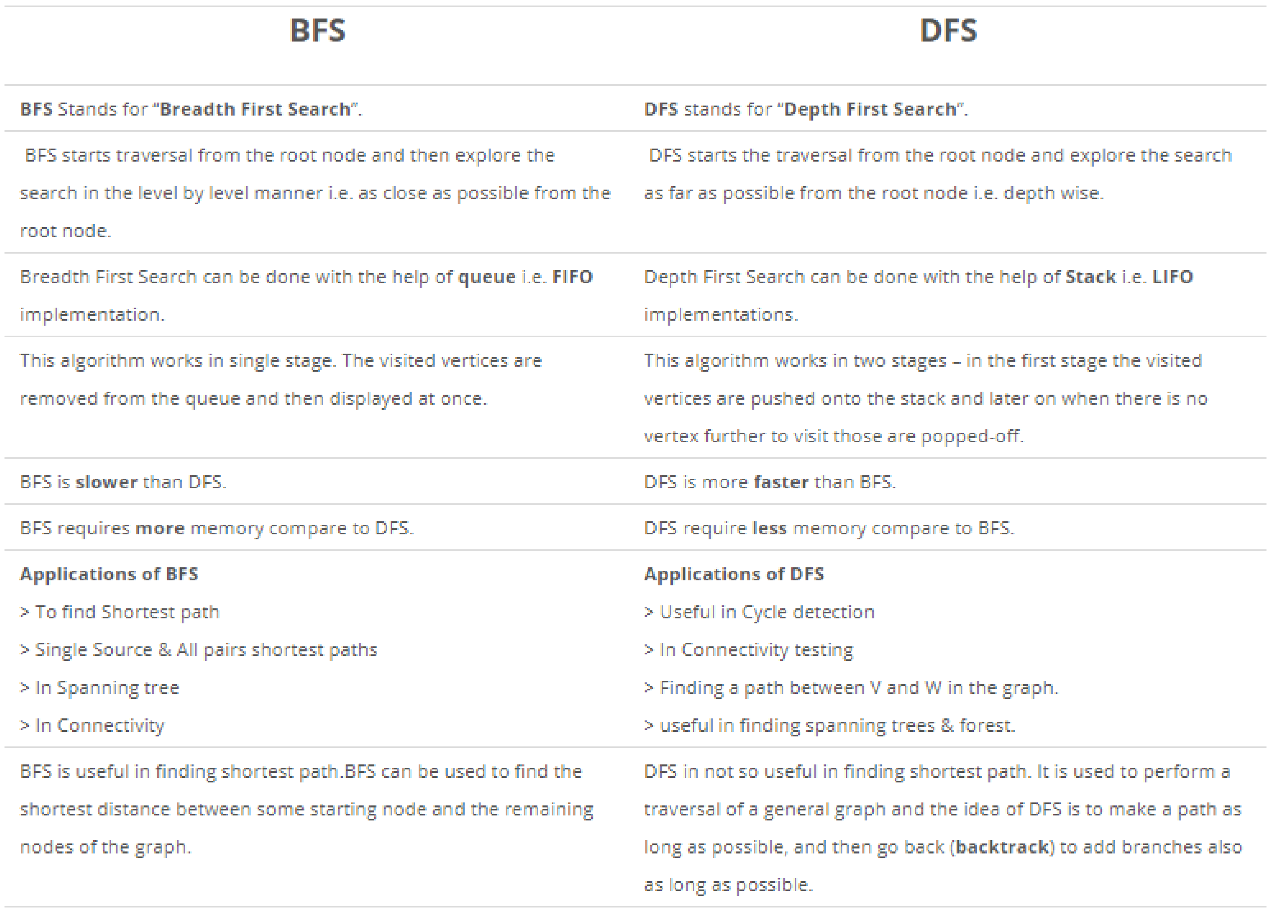
Şekil 2: Test için oluşturulan graph örneği



```
"C:\Program Files\Java\jdk1.8.0_111\bin\java" ...
############### DIRECTED GRAPH ##############################

1 weight:46.0  ->  2 weight:46.0  ->  9 weight:46.0
2 weight:33.0  ->  1 weight:33.0  ->  9 weight:33.0  ->  3 weight:33.0  ->  6 weight:33.0  ->  5 weight:33.0
3 weight:89.0  ->  7 weight:89.0  ->  5 weight:89.0  ->  4 weight:89.0  ->  10 weight:89.0
4 weight:30.0  ->  10 weight:30.0
5 weight:33.0  ->  4 weight:33.0  ->  10 weight:33.0
6 weight:21.0  ->  5 weight:21.0
7 weight:59.0  ->  5 weight:59.0  ->  2 weight:59.0  ->  6 weight:59.0
8 weight:96.0  ->  7 weight:96.0  ->  9 weight:96.0  ->  2 weight:96.0
9 weight:84.0  ->  6 weight:84.0
10 weight:52.0  ->  6 weight:52.0
is_indirected : false
Test is_connected: There is no path from 3 to 1

################  Depth First Search Test ###############################
0 -> 10 -> 4 -> 5 -> 6 -> 9 -> 7 -> 3 -> 2 -> 1 -> 8 ->
################  Breadth First Search Test ###############################
-1 -> -1 -> 1 -> 2 -> 3 -> 2 -> 2 -> 3 -> -1 -> 1 -> 3 ->

############### UNDIRECTED GRAPH ##############################

1  ->  2  ->  9  ->  2
2  ->  1  ->  1  ->  9  ->  3  ->  6  ->  5  ->  7  ->  8
3  ->  2  ->  7  ->  5  ->  4  ->  10
4  ->  3  ->  10  ->  5
5  ->  2  ->  3  ->  4  ->  10  ->  6  ->  7
6  ->  2  ->  5  ->  7  ->  9  ->  10
7  ->  3  ->  5  ->  2  ->  6  ->  8
8  ->  7  ->  9  ->  2
9  ->  1  ->  2  ->  8  ->  6
10  ->  3  ->  4  ->  5  ->  6
is_indirected : true

Process finished with exit code 0
```

Şekil 3: Testin ekran çıktısı

2 Q4



BFS	DFS
BFS Stands for “**Breadth First Search**”.	**DFS** stands for “**Depth First Search**”.
BFS starts traversal from the root node and then explore the search in the level by level manner i.e. as close as possible from the root node.	DFS starts the traversal from the root node and explore the search as far as possible from the root node i.e. depth wise.
Breadth First Search can be done with the help of **queue** i.e. **FIFO** implementation.	Depth First Search can be done with the help of **Stack** i.e. **LIFO** implementations.
This algorithm works in single stage. The visited vertices are removed from the queue and then displayed at once.	This algorithm works in two stages – in the first stage the visited vertices are pushed onto the stack and later on when there is no vertex further to visit those are popped-off.
BFS is **slower** than DFS.	DFS is more **faster** than BFS.
BFS requires **more** memory compare to DFS.	DFS require **less** memory compare to BFS.
Applications of BFS > To find Shortest path > Single Source & All pairs shortest paths > In Spanning tree > In Connectivity	**Applications of DFS** > Useful in Cycle detection > In Connectivity testing > Finding a path between V and W in the graph. > useful in finding spanning trees & forest.
BFS is useful in finding shortest path.BFS can be used to find the shortest distance between some starting node and the remaining nodes of the graph.	DFS in not so useful in finding shortest path. It is used to perform a traversal of a general graph and the idea of DFS is to make a path as long as possible, and then go back (**backtrack**) to add branches also as long as possible.