task1 final version

April 29, 2020

In [6]:
```
### use generate_graph() to transform adjeceny matrix to graph
import numpy as np
from graph_tools import *
def generate_graph(AM):
    d = Graph(directed=False)
    d.add_vertex(len(AM)-1)
    for i in range(len(AM)):
        for j in range(i,len(AM)):
            if AM[i][j]==1:
                d.add_edge(i,j)
    return d
```

In [7]:
```
generate_graph([[0, 0, 1, 1],[ 0 ,0,1, 1],[ 1, 1, 0, 0],[ 1, 1, 0, 0]])
```

Out[7]:
```
// Generated by graph-tools (version 1.0) at 2020/17/04/29/20 16:17:44
// undirected, 4 vertices, 4 edges
graph export_dot {
  node [color=gray90,style=filled];
  "0";
  "1";
  "2";
  "3";
  "0" -- "2";
  "0" -- "3";
  "1" -- "2";
  "1" -- "3";
}
```

In [8]:
```
###get adjeceny matrix from user
def enter_adjMatrix():
    n = int(input("Enter number of rows(columns) in the matrix: "))
    matrix = []
    print("Enter the %s x %s matrix row by row: "% (n, n))
    for i in range(n):
        matrix.append(list(map(int, input().rstrip().split())))
    return matrix
```

In [9]:
```
AM=enter_adjMatrix()
AM1=enter_adjMatrix()
```

```
Enter number of rows(columns) in the matrix: 4
Enter the 4 x 4 matrix row by row:
0 1 0 1
1 0 1 0
0 1 0 1
1 0 1 0
Enter number of rows(columns) in the matrix: 4
Enter the 4 x 4 matrix row by row:
0 0 1 1
0 0 1 1
1 1 0 0
1 1 0 0
```

```
In [10]: g0=generate_graph(AM)
         g0
```

```
Out[10]: // Generated by graph-tools (version 1.0) at 2020/18/04/29/20 16:18:35
         // undirected, 4 vertices, 4 edges
         graph export_dot {
           node [color=gray90,style=filled];
           "0";
           "1";
           "2";
           "3";
           "0" -- "1";
           "0" -- "3";
           "1" -- "2";
           "2" -- "3";
         }
```

```
In [11]: g1=generate_graph(AM1)
         g1
```

```
Out[11]: // Generated by graph-tools (version 1.0) at 2020/18/04/29/20 16:18:35
         // undirected, 4 vertices, 4 edges
         graph export_dot {
           node [color=gray90,style=filled];
           "0";
           "1";
           "2";
           "3";
           "0" -- "2";
           "0" -- "3";
           "1" -- "2";
           "1" -- "3";
         }
```

```
In [12]: def composite_permutition(p,g):
             g1=[0]*len(p)
```

```
            s=min(g)
            if len(p)==len(g):
                for i in range(len(p)):
                    g1[i]=g[((p[i]-s)%len(p))]
                pass
            return g1
        composite_permutition([0,1,3,2],[0,1,3,2])
```

Out[12]:
```
[0, 1, 2, 3]
```

In [14]:
```
### apply permutation to the adjeceny matrix by using apply_permute()
def apply_permut(AM,p):
    M=np.array(AM)
    s=min(p)
    for i in range(len(p)):
        p[i]=p[i]-s
    for i in range(len(p)):
        M[:,i] = M[p,i]
    for i in range(len(p)):
        M[i,:] = M[i,p]
    return M
```

In [15]:
```
s=apply_permut(AM,[1,4,2,3])
s
generate_graph(s)
```

Out[15]:
```
// Generated by graph-tools (version 1.0) at 2020/18/04/29/20 16:18:39
// undirected, 4 vertices, 4 edges
graph export_dot {
  node [color=gray90,style=filled];
  "0";
  "1";
  "2";
  "3";
  "0" -- "1";
  "0" -- "2";
  "1" -- "3";
  "2" -- "3";
}
```

In [16]:
```
def apply_permut(AM,p):
    M=np.array(AM)
    M1=np.zeros([len(p),len(p)])
    #print(M)
    for i in range(len(p)):
        M1[:,i] = M[p[i],i]
    for i in range(len(p)):
        M1[i,:] = M[i,p]
    return M1
```

```
In [17]: apply_permut([[0, 1,0, 1],[ 1,0, 1,0],[0, 1,0, 1],[ 1,0, 1,0]],[0,3,1,2])

Out[17]: array([[0., 1., 1., 0.],
                [1., 0., 0., 1.],
                [0., 1., 1., 0.],
                [1., 0., 0., 1.]])

In [18]: ## check if two permutation produce the same graph using are_same()
         def are_same(g0,g1):
             s=len(g0.vertices())
             q=len(g1.vertices())

             if s==q :
                 if sorted(g0.edges())==sorted(g1.edges()):
                     return True
                 else:
                     return False
             else :
                 return False

In [19]: are_same(g0,g1)

Out[19]: False

In [20]: def tran_permu_to_graph(g):
             d = Graph(directed=False)
             d.add_vertex(len(g)-1)
             for i in range(len(g)):
                 d.add_edge(g[(i)],g[(i+1)%len(g)])
             return d

In [21]: tran_permu_to_graph([0,3,1,2])

Out[21]: // Generated by graph-tools (version 1.0) at 2020/18/04/29/20 16:18:43
         // undirected, 4 vertices, 4 edges
         graph export_dot {
           node [color=gray90,style=filled];
           "0";
           "1";
           "2";
           "3";
           "0" -- "3";
           "0" -- "2";
           "1" -- "3";
           "1" -- "2";
         }

In [22]: ### find the inverse of permutation using inv()
         def inv(perm):
```

```
            s=min(perm)
            inverse = [0] * len(perm)
            for i, p in enumerate(perm):
                inverse[p-s] = i+s
            return inverse
```

In [23]:
```
pi=[0,3,1,2]
p=inv(pi)
composite_permutition(p,[0,3,1,2])
```

Out[23]:
```
[0, 1, 2, 3]
```

In [24]:
```
### transform the permutation to list of list using

def To_list(l):
    l1=[]
    for i in l:
        l1.append(list(i))
    return l1
```

In [32]:
```
def graph_isomorphism(P,V):
    message_list=[]
    prover_move=True
    while len(message_list)<4:
        if prover_move :
            message_list=P(message_list)
        else :
            message_list=V(message_list)
        prover_move= not prover_move
        print(message_list[-1])
        if message_list[-1]== 'accept' or   message_list[-1]== 'reject':
            return message_list
    return message_list
```

In [33]:
```
import math
def honest_prover(AM,AM1,pi,mess_list):
    n=len(AM)
    sigma = np.random.permutation(range(0,n))
    if len(mess_list)==0:
        h=apply_permut(AM,sigma)
        H=generate_graph(h)
        mess_list.append(H)
    if len(mess_list)==2:
        ch=mess_list[1]
        if ch==0:
            mess_list.append(sigma)
        else:
            p=inv(pi)
            message_list.append(composite_permut(p,sigma))
    return mess_list
```

```
In [34]: def honest_verifier(AM,AM1,mess_list):
             if len(mess_list)==1:
                 ch=random.choice(range(1))
                 mess_list.append(ch)
             if len(mess_list)==3:
                 phi=mess_list[2]
                 H=mess_list[0]
                 H0=apply_permut(AM,phi)
                 H1=generate_graph(H0)
                 r=are_same(H,H1)
                 if r==True:
                     mess_list.append('accept')
                 else:
                     mess_list.append('reject')

             return mess_list
```

```
In [35]: def test_isomorphism():
             # AM=enter_adjMatrix()
              #AM1=enter_adjMatrix()
              #pi=list(map(int, input('Enter pi:').split()))
              V=lambda msg: honest_verifier(AM,AM1,msg)
              P=lambda msg: honest_prover(AM,AM1,pi,msg)
              return graph_isomorphism(P,V)
```

```
In [36]: test_isomorphism()
```

```
// Generated by graph-tools (version 1.0) at 2020/20/04/29/20 16:20:52
// undirected, 4 vertices, 5 edges
graph export_dot {
  node [color=gray90,style=filled];
  "0";
  "1";
  "2";
  "3";
  "0" -- "0";
  "0" -- "3";
  "1" -- "1";
  "1" -- "2";
  "2" -- "3";
}

0
[3 2 0 1]
accept


/usr/lib/python3/dist-packages/ipykernel_launcher.py:11: FutureWarning: elementwise comparison
  # This is added back by InteractiveShellApp.init_path()
```

Out[36]:
```
[// Generated by graph-tools (version 1.0) at 2020/20/04/29/20 16:20:52
 // undirected, 4 vertices, 5 edges
 graph export_dot {
   node [color=gray90,style=filled];
   "0";
   "1";
   "2";
   "3";
   "0" -- "0";
   "0" -- "3";
   "1" -- "1";
   "1" -- "2";
   "2" -- "3";
 }, 0, array([3, 2, 0, 1]), 'accept']
```

In [ ]:

In [ ]: