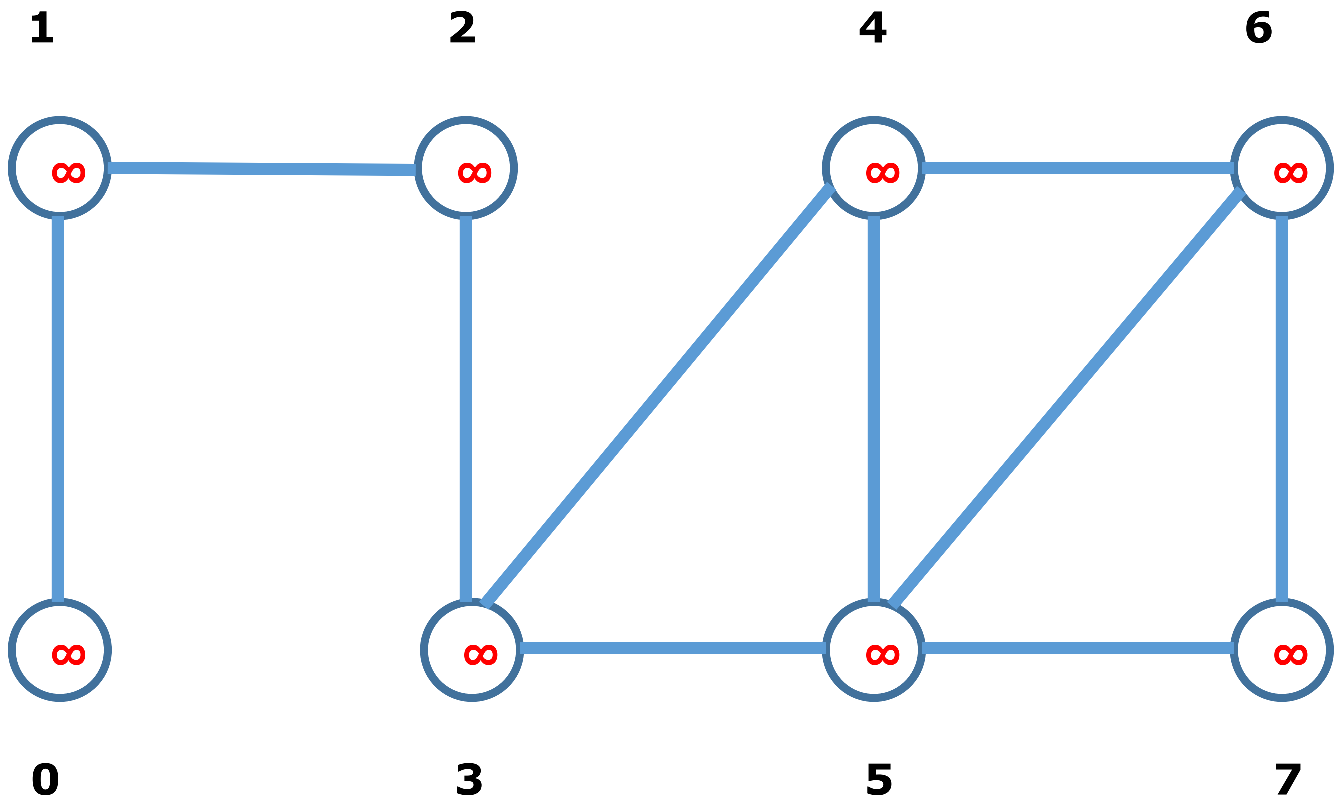
BFS(G , s) breadth-first search

G = (V, E) – graph , s – start (source vertex)

An adjacency-list representation of G

vertex u	vertex v
0	1
1	0 2
2	1 3
3	2 4 5
4	3 5 6
5	3 4 6 7
6	4 5 7
7	5 6

pred – predecessor or parent
dist – distance



pred: [-1, -1, -1, -1, -1, -1, -1, -1]

dist: [-1, -1, -1, -1, -1, -1, -1, -1] ∞ ~ -1

color: [1, 1, 1, 1, 1, 1, 1, 1]

queue: []

pseudocode of BFS on a sample graph.

```
for each vertex u ϵ G:V - (s)
    u.color = WHITE
    u.dist  = 1
    u.pred  = NIL
```

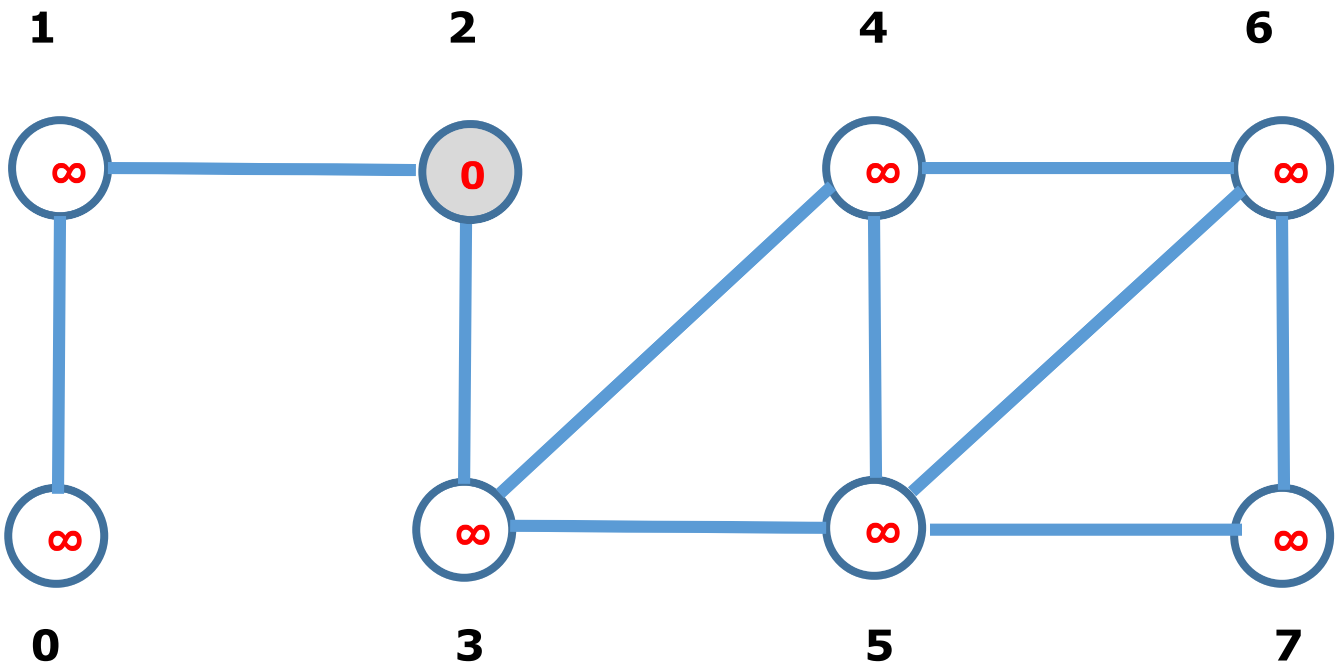
u.color – צבע

u.dist – מרחק מ-s ל-u

u.pred – הורה ל-u

```
s:color  = GRAY
s:dist   = 0
s:pred   = NIL
Q = []
ENQUEUE.Q(s)
```

```
while Q ≠ ∅
u = DEQUEUE(Q)
for each v ϵ G:Adj(u)
    if  v.color == WHITE
        v.color = GRAY
        v.dist = u.dist + 1
        v.pred = u
        ENQUEUE(Q, v)
u:color = BLACK
```



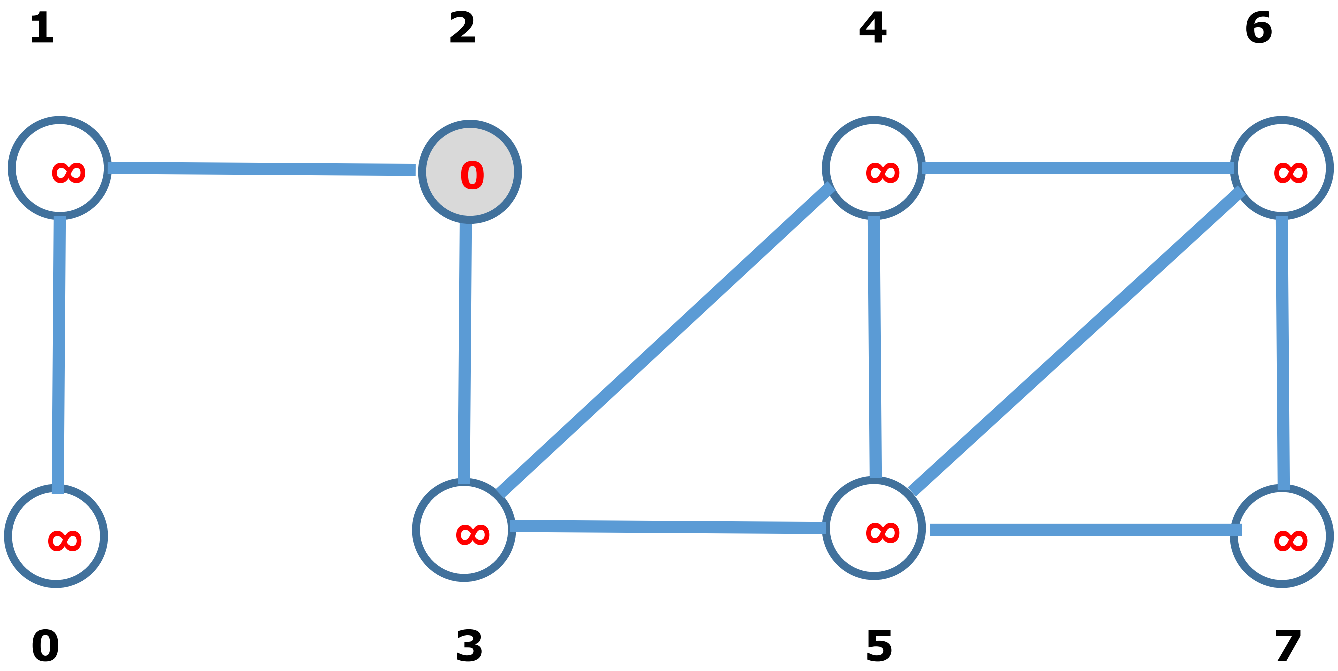
start = 2

pred: [-1, -1, -1, -1, -1, -1, -1, -1]

dist: [-1, -1, 0, -1, -1, -1, -1, -1]

color: [1, 1, 2, 1, 1, 1, 1, 1]

Q = [2]




```
while Q ≠ ∅
u = DEQUEUE(Q)
for each v ϵ G:Adj(u)
    if  v.color == WHITE
        v.color = GRAY
        v.dist = u.dist + 1
        v.pred = u
        ENQUEUE(Q, v)
u:color = BLACK
```

u = DEQUEUE(Q) = **2** graph[2] : [1, 3]

q : []

u = 2 v = 1

dist[1] = 1 pred[1] = 2 color[1] = 2 **q : [1]**

pred: [-1, **2**, -1, -1, -1, -1, -1, -1]
dist: [-1, **1**, **0**, -1, -1, -1, -1, -1]
color: [1, **2**, **2**, 1, 1, 1, 1, 1]

u = 2 v = 3

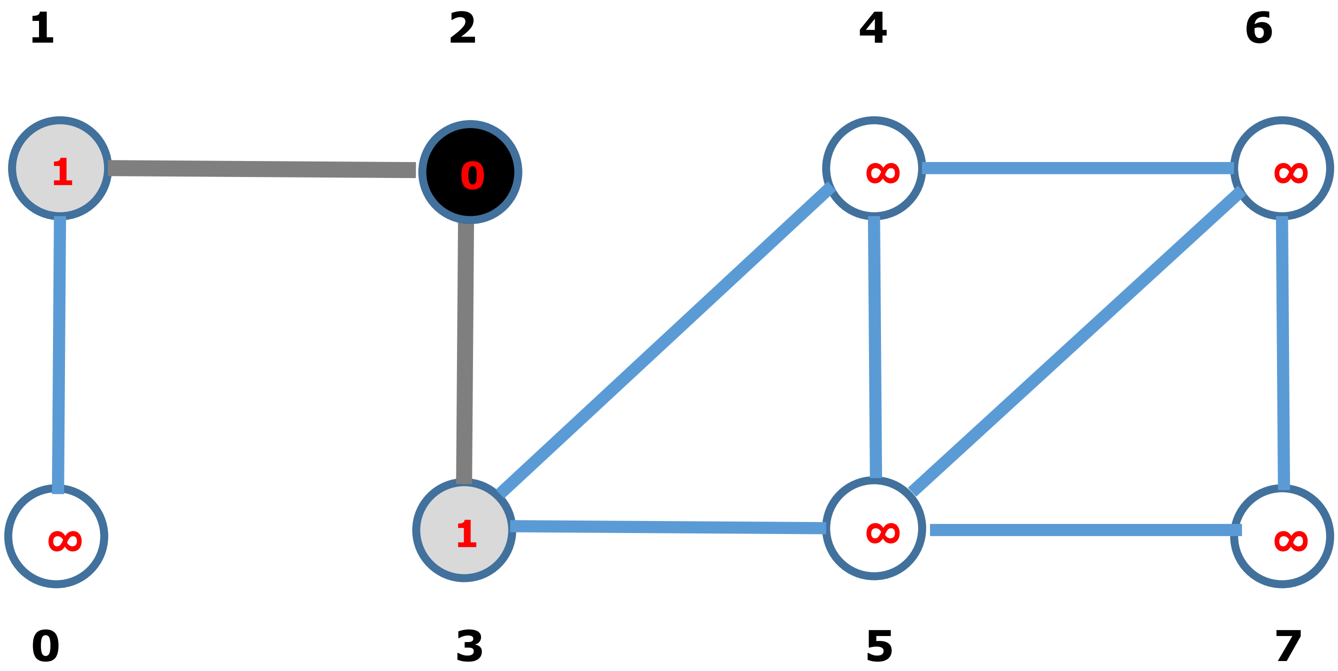
dist[3] = 1 pred[3] = 2 color[3] = 2 **q : [3, 1]**

pred: [-1, **2**, -1, **2**, -1, -1, -1, -1]
dist: [-1, **1**, **0**, **1**, -1, -1, -1, -1]
color: [1, **2**, **2**, **2**, 1, 1, 1, 1]

color[2] = 3

color: [1, 2, 3, 2, 1, 1, 1, 1]

q : [3, 1]




```
while Q ≠ ∅
u = DEQUEUE(Q)
for each v ∈ G:Adj(u)
        if  v.color == WHITE
                v.color = GRAY
                v.dist = u.dist + 1
                v.pred = u
                ENQUEUE(Q, v)
u:color = BLACK
```

u = DEQUEUE(Q) = **1**   graph[1] : [0, 2]

q : [3]

u = 1   **v = 0**

dist[0] = 2   pred[0] = 1   color[0] = 2

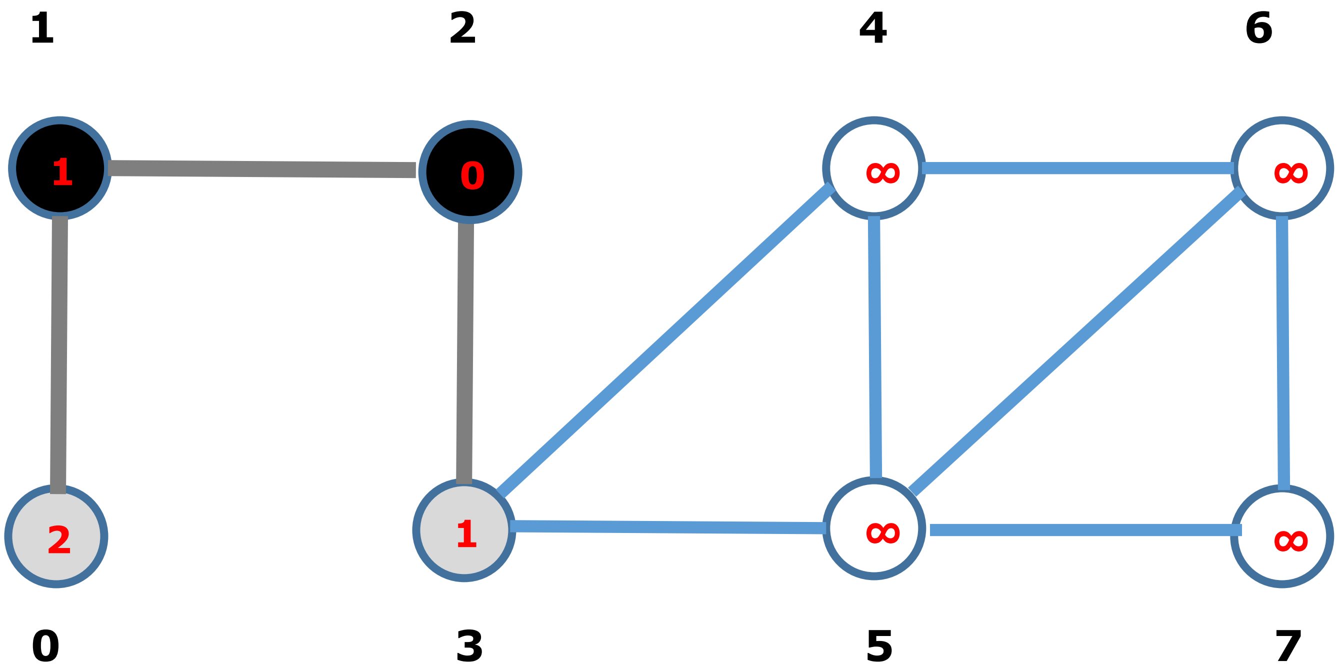
q : [0, 3]

pred: [**1**, **2**, -1, **2**, -1, -1, -1, -1]
dist: [**2**, **1**, **0**, **1**, -1, -1, -1, -1]
color: [**2**, **2**, **3**, **2**, 1, 1, 1, 1]

u = 1   **v = 2**

color: [2, 3, 3, 2, 1, 1, 1, 1]

q : [0, 3]




```
while Q ≠ ∅
u = DEQUEUE(Q)
for each v ϵ G:Adj(u)
        if  v.color == WHITE
                v.color = GRAY
                v.dist = u.dist + 1
                v.pred = u
                ENQUEUE(Q, v)
u:color = BLACK
```

u = DEQUEUE(Q) = **3**   graph[3] : [2, 4, 5]

q : [0]

u = 3   v = 2
u = 3   v = 4

dist[4] = 2   pred[4] = 3   color[4] = 2   **q : [4, 0]**

pred: [**1**, **2**, -1, **2**, **3**, -1, -1, -1]
dist: [**2**, **1**, **0**, **1**, **2**, -1, -1, -1]
color: [**2**, **3**, **3**, **2**, **2**, 1, 1, 1]

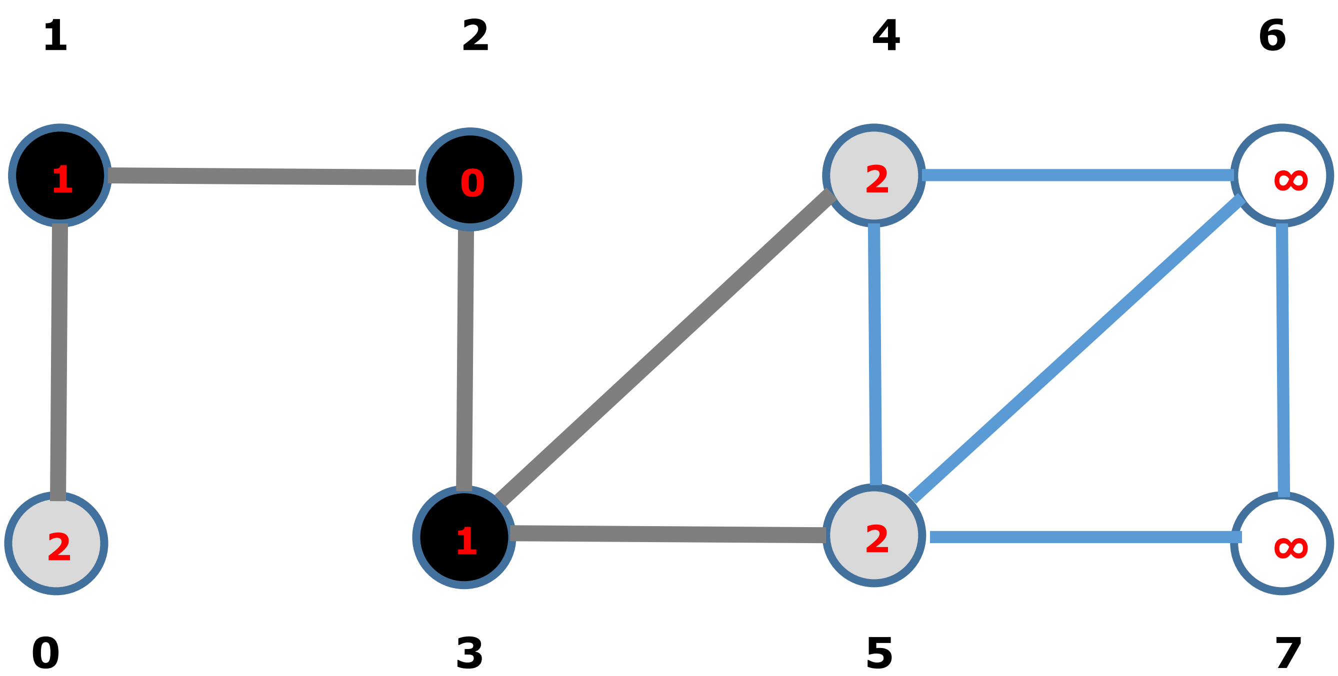
u = 3   v = 5

dist[5] = 2   pred[5] = 3   color[5] = 2   **q : [5, 4, 0]**

pred: [**1**, **2**, -1, **2**, **3**, **3**, -1, -1]
dist: [**2**, **1**, **0**, **1**, **2**, **2**, -1, -1]
color: [**2**, **3**, **3**, **2**, **2**, **2**, 1, 1]

color: [2, 3, 3, 3, 2, 2, 1, 1]

q : [5, 4, 0]



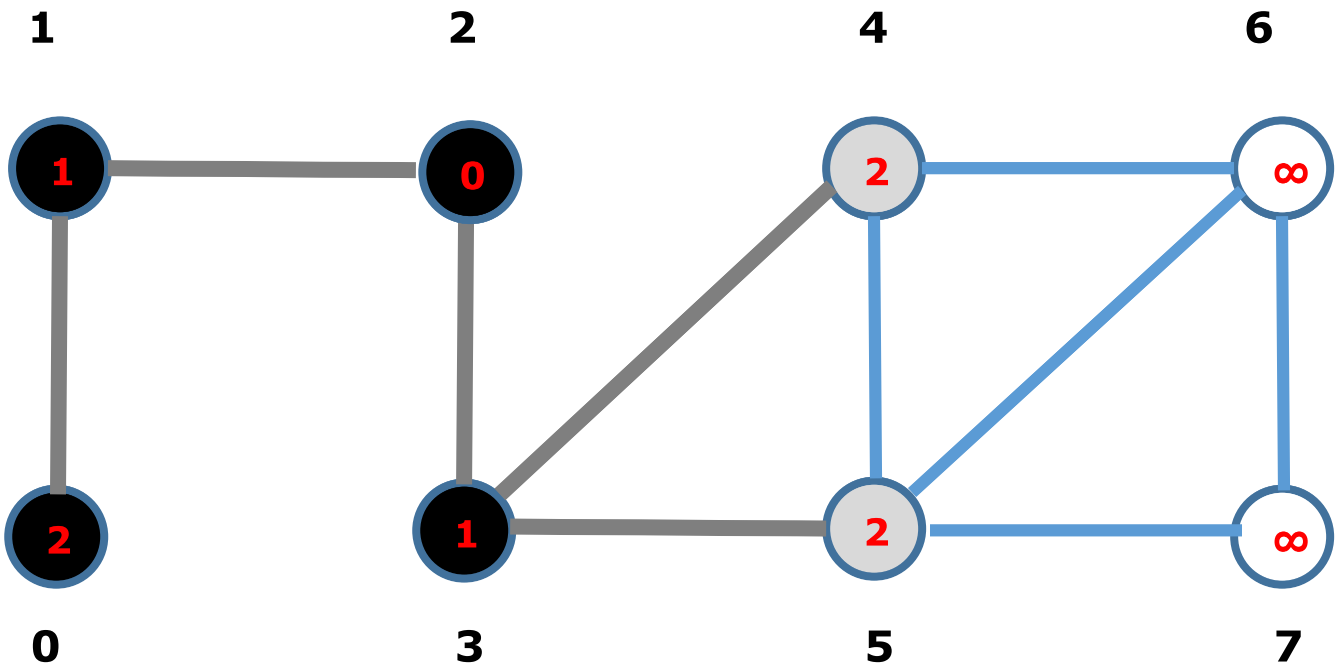

```
while Q ≠ ∅
u = DEQUEUE(Q)
for each v ϵ G:Adj(u)
      if  v.color == WHITE
            v.color = GRAY
            v.dist = u.dist + 1
            v.pred = u
            ENQUEUE(Q, v)
u:color = BLACK
```

u = DEQUEUE(Q) = **0**   graph[0] : [1]

u = 0   v = 1

color: [3, 3, 3, 3, 2, 2, 1, 1]

q : [5, 4]




```
while Q ≠ ∅
u = DEQUEUE(Q)
for each v ∊ G:Adj(u)
      if  v.color == WHITE
            v.color = GRAY
            v.dist = u.dist + 1
            v.pred = u
            ENQUEUE(Q, v)
u:color = BLACK
```

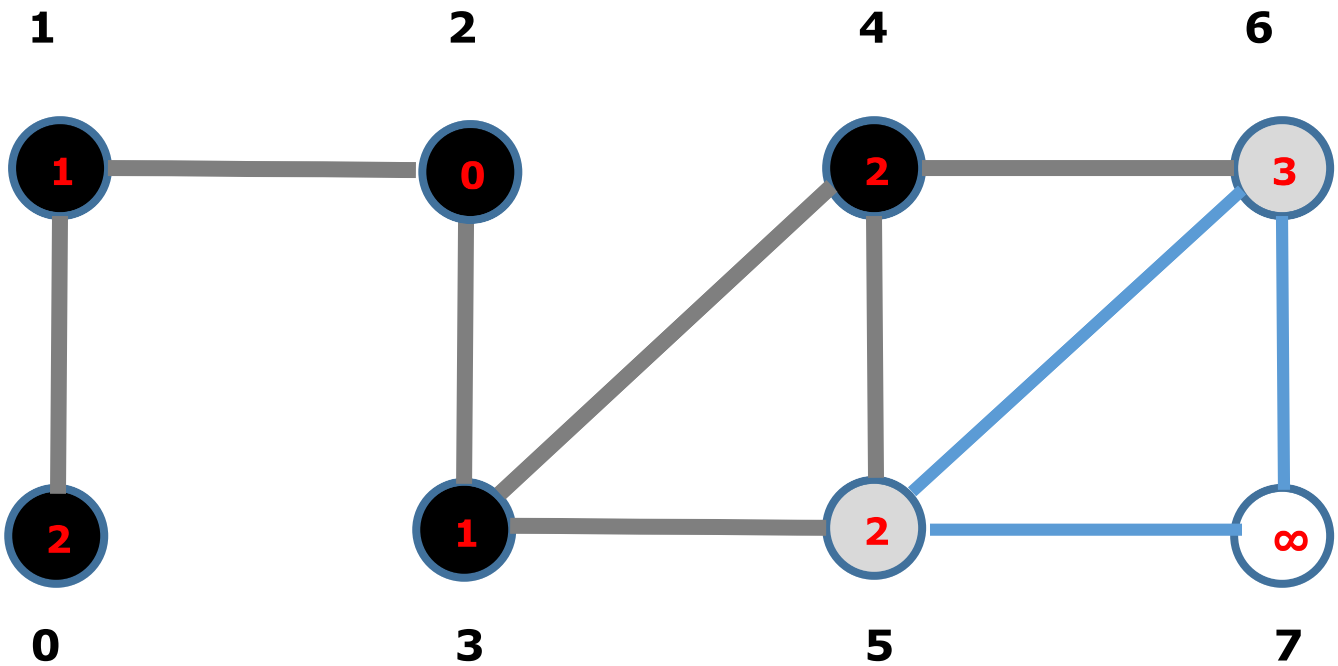
u = DEQUEUE(Q) = **4** graph[4] : [3, 5, 6]

q : [5]

u = 4 **v = 3**
u = 4 **v = 5**
u = 4 **v = 6**
dist[6] = 3 pred[6] = 4 color[6] = 2 **q : [6, 5]**
pred: [**1, 2, -1, 2, 3, 3, 4**, -1]
dist: [**2, 1, 0, 1, 2, 2, 3**, -1]
color: [**3, 3, 3, 3, 2, 2, 2**, 1]

color: [3, 3, 3, 3, 3, 2, 2, 1]

q : [6, 5]




```
while Q ≠ ∅
u = DEQUEUE(Q)
for each v ∈ G:Adj(u)
    if  v.color == WHITE
        v.color = GRAY
        v.dist = u.dist + 1
        v.pred = u
        ENQUEUE(Q, v)
u:color = BLACK
```

u = DEQUEUE(Q) = **5**   graph[4] : [3, 4, 6, 7]

q : [6]

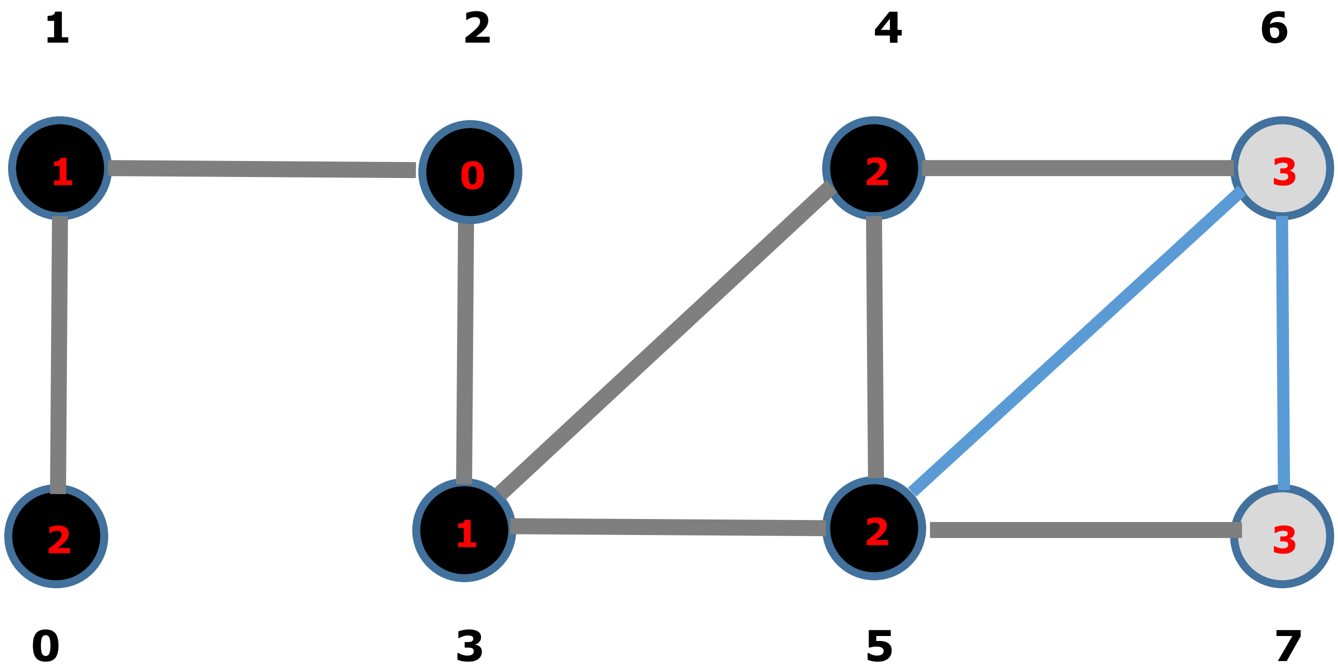
u = 5   v = 3
u = 5   v = 4
u = 5   v = 6
u = 5   v = 7

dist[7] = 3   pred[7] = 5   color[7] = 2   **q : [7, 6]**

pred: [**1, 2, -1, 2, 3, 3, 4**, **5**]
dist: [**2, 1, 0, 1, 2, 2, 3**, **3**]
color: [**3, 3, 3, 3, 2, 2, 2**, **2**]

color: [3, 3, 3, 3, 3, 3, 2, 2]

q : [7, 6]




```
while Q ≠ ∅
u = DEQUEUE(Q)
for each v ϵ G:Adj(u)
      if  v.color == WHITE
            v.color = GRAY
            v.dist = u.dist + 1
            v.pred = u
            ENQUEUE(Q, v)
u:color = BLACK
```

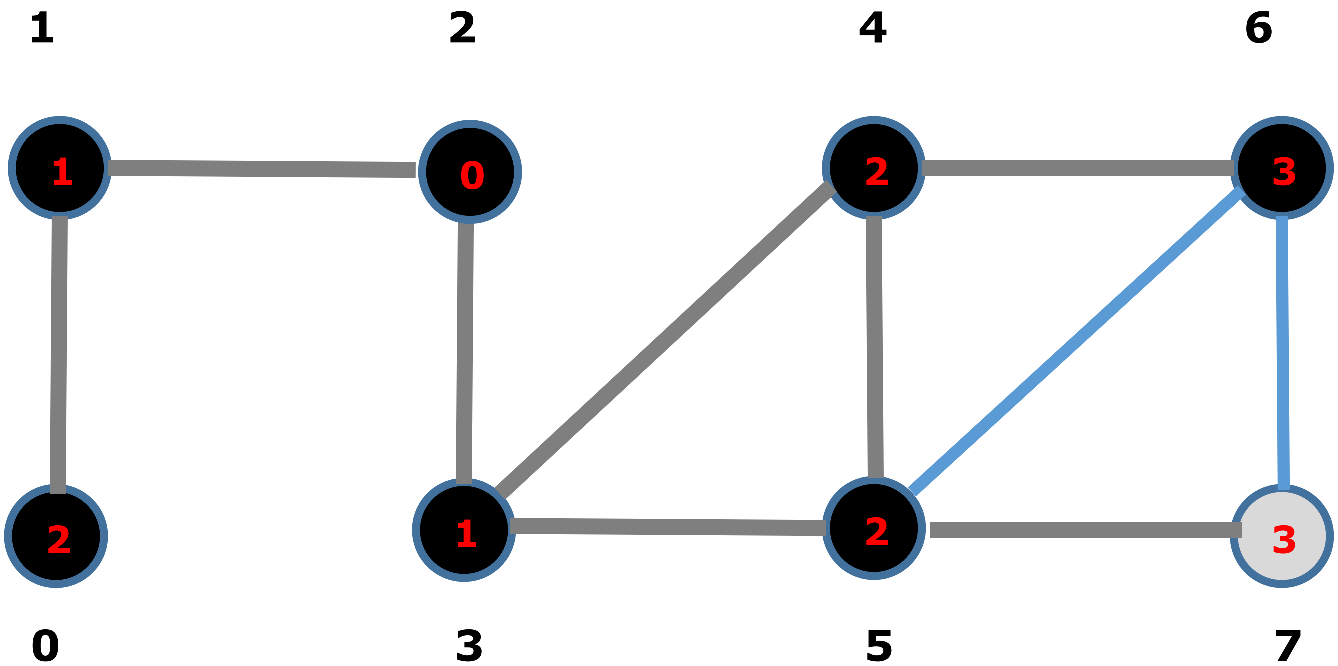
u = DEQUEUE(Q) = **6**   graph[4] : [4, 5, 7]

q : [7]

u = 6   v = 4
u = 6   v = 5
u = 6   v = 7
dist[7] = 3   pred[7] = 5   color[7] = 2   **q : [7]**

color: [3, 3, 3, 3, 3, 3, 3, 2]

q : [7]




```
while Q ≠ ∅
u = DEQUEUE(Q)
for each v ϵ G:Adj(u)
        if  v.color == WHITE
                v.color = GRAY
                v.dist = u.dist + 1
                v.pred = u
                ENQUEUE(Q, v)
u:color = BLACK
```

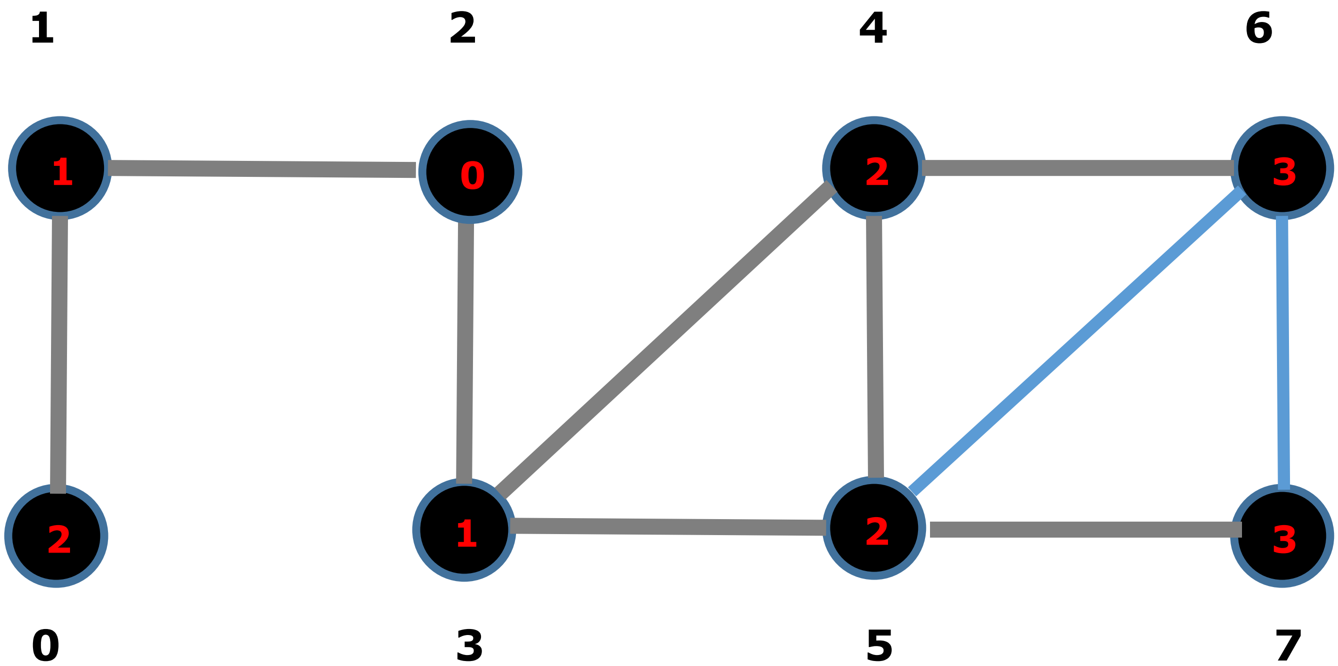
u = DEQUEUE(Q) = **7**   graph[4] : [5, 6]

q : []

u = 7   **v = 5**
u = 7   **v = 6**

color: [3, 3, 3, 3, 3, 3, 3, 3]

q : []



1
2
4
6
1
0
2
3
2
1
2
3
0
3
5
7

כתוב מתודה :

```
public void bfs(int s) { … }
```

BFS applications :

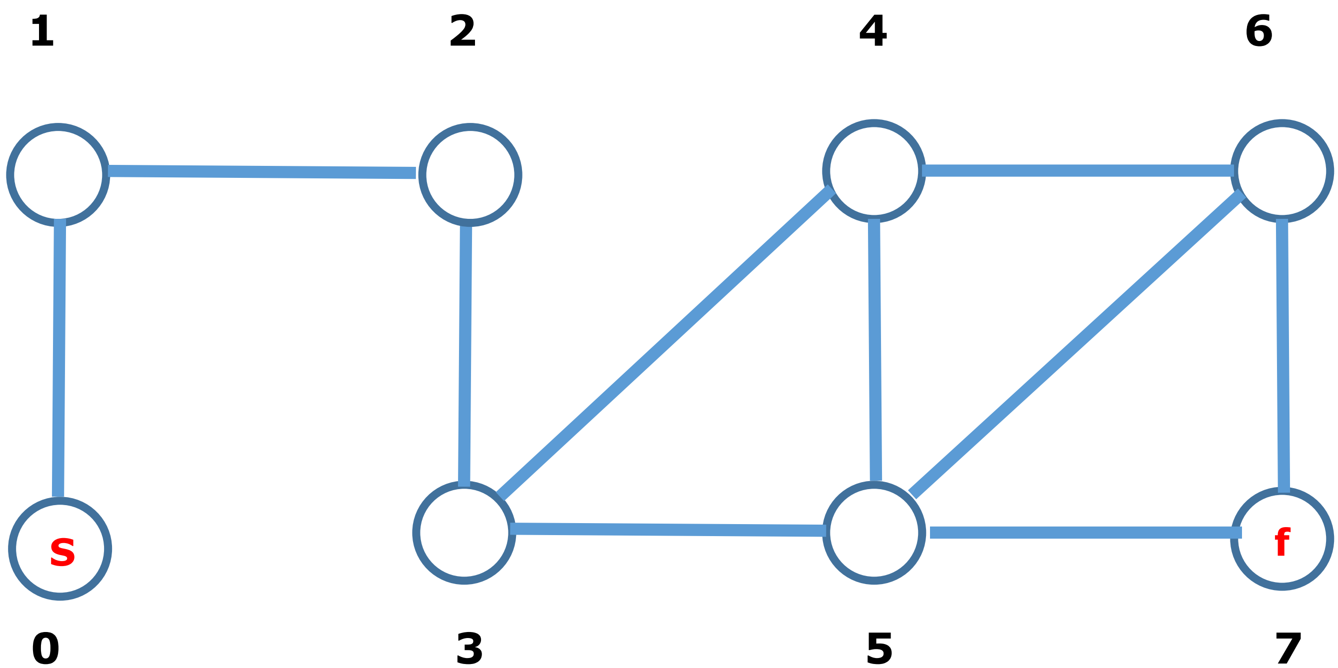
1. Path from vertex to vertex
2. Check the graph connectivity
3. Get component’s array of the graph
4. Get all components of the graph
5. Bipartite graph

Path from vertex s to vertex f.

An adjacency-list representation of G

vertex u	vertex v
0	1
1	0 2
2	1 3
3	2 4 5
4	3 5 6
5	3 4 6 7
6	4 5 7
7	5 6

s → 0 f → 7



After BFS :
read bfs(s)

pred: [-1, 0, 1, 2, 3, 3, 4, 5]
dist: [0, 1, 2, 3, 4, 4, 5, 5]
color: [3, 3, 3, 3, 3, 3, 3, 3]

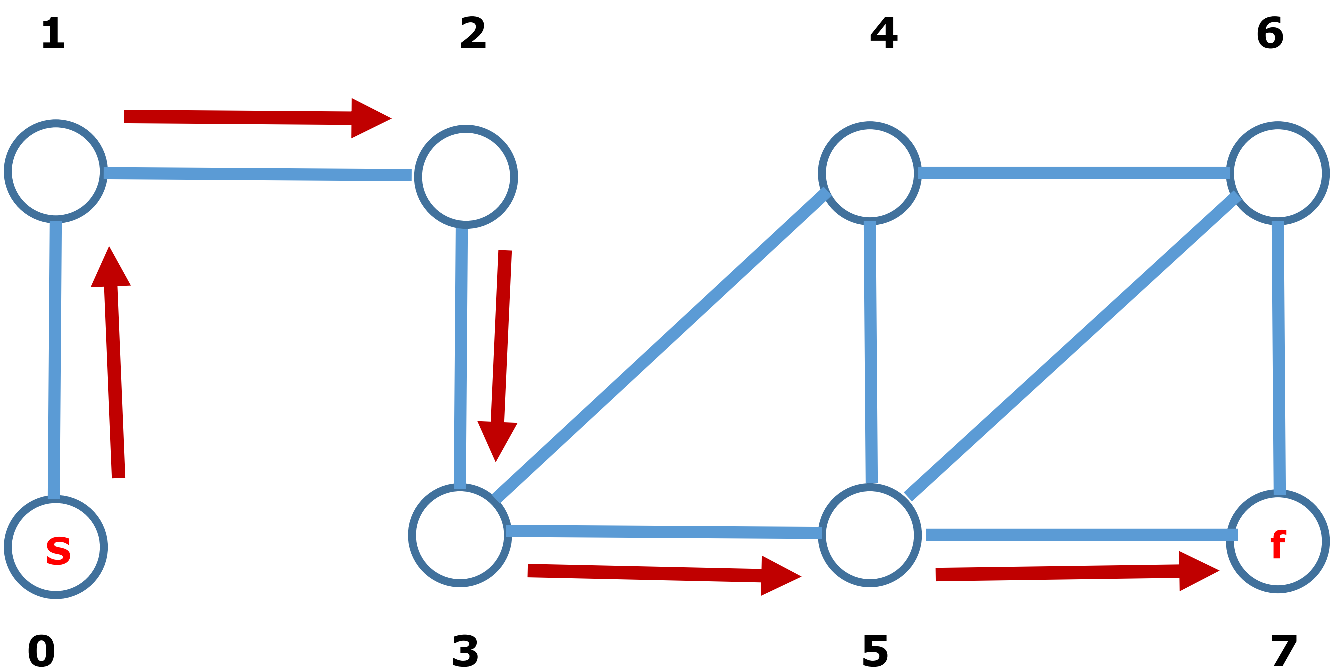
Algorithm :

```
read  bfs(s)
 path = “”

 if  dist[f]==NIL  →   null   →  EXIT

 if  f==s            →   path = path + s  → EXIT

 path = path + f
 t = pred[f]
 while t != NIL
     path = t + “→” + path
     t = pred[t]

  EXIT
```

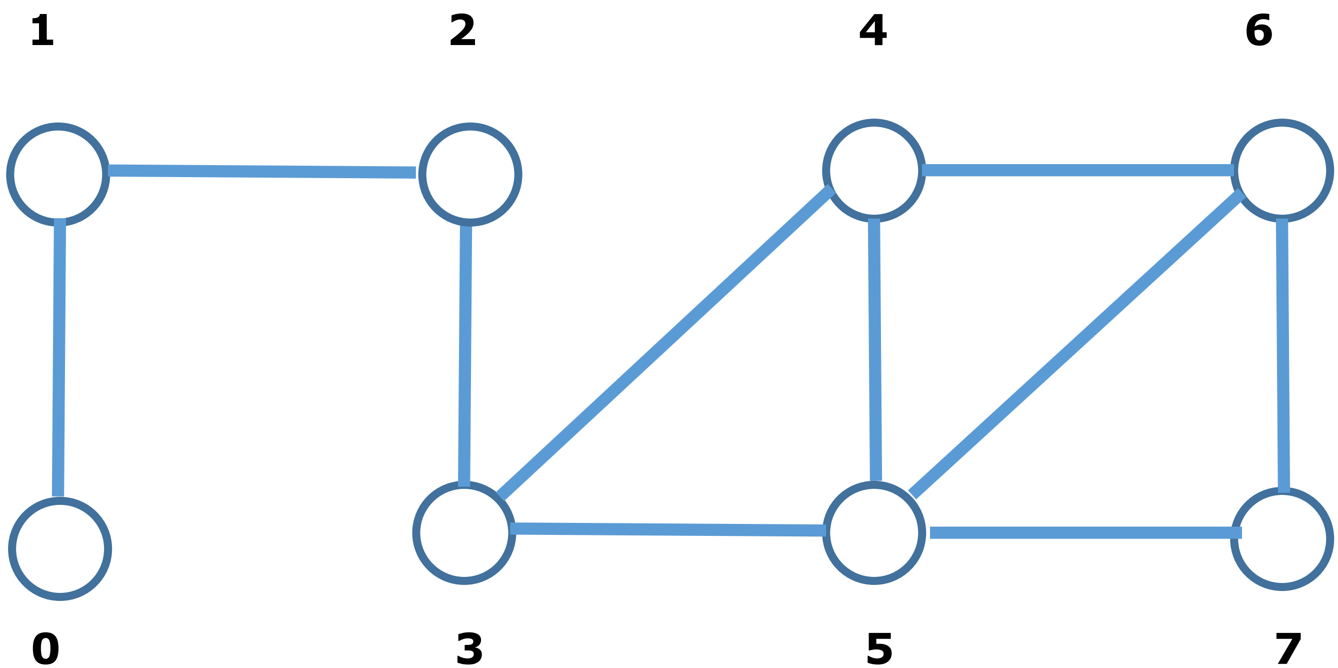
כתוב מתודה :

public String getPath(int s, int f)

Result : 0->1->2->3->5->7



Check the graph connectivity



After BFS :
read BFS(0)

pred: [-1, 0, 1, 2, 3, 3, 4, 5]
dist: [0, 1, 2, 3, 4, 4, 5, 5]
color: [3, 3, 3, 3, 3, 3, 3, 3]

Algorithm :

read bfs(**s**)

boolean ans = true;

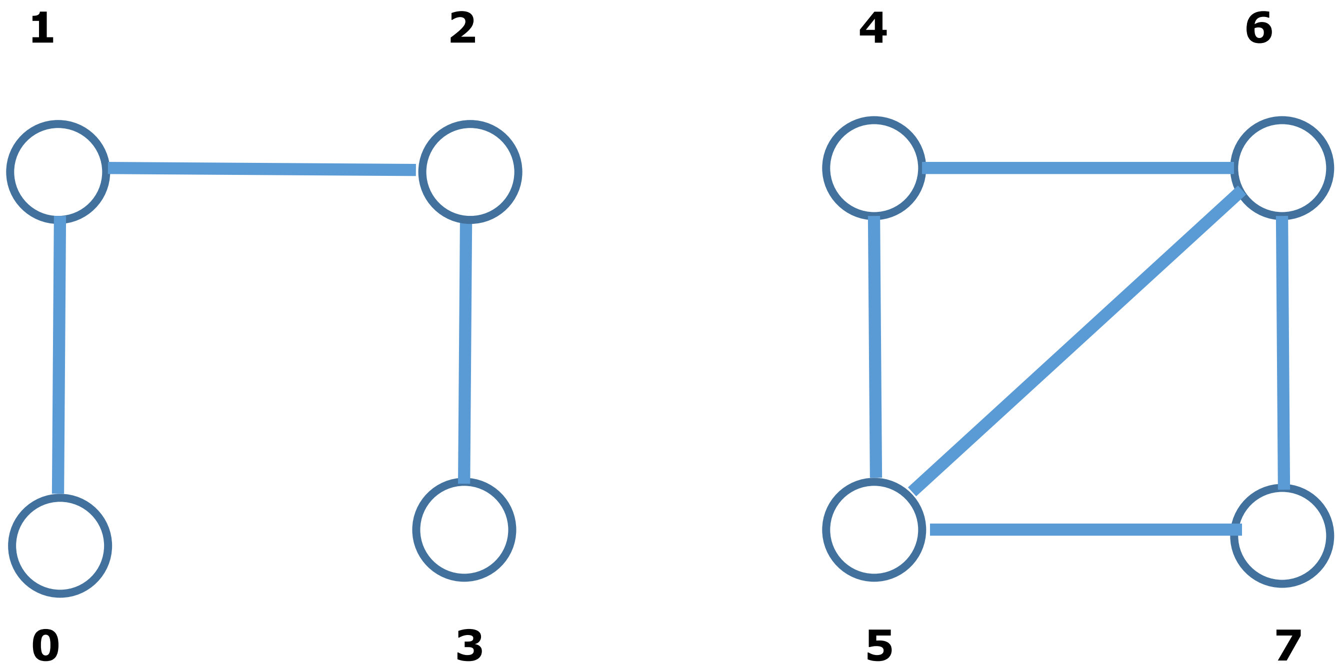
loop begin

```
      i = 0     i < dist.size() and ans          i++
        if  (dist[i] == NIL)
           ans = false
loop end
return ans;
```

כתוב מתודה :

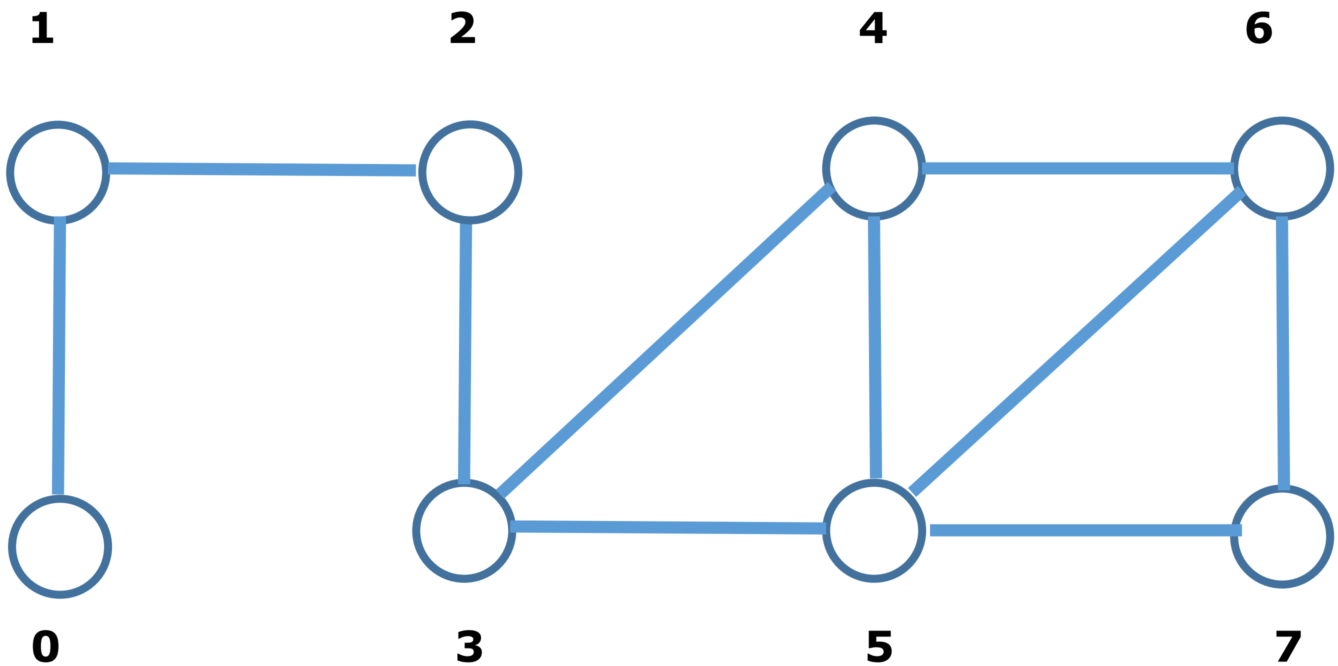
public boolean isConnected()

Result :   is connected? true



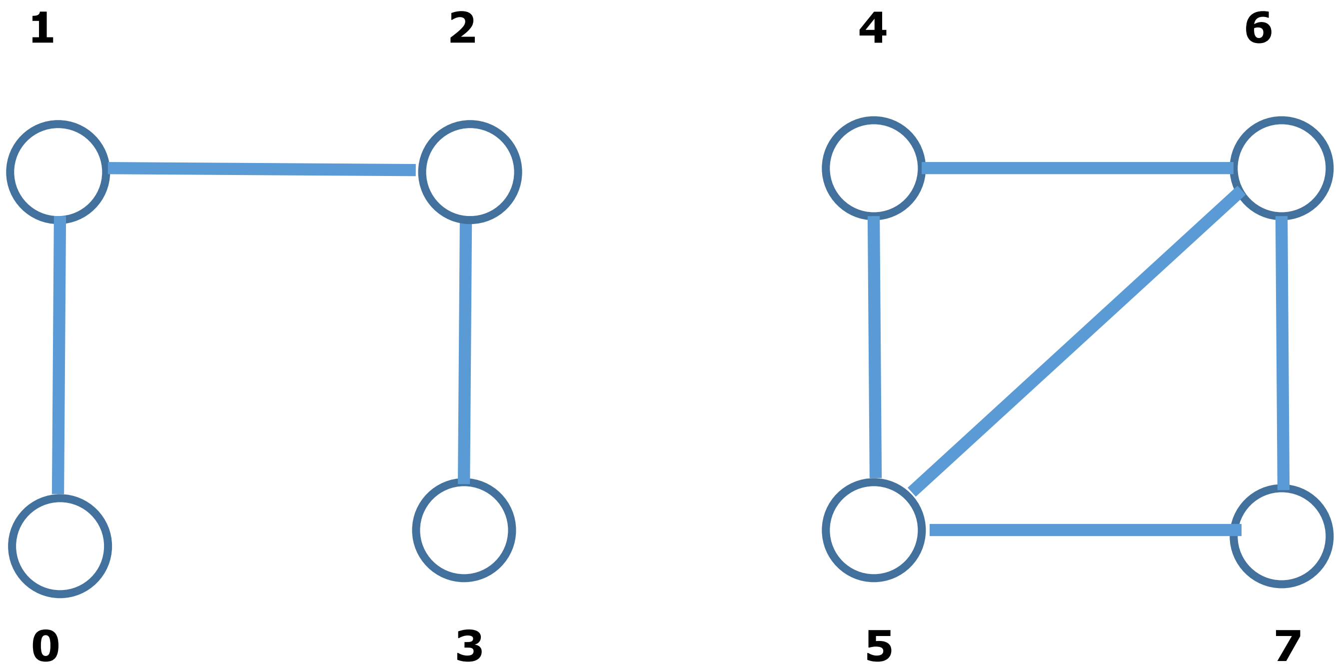
Result : **is connected? false**

Get component's array of the graph



int **components[] = { 1, 1, 1, 1, 1, 1, 1, 1 }**

Example 1



int **components[] = { 1, 1, 1, 1, 2, 2, 2, 2 }**

Example 2

Algorithm :

```
while  (hasNextComponent())
    numComps++
    read  bfs(s)

    loop begin
        i = 0     i < components.length        i++
        if  (dist[i] != NIL)
                components[i] = numComps
    loop end
```

כתוב מתודה :

private void connectedComponents()

Result :

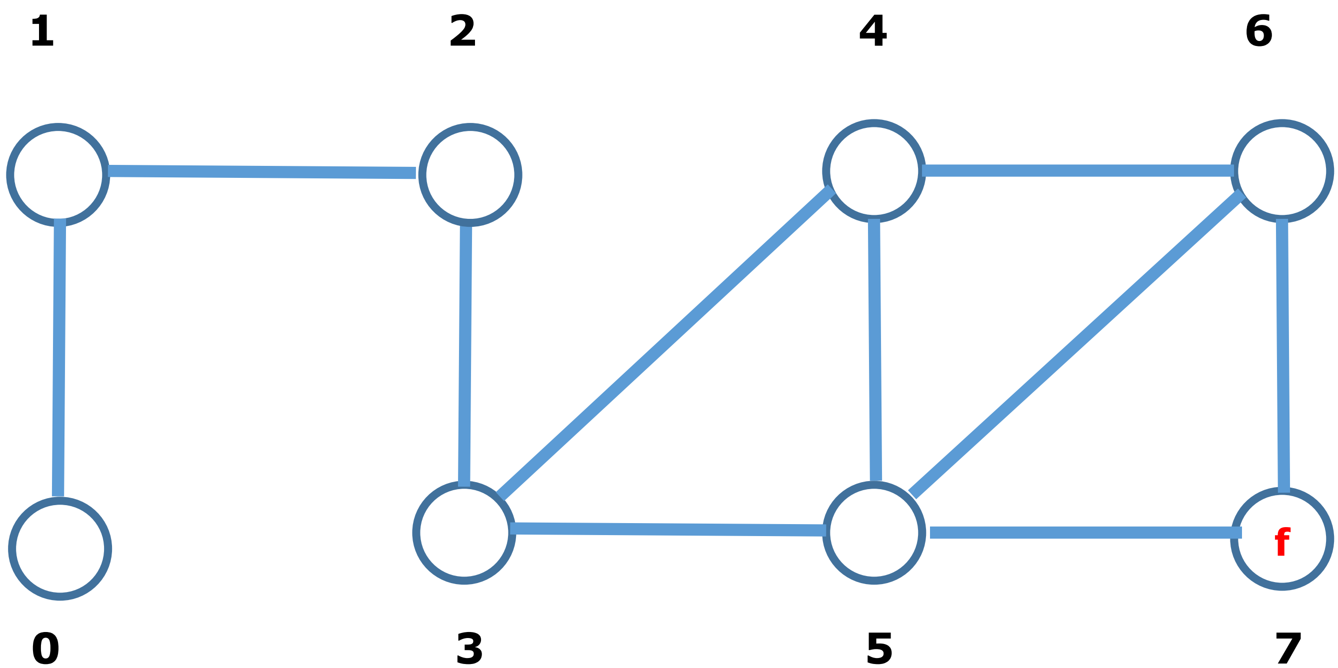
Example 1

components: [1, 1, 1, 1, 1, 1, 1, 1]

Example 2

components: [1, 1, 1, 1, 2, 2, 2, 2]

Get all components of the graph



After BFS :
read BFS(0)

pred: [-1, 0, 1, 2, 3, 3, 4, 5]
dist: [0, 1, 2, 3, 4, 4, 5, 5]
color: [3, 3, 3, 3, 3, 3, 3, 3]

Algorithm :

```
connectedComponents()
ArrayList<Integer>[] compsList = new ArrayList[numComps]
Loop begin
 i = 0       i < compsList.length     i++
          compsList[i] = new ArrayList<Integer>()
Loop end
Loop begin
i = 0       i < compsList.length     i++
     int n = components[i]
        compsList[n-1].add(i)
Loop end
String ans = new String()
```

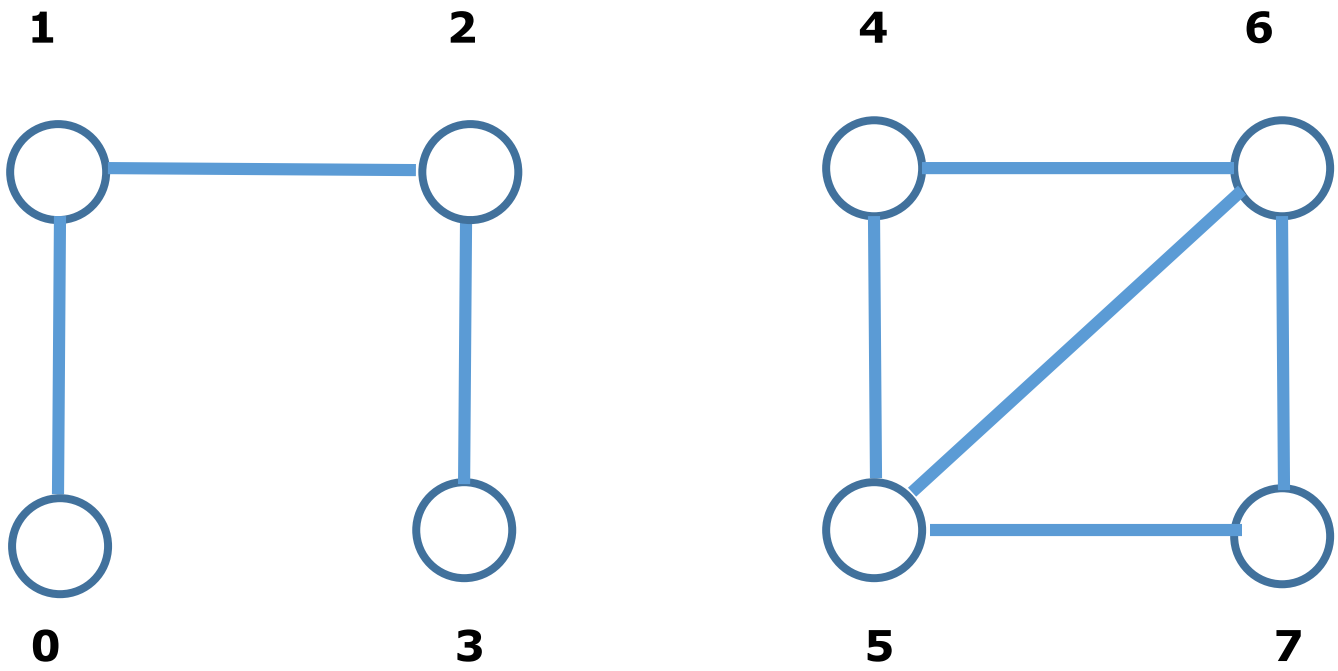
```
Loop begin
i = 0        i < compsList.length      i++
    ans = ans + compsList[i] + "\n"
Loop end
return ans
```

כתוב מתודה :

public String getAllComponents()

Result :

[7 ,6 ,5 ,4 ,3 ,2 ,1 ,0]

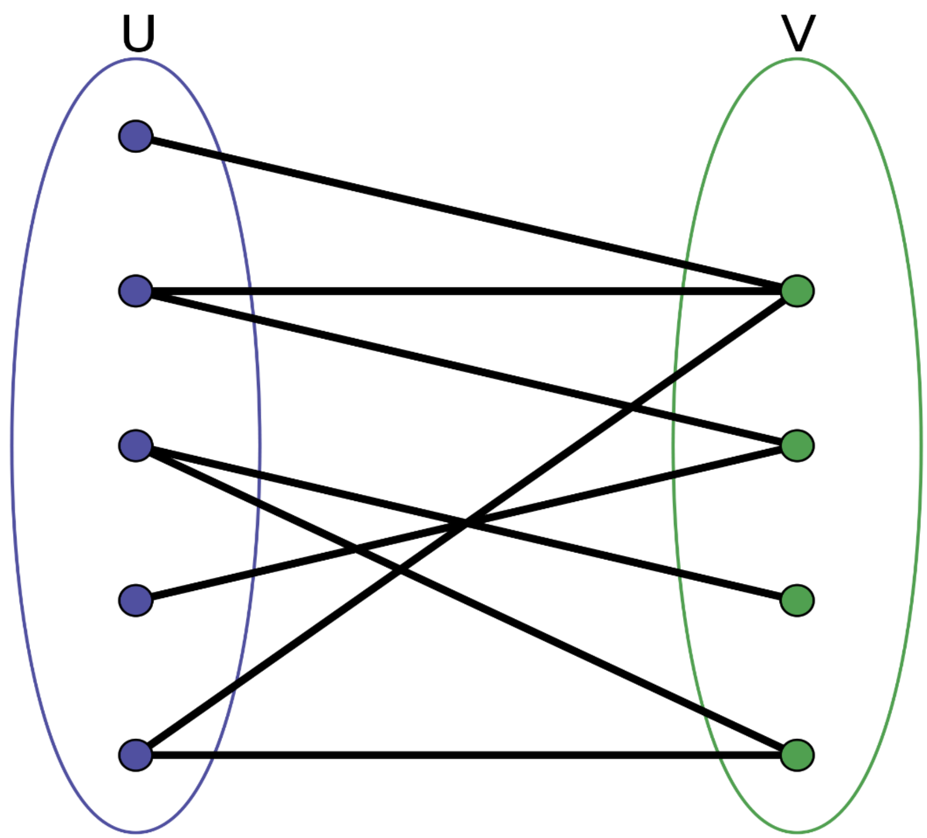


Result :

[3 ,2 ,1 ,0]

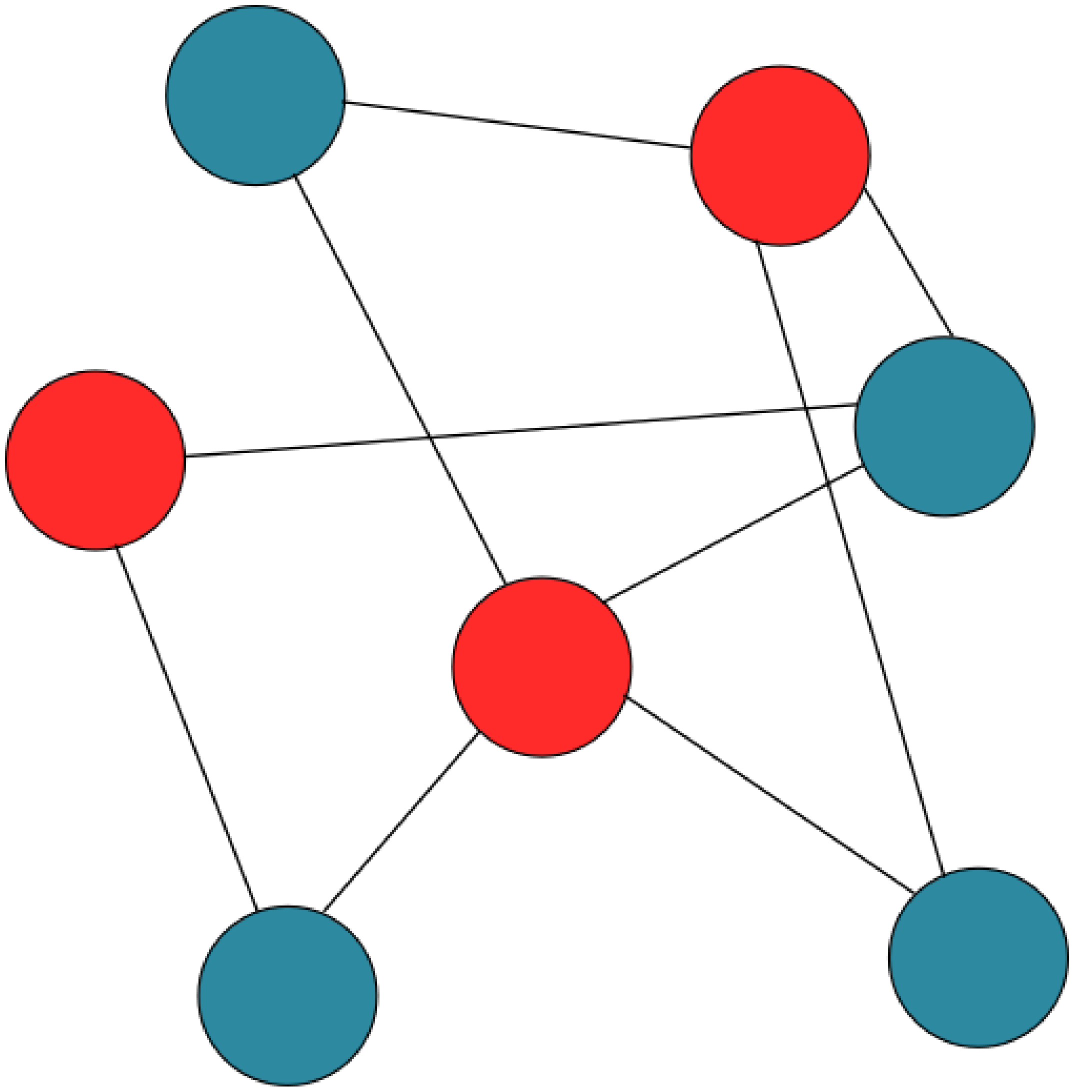
[7 ,6 ,5 ,4]

Bipartite graph

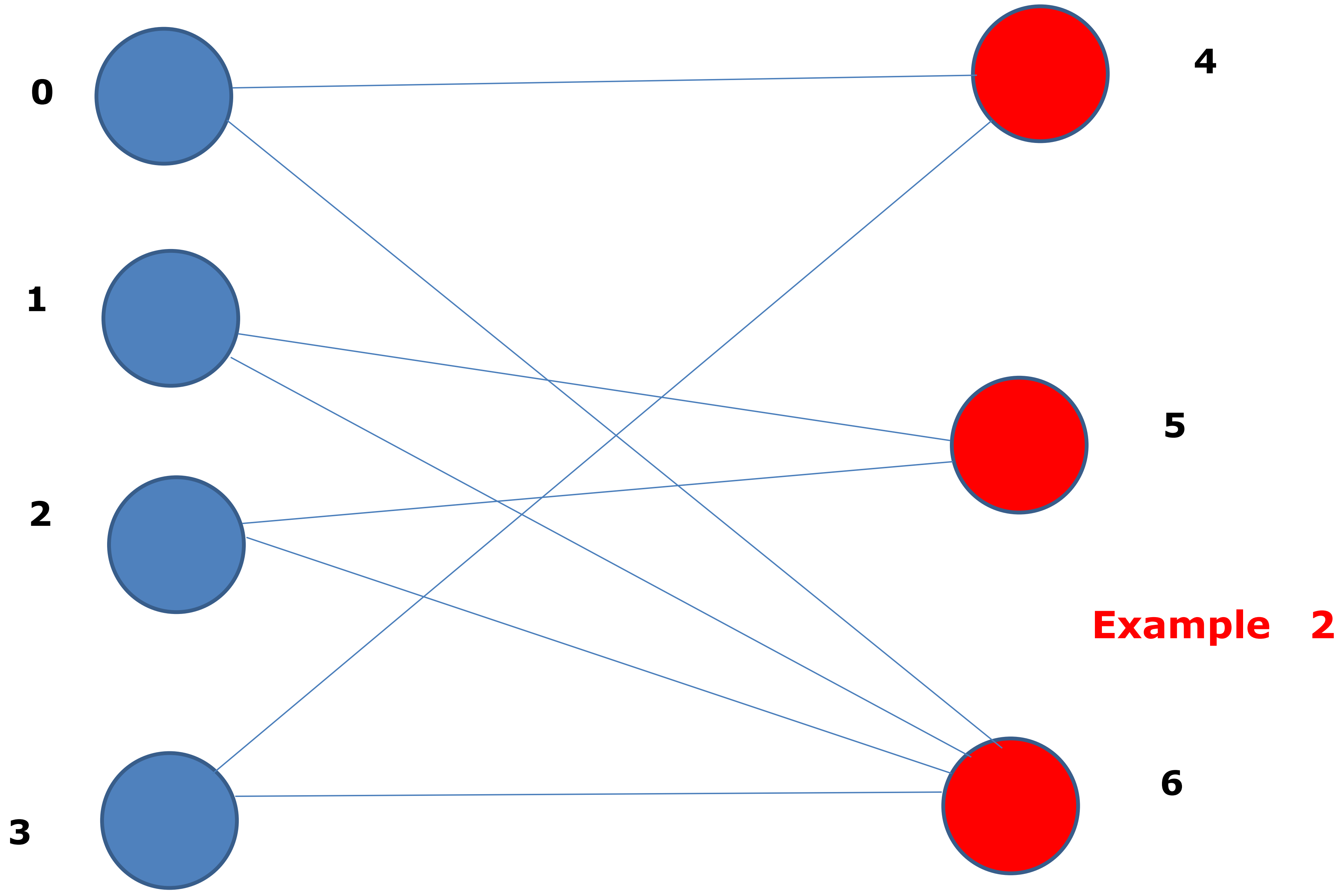


A bipartite graph, also called a **bigraph**, is a set of graph vertices decomposed into two **disjoint** sets such that no two graph vertices within the same set are **adjacent**.

גרף bipartite, המכונה גם digraph, הוא קבוצה של קודקודי גרף מפורקים **לשתי קבוצות זרות** כאלה שאין שני קודקודי גרף בתוך אותה הקבוצה נושקות.



Example 1


0
1
2
3
4
5
6
Example 2

Example 1 & Example 2 are isomorphic

(see Appendix)

How to define that the graph is a bipartite graph ???

Algorithm :

```
boolean  bipartite = true;
for each vertex u in V[G]
    color[u] = WHITE
    distance[u] = infinity
    parent[u] = null
    partition[u] = 0
end for
color[s] = GRAY
distance [s] = 0
parent [s] = null
partition[s] = 1
Q = empty
ENQUEUE(Q, s)
```

```
while (Q != EMPTY and bipartite)
    u = DEQUEUE(Q)
    for each vertex v in Adj[u]
        if (partition [u] = partition [v]){
            bipartite = false;
        }
        else if (color[v] = WHITE) then
             color[v] = GRAY
            distance [v] = d[u] + 1
            parent [v] = u
            partition[v] = 3 - partition[u]
            ENQUEUE(Q, v)
        endif
    endfor
    color[u] = BLACK
end while
return bipartite
```

כתוב מתודה :

public boolean isBipartite ()

Result :

partition : [1, 1, 1, 1, 2, 2, 2]

is bipartite? true

Appendix 1 נספח

Java help code

1. InitGraph class

```
package bfs;

import java.util.ArrayList;

public class InitGraph {

	public static ArrayList<Integer>[] initGraph3(){//connected graph with circle
	    int size = 8;
	    ArrayList<Integer>[] graph = new ArrayList[size];
	    for (int i = 0; i < size; i++) {
	        graph[i] = new ArrayList<Integer>();
	    }
	      graph[0].add(1);

	      graph[1].add(0);
	      graph[1].add(2);

	      graph[2].add(1);
	      graph[2].add(3);

	      graph[3].add(2);
	      graph[3].add(4);
	      graph[3].add(5);
```

```java
        graph[4].add(3);
        graph[4].add(5);
        graph[4].add(6);

        graph[5].add(3);
        graph[5].add(4);
        graph[5].add(6);
        graph[5].add(7);

        graph[6].add(4);
        graph[6].add(5);
        graph[6].add(7);

        graph[7].add(5);
        graph[7].add(6);

        return graph;
    }
```

```java
    public static ArrayList<Integer>[] initGraphBi(){//connected graph with circle
        int size = 7;
        ArrayList<Integer>[] graph = new ArrayList[size];
        for (int i = 0; i < size; i++) {
            graph[i] = new ArrayList<Integer>();
```

```java
        }
          graph[0].add(4);
          graph[0].add(6);

          graph[1].add(5);
          graph[1].add(6);

          graph[2].add(5);
          graph[2].add(6);

          graph[3].add(4);
          graph[3].add(6);

          graph[4].add(0);
          graph[4].add(1);

          graph[5].add(1);
          graph[5].add(2);

          graph[6].add(0);
          graph[6].add(1);
          graph[6].add(2);
          graph[6].add(3);

          return graph;
    }
}
```

2. TestBFS class

```java
package bfs;

public class TestBFS {

    public static void main(String[] args) {
        BFS bfs = new BFS(InitGraph.initGraph3());
        bfs.bfs(2);
        bfs.print();
        System.out.println(bfs.getPath(0, 7));
        System.out.println("is bipartite? " + bfs.isBipartite());
        System.out.println("is connected? " + bfs.isConnected());
        System.out.println(bfs.getAllComponents());

        BFS bi = new BFS(InitGraph.initGraphBi());
        System.out.println(bi.getAllComponents());
        System.out.println("is bipartite? " + bi.isBipartite());
    }
}
```

3. BFS class

```
package bfs;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Queue;
import java.util.concurrent.ArrayBlockingQueue;


public class BFS {

    private  int size;                                           //number of vertexes
    private  Queue<Integer> q;
    private  int dist[], pred[], color[], partition[];
    private  final int WHITE=1, GRAY=2,  BLACK=3, NIL = -1;
    private  ArrayList<Integer> graph[];
    private  int numComps, source;
    private  int components[];
```

```
public BFS(ArrayList<Integer> g[]){
        size = g.length;
        q = new ArrayBlockingQueue<Integer>(size);
        dist = new int[size];
        pred = new int[size];
        color = new int[size];
        partition = new int[size];
        graph = new ArrayList[size];
        components = new int[size];

        for (int i = 0; i < size; i++)
             graph[i] = new ArrayList<Integer>(g[i]);

        source = 0;
        numComps = 0;
}
```

Appendix 2 נספח

In graph theory, an **isomorphism of graphs** *G* and *H* is a bijection between the vertex sets of *G* and *H*

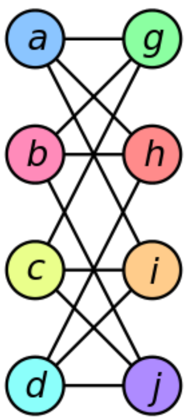
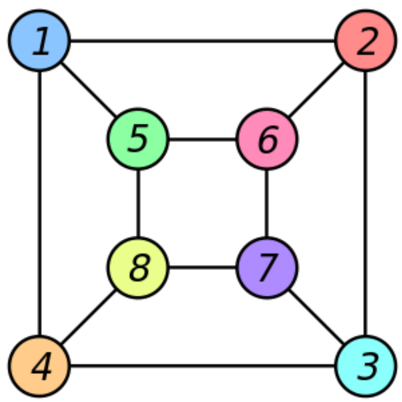
$$f: V(G) \to V(H)$$

such that any two vertices *u* and *v* of *G* are adjacent in *G* if and only if *ƒ*(*u*) and *ƒ*(*v*) are adjacent in *H*. This kind of bijection is commonly described as "edge-preserving bijection", in accordance with the general notion of isomorphism being a structure-preserving bijection.

If an isomorphism exists between two graphs, then the graphs are called **isomorphic** and denoted as $G \simeq H$. In the case when the bijection is a mapping of a graph onto itself, i.e., when *G* and *H* are one and the same graph, the bijection is called an automorphism of *G*.

Graph isomorphism is an equivalence relation on graphs and as such it partitions the class of all graphs into equivalence classes. A set of graphs isomorphic to each other is called an **isomorphism class of graphs**.

The two graphs shown below are isomorphic, despite their different looking drawings.

Graph G	Graph H	An isomorphism between G and H
		$f(a) = 1$ $f(b) = 6$ $f(c) = 8$ $f(d) = 3$ $f(g) = 5$ $f(h) = 2$ $f(i) = 4$ $f(j) = 7$

Glossary

breadth	**רוחב**
disjoint	**פרוק**
adjacent	**סמוך**