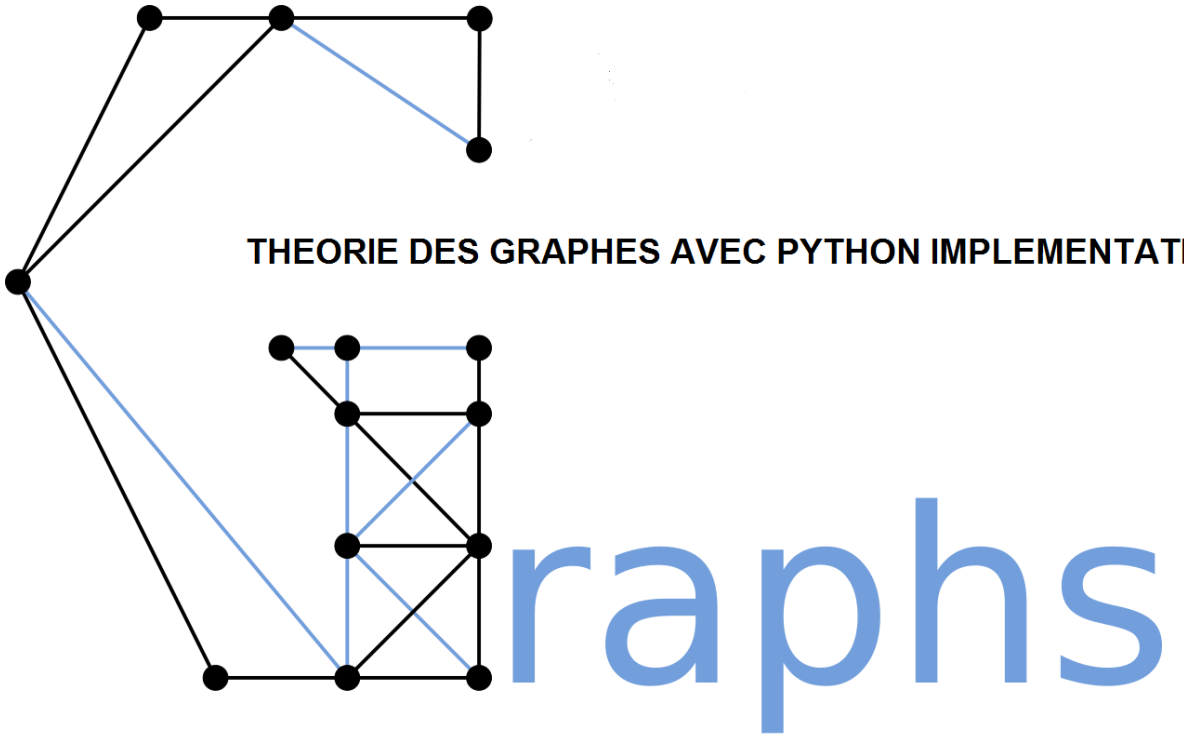

- **PROFESSEUR :**

AZOUAOUI Ahmed

- **REALISE PAR :**

EL IRAOUI Amine

Comment creer un graphe orienté pondéré

Un graphe est constitué d'un ensemble de sommets S d'un ensemble d'arêtes A ⊆ S×S reliant les sommets. Les arêtes sont orientées (flèche) ou non orientées (segment).

- </p>
  Un graphe est valué (ou pondéré) lorsqu'on y associe une fonction de coût sur les arêtes : w : S×S → R
- </p>

In [1]:

```
lSommets=[]
graphe=dict()
arcsCout=dict()
def GrapheOpendere():
    s=input("entrer les sommets sous forme x,y,...:\n")
    l=s.split(',')
    for sommet in l:
        lSommets.append(sommet)
    arcs=input("entrer les arcs sous la forme x.y.cout|a.b.cout|... :\n").split('|')
    for arc in arcs:
        l1=arc.split('.')
        if l1[0] not in graphe.keys():
            graphe[l1[0]]=[]
        graphe[l1[0]].append(l1[1])
        arcsCout[(l1[0],l1[1])]=int(l1[2])
    print(graphe)
    print(arcsCout)
    return graphe
g=GrapheOpendere()
```

```
entrer les sommets sous forme x,y,...:
a,b,c,d
entrer les arcs sous la forme x.y.cout|a.b.cout|... :
a.b.5|b.c.8|c.d.4|d.a.2
{'a': ['b'], 'b': ['c'], 'c': ['d'], 'd': ['a']}
{('a', 'b'): 5, ('b', 'c'): 8, ('c', 'd'): 4, ('d', 'a'): 2}
```

Comment creer un graphe orienté non pondéré

In [89]:

```
def grapheOnonpondere() :
    graph = {}
    arcs=input("entrer les arcs sous la forme x.y|a.b|... :\n").split('|')
    for arc in arcs:
        l1=arc.split('.')
        if not(l1[0] in graph) :
            graph[l1[0]] = list()
        graph[l1[0]].append(l1[1])
    return graph
h=grapheOnonpondere()
h
```

```
entrer les arcs sous la forme x.y|a.b|... :
a.b|b.c|c.d
```

Out[89]:

```
{'a': ['b'], 'b': ['c'], 'c': ['d']}
```

Comment creer un graphe non orienté non pondéré

In [101]:

```
def grapheNonO() :
    graph = {}
    arcs=input("entrer les arcs sous la forme x.y|a.b|... :\n").split('|')
    for arc in arcs:
        l1=arc.split('.')
        if not(l1[0] in graph) :
            graph[l1[0]] = list()
        if not(l1[1] in graph) :
            graph[l1[1]] = list()
        graph[l1[0]].append(l1[1])
        graph[l1[1]].append(l1[0])
    return graph
gn=grapheNonO()
gn
```

```
entrer les arcs sous la forme x.y|a.b|... :
a.b|b.c|c.d
```

Out[101]:

```
{'a': ['b'], 'b': ['a', 'c'], 'c': ['b', 'd'], 'd': ['c']}
```

Matrice d adjacence

Une matrice d'adjacence pour un graphe G à n sommets est une matrice de dimension n × n dont où l'élément à la position i, j vaut 1 s'il existe une arête reliant le sommet i au sommet j, 0 sinon.

In [2]:

```
def Matrice_Adjacence(g):
    """
    """
    matt={}
    matt['0']=['']+lSommets
    print(matt['0'])
    for l in lSommets:
        matt[l]=[l]
        for j in range(len(lSommets)):
            if l in list(g.keys()):
                if lSommets[j] in g[l]:
                    matt[l].append(1)
                else:
                    matt[l].append(0)
            else:
                matt[l].append(0)
        print(matt[l])
Matrice_Adjacence(g)
```

```
['', 'a', 'b', 'c', 'd']
['a', 0, 1, 0, 0]
['b', 0, 0, 1, 0]
['c', 0, 0, 0, 1]
['d', 1, 0, 0, 0]
```

Ajouter un arc

In [3]:

```
def ajouter_arc(g,sommetD,sommetF,poid):
    if sommetD not in lSommets:
        lSommets.append(sommetD)
    if sommetF not in lSommets:
        lSommets.append(sommetF)
    if sommetD not in list(g.keys()):
        g[sommetD]=[]
    g[sommetD].append(sommetF)
    arcsCout[sommetD,sommetF]=int(poid)
ajouter_arc(g,'b','d',6)
g
```

Out[3]:

```
{'a': ['b'], 'b': ['c', 'd'], 'c': ['d'], 'd': ['a']}
```

Supprimer un arc

In [4]:

```
def supprimer_arc(g,s,v) :
    if s in g.keys() :
        if v in g[s] :
            g[s].remove(v)
            return
supprimer_arc(g,'b','d')
g
```

Out[4]:

```
{'a': ['b'], 'b': ['c'], 'c': ['d'], 'd': ['a']}
```

Ajouter un sommet

In [5]:

```
def ajouter_sommet(g, sommet):
        if sommet not in g.keys():
            g[sommet] = []
ajouter_sommet(g, 'e')
g
```

Out[5]:

```
{'a': ['b'], 'b': ['c'], 'c': ['d'], 'd': ['a'], 'e': []}
```

Supprimer une sommet

In [6]:

```
def supprimer_sommet(graph, sommet):
    for n in graph[sommet]:
        if n != sommet:
            graph[n].remove(sommet)
    del graph[sommet]
supprimer_sommet(g, 'e')
g
```

Out[6]:

```
{'a': ['b'], 'b': ['c'], 'c': ['d'], 'd': ['a']}
```

les arcs avec leurs poids

In [7]:

```
arcsCout
```

Out[7]:

```
{('a', 'b'): 5, ('b', 'c'): 8, ('c', 'd'): 4, ('d', 'a'): 2, ('b', 'd'):
6}
```

Taille de graphe

In [8]:

```
def taille_graphe(g):#le nmbre d'arcs
    i=0
    for arc in arcsCout.keys():
        i=i+1
    return i
taille_graphe(g)
```

Out[8]:

```
5
```

Ordre de graphe

In [9]:

```
def ordre_graphe(g):#le nmbre des sommets
    i=0
    for s in lSommets:
        i=i+1
    return i
ordre_graphe(g)
```

Out[9]:

```
4
```

Les successeurs d un sommet

In [10]:

```
def successeurS(g,sommet):
    return g[sommet]
successeurS(g,'a')
```

Out[10]:

```
['b']
```

Les predecesseurs d un sommet

In [11]:

```
def predecesseurS(g,sommet):
    pd=[]
    for k in list(g.keys()):
        if sommet in g[k]:
            pd.append(k)
    if (len(pd) == 0):
        return []
    return pd
predecesseurS(g,'d')
```

Out[11]:

```
['c']
```

Degree d un sommet

In [12]:

```
def DegreeS(g,sommet):
    pred = predecesseurS(g,sommet)
    succ = successeurS(g,sommet)
    return (len(pred)+len(succ))
DegreeS(g,'a')
```

Out[12]:

2

Trouver un chemin entre Sdepart et Sarret

Un chemin de longueur k de u à v dans un graphe G = (S, A) est une suite de sommets u0, u1, ...uk telle que u0 = u, uk = v et on ne passe que par des arêtes de G. ∀i ∈ J0, kK ∈ A

In [13]:

```
def find_chemin(g , Sdepart, Sarret, chemin=None):
    if chemin == None:
        chemin = []
    chemin = chemin + [Sdepart]
    if Sdepart == Sarret:
        return chemin
    if Sdepart not in lSommets:
        return None
    for sommet in g[Sdepart]:
        if sommet not in chemin:
            extended_chemin = find_chemin(g,sommet, Sarret, chemin)
            if extended_chemin:
                return extended_chemin
    return None
find_chemin(g, 'a', 'd')
```

Out[13]:

['a', 'b', 'c', 'd']

Chemin simple ou non

Un chemin simple est un chemin qui ne passe pas 2 fois

par le même sommet.

Exemples sur la figure 1.3 :

— 1, 3, 2, 4 est une chemin de longueur 3, de 1 à 4.

— 1, 4, 5 n'est pas un chemin.

In [14]:

```
ch=find_chemin(g,'a','d')
def cheminisSIMPLE(chemin):
    #Un chemin simple est un chemin qui ne passe pas 2 fois par le même sommet.
    l=chemin
    for s in l:
        if(l.count(s)==1):
            a=1
        else:
            a=0
    if(a==0):
        print("le chemin n'est pa simple")
    else:
        print("le chemin est simple")
cheminisSIMPLE(ch)
```

```
le chemin est simple
```

Trouver toutes les chemins entre un Sdepart et Sarret

In [15]:

```
def find_all_chemins(g, sDepart, sArret, chemin=[]):
        chemin = chemin + [sDepart]
        if sDepart == sArret:
            return [chemin]
        if sDepart not in g.keys():
            return []
        chemins = []
        for s in g[sDepart]:
            if s not in chemin:
                extended_chemins = find_all_chemins(g,s, sArret, chemin)
                for p in extended_chemins:
                    chemins.append(p)
        return chemins
find_all_chemins(g, 'a', 'd')
```

Out[15]:

```
[['a', 'b', 'c', 'd']]
```

Trouver les sommets isolers

In [16]:

```
def find_isolant_sommet(g):
    isoler=[]
    for sommet in g.keys():
        if((predecesseurS(g,sommet) ==[]) & (successeurS(g,sommet) ==[])):
            isoler += [sommet]
    return isoler
find_isolant_sommet(g)
```

Out[16]:

```
[]
```

Trouver le degree minimum des sommets

In [17]:

```
def minimum_degree(g):
    min = 100000000
    for s in g.keys():
        degrees = DegreeS(g,s)
        if degrees < min:
            min = degrees
    return min
minimum_degree(g)
```

Out[17]:

2

Trouver le degree maximum des sommets

In [18]:

```
def maximum_degree(g):
    max = 0
    for s in g.keys():
        degrees = DegreeS(g,s)
        if degrees > max:
            max = degrees
    return max
maximum_degree(g)
```

Out[18]:

2

Trouver une chaine entre deux sommets

In [19]:

```
import random as rand
def Chaine(g , s, t) :
    if s not in g.keys():
        print("n'exite pas une chaine")
    d = list()
    current = s
    while current != t :
        if current not in d:
            d.append(current)
        current = g[current][rand.randint(0, len(g[current]) - 1)]
    d.append(t)
    return d
Chaine(g, 'a', 'd')
```

Out[19]:

['a', 'b', 'c', 'd']

Trouver si le graphe simple ou non

In [20]:

```
def testSimple(g):
    for s in g.keys():
        if s in g[s]:
            sr="le graphe n'est pas simple"
            return sr
        if (find_isolant_sommet(g) != []):
            sr="le graphe n'est pas simple"
            return sr
        for n in g.keys():
            if s in g[n]:
                if s in g.keys():
                    if n in g[s]:
                        sr="le graphe n'est pas simple"
                        return sr
    tr="le graphe simple"
    return tr
testSimple(g)
```

Out[20]:

```
'le graphe simple'
```

Trouver si le graphe complet ou non

In [21]:

```
h={'a':['b','c'],'b':['a','c'],'c':['a','b']}
lSommetsh=['a','b','c']
def grapheComplet(g):
    i=0
    lenghtS=len(lSommets)
    for sommet in lSommets:
        i+=1
        if len(g[sommet]) != lenghtS-1 or sommet in g[sommet]:
            break
    if i == lenghtS:
        print("Ce graphe est complet")
    else:
        print("Ce graphe n'est pas complet")
grapheComplet(g)
```

```
Ce graphe n'est pas complet
```

Graphe conexe ou non

Un graphe est connexe si ∀u, v ∈ S il existe un chemin de u à v dans G.

In [22]:

```
def graphe_connexe(g):
    """Un graphe est connexe si ∀u, v ∈ S il existe un chemin de u à v
    dans G."""
    for s in lSommets:
        for n in lSommets:
            if(find_chemin(g, s, n) != []):
                x=0
            else:
                x=1
    if(x==1):
        print("ce graphe n'est pas connexe")
        return False
    else:
        print('ce graphe est connexe')
        return True
graphe_connexe(g)
```

```
ce graphe est connexe
```

Out[22]:

```
True
```

composante fortement connexe

i) Algorithme pour déterminer une composante fortement connexe d'un Graphe G contenant le sommet a :

1. Donner à a les marques + et -; Aller en 2).

2. Marquer d'un + tout successeur non encore marqué d'un sommet déjà marqué +.

3. Marquer d'un - tout prédécesseur non encore marqué - d'un sommet déjà marqué -.

4. Les sommets marqués à la fois + et - constituent la composante fortement connexe contenant a.
ii) Pour déterminer une 2ème composante fortement connexe éventuelle de G, on répète la procédure ci dessus à partir d'un sommet n'appartenant pas à la première. En répétant au besoin, on détermine toutes les composantes fortement connexes de G.

In [23]:

```
# list des composantes fortement connexes
composants = []
# ensemble des noeuds marqués plus
plus = set()
# ensemble des noeuds marqués moins
moins = set()
# ensemble des noeuds qui ne sont pas encore affectés à une composante
noeuds_restants = set()
```

Marquer les successeurs avec +

In [24]:

```
# marquer les successeurs avec +
def marquer_successeurs(graph,v):
    global plus, noeuds_restants
    succ = graph[v] # récupérer la list des successeurs de ce noeud
    for u in succ: # parcourir les successeurs de v
        if u in noeuds_restants and not u in plus: # si un successeur u n'est pas marqu
é +
            plus.add(u) #marquer u avec +
            marquer_successeurs(graph ,u) # marquer les successeurs de u avec +
marquer_successeurs(g,'a')
```

Marquer les predecesseurs avec -

In [25]:

```
# marquer les predecesseurs avec -
def marquer_predecesseurs(graph,v):
    global moins, noeuds_restants
    for u in graph: #parcourir les éléments u du graph
        if u in noeuds_restants:
            succ = graph[u] #récupérer les successeurs de chaque élément
            if (v in succ) and (u not in moins): # si v est un successeur de u et u n'e
st pas marqué -
                moins.add(u) # marquer u avec -
                marquer_predecesseurs(graph,u) # marquer les predecesseurs de u avec -
marquer_predecesseurs(g,'a')
```

fonction composantes_fortement_connexes

In [26]:

```
def composantes_fortement_connexes(graph):
    #Global: c'est pour accéder aux variables globales
    #Python considère tous qui est dans une fonction comme variable locale
    global composants, plus, moins, noeuds_restants
    # initialiser les listes
    composants = []
    plus = set()
    moins = set()
    # récupérer les noeuds à partir du graph
    noeuds_restants = set(graph.keys())

    while len(noeuds_restants) > 0 : # s'il existe des noeads restants (non affectés)
        v = list(noeuds_restants)[0] # récupérer un des noeuds restants
        plus.add(v) # marquer v avec +
        moins.add(v) # marquer v avec -
        marquer_successeurs(graph,v) # marqer ses successeurs avec +
        marquer_predecesseurs(graph,v) #marquer ses predecesseurs avec -
        inter = plus & moins # ensemble des noeuds marqués + et -
        composants.append(list(inter)) # ajouter cet ensemble comme  composante connexe
        noeuds_restants =  noeuds_restants - inter # supprimer ses éléments de la liste
des noeuds restants
        plus = set() # vider la list des +
        moins = set() # vider la list des -

    return composants # retourner les composantes connexes
composantes_fortement_connexes(g)
```

Out[26]:

```
[['d', 'b', 'a', 'c']]
```

Algorithme de dijkstra

In [27]:

```
graph = {'a':{'b':10,'c':3},'b':{'c':1,'d':2},'c':{'b':4,'d':8,'e':2},'d':{'e':7},'e':{
'd':9}}

def dijkstra(graph,debut,fin):
    poidmin = {}
    predecesseur = {}
    nonvisitedS = graph
    infinie = 9999
    cheminpluscourt = []
    for s in nonvisitedS:
        poidmin[s] = infinie
    poidmin[debut] = 0
    while nonvisitedS:
        minS = None
        for s in nonvisitedS:
            if minS is None:
                minS = s
            elif poidmin[s] < poidmin[minS]:
                minS = s
        for sfils, poid in graph[minS].items():
            if poid + poidmin[minS] < poidmin[sfils]:
                poidmin[sfils] = poid + poidmin[minS]
                predecesseur[sfils] = minS
        nonvisitedS.pop(minS)
    currentNode = fin
    while currentNode != debut:
        try:
            cheminpluscourt.insert(0,currentNode)
            currentNode = predecesseur[currentNode]
        except KeyError:
            print('Path not reachable')
            break
    cheminpluscourt.insert(0,debut)
    if poidmin[fin] != infinie:
        print('le plus court chemin est  ' + str(cheminpluscourt)+" de poid de "+ str(p
oidmin[fin]))

dijkstra(graph, 'a', 'd')
```

```
le plus court chemin est  ['a', 'c', 'b', 'd'] de poid de 9
```

Existence d un cycle

In [28]:

```
def cycle(g):
    chaine=[]
    sTraite=[]
    keys=list(g.keys())
    key=keys[0]
    reponse='false'
    cont=1
    while key in keys:
        y=0
        chaine.append(key)
        sTraite.append(key)
        ind=len(chaine)
        nexts=[]
        for v in g[key]:
            if len(g[v])!=1:
                if ind <= 1:
                    nexts.append(v)
                    break
                if v in chaine:
                    if v != chaine[ind-2]:
                        chaine.append(v)
                        reponse='true'
                        print('il y a un cycle : {}'.format(chaine[chaine.index(v):]))
                        y=1
                        break
                else:
                    nexts.append(v)
        if y == 1:
            break
        elif len(nexts) != 0:
            key=nexts[0]
        else:
            if len([i for i in keys if i not in sTraite]) != 0:
                key=[i for i in keys if i not in sTraite][0]
                chaine=[]
            else:
                print("n'a pas des cycles")
                break
        cont+=1
    return reponse
cycle(g)
```

```
n'a pas des cycles
```

Out[28]:

```
'false'
```

Algorithme de Kruskal (1956)

L'algorithme de Kruskal (1956) permet de trouver un arbre couvrant minimal dans un graphe valué G = (S, A) , w : S×S → R+.

1. Trier les arêtes par ordre de coût.

2. Parcourir toutes les arêtes (u, v) dans cet ordre. Si (u, v) ne forme pas de cycle, l'ajouter à l'arbre.

In [ ]:

```
def create_graph(f):
    G=dict()
    i=0
    while i < len(f):
        if f[i][0] not in G.keys():
            G[f[i][0]]=f[i][1]
            i+=1
    return G
def kruskal(G,grapheP):
    arcs=grapheP.keys()
    #le graphe initial doit etre connexe et sans cycle pour trouver un arbre de recouvr
ement minimal.
    if (graphe_connexe(G)):
        poids_t = 0 # poids total de l'arbre choisit
        poids = {} #les arretes triées
        n = ordre_graphe(G) #le nombres de sommet
        F = [] #l'ensemble des aretes du grahe partiel
        #creer un dictionnaire contenant les arrete et leur poids
        for e in arcs:
            poids[(e[0],e[1])] = int(grapheP[(e[0],e[1])])
        #Trier les e arêtes du graphe par valeurs croissantes.
        #trie la liste en fonction de la valeur de la clé appliquée à chaque élément de
la liste.
        #renvoie la valeur manimale par key pour chaque élément
        poids = sorted(poids.items(), key=lambda t: t[1])
        #metre le plus petit poid dans F
        F.append(poids[0][0])
        #sauvgarder les aretes choisis avec leurs poids
        f_poids = {}
        f_poids[poids[0][0]] = poids[0][1]
        poids_t += poids[0][1]
        del poids[0]
        #A la fin on doit trouver p=n-1
        p = 1  #le nombre d'arretes palacées dans le graphe partiel
        while ((p < (n-1)) & (not(not poids))):
            #ajouté l'arrete min en F pour verifier si il fait un cycle ou pas
            F.append(poids[0][0])
            print(F)
            #creer un graphe
            G1 = create_graph(F)
            print(G1)
            #verifier l'existence du cycle
            if (not(cycle(G1))):
                #incrementation des variable
                p+=1
                poids_t += poids[0][1]
                #sauvgarder l'arete choisis avec son poids
                f_poids[poids[0][0]] = poids[0][1]
                del poids[0]
            else:
                #suprimer l'element ajouté
                F.remove(poids[0][0])
                del poids[0]
        return F , poids_t
    else:
        return "Le graphe n'est pas un arbre"
kruskal(g,arcsCout)
```

Algorithme de Prim

In [30]:

```
def Prim(g,gpoids,Sstart):
    prim={Sstart:0}
    iteration=1
    poidMin=0
    nextSommet=dict()
    if (graphe_connexe(g)):
        while iteration < len(lSommets):
            for j in g[Sstart]:
                if j not in prim.keys():
                    nextSommet[Sstart,j]=gpoids[Sstart,j]
            trierSommets=sorted(nextSommet.items(), key=lambda t: t[1])
            snext=[i for i in trierSommets if i[0][1] not in prim.keys()][0]
            poidMin+=snext[1]
            prim[snext[0][1]]=snext[1]
            del nextSommet[snext[0][0],snext[0][1]]
            Sstart=snext[0][1]
            iteration+=1
        print("L'arbre de prim est {} de poids Min {}".format(prim,poidMin))
    else:
        print(" enter une graphe connexe !!!")
Prim(g,arcsCout,'a')
```

```
ce graphe est connexe
L'arbre de prim est {'a': 0, 'b': 5, 'c': 8, 'd': 4} de poids Min 17
```

Algorithme de Coloration

In [67]:

```
gc = {}
gc['a'] = ['b', 'c']
gc['b'] = ['a', 'c ', 'd']
gc['c'] = ['a', 'b', 'e']
gc['d'] = ['b', 'c']

def coloration(g):
    colors = ['Red', 'Blue', 'Green', 'Yellow', 'Black']
    states = g.keys()
    neighbors = {}
    for s in states:
        neighbors[s]=g.get(s)
    colors_of_states = {}
    def promising(state, color):
        for neighbor in neighbors[state]:
            color_of_neighbor = colors_of_states.get(neighbor)
            if color_of_neighbor == color:
                return False
        return True
    def get_color_for_state(state):
        for color in colors:
            if promising(state, color):
                return color
    def affich():
        for state in states:
            colors_of_states[state] = get_color_for_state(state)
        print (colors_of_states)
    return affich()
coloration(gc)
```

```
{'a': 'Red', 'b': 'Blue', 'c': 'Green', 'd': 'Red'}
```


Merci de votre attention
EL IRAOUI Amine