

Jean de Bonfils Lavernelle

Matthieu Pierre

Rapport du projet de grammaire

Tout d'abord, nous avons choisi le python pour programmer ces deux algorithmes. Nous avons choisi ce langage pour la simplicité de la syntaxe ainsi que pour les nombreuses opérations qu'offre ce langage sur les listes et les tableaux.

Transformation d'un AFN en AFD

Explications du code

Nous avons programmé, pour cet exercice, deux fonctions. La première fonction permet de rechercher un tableau dans un autre tableau. La deuxième fonction, retourne la liste des successeurs à partir d'un état et d'un symbole.

L'utilisateur commence par saisir le nombre d'état et renseigne les différents liens de l'automate. L'ensemble des liens possibles de l'automate est enregistré dans un tableau 1D pour le symbole A et un autre pour le symbole B. Il est cependant très facile de retrouver un prédécesseur ou un successeur dans ces tableaux. La taille du tableau sera de taille $NbEtat^2$ ce qui permettra de visualiser tous les liens possibles entre les états.

Par exemple :

Voici un automate tel qu'il pourrait être enregistré :

TABA = [0,1,0,1,0,0,0,0]

Nous pouvons facilement obtenir le nombre d'état en faisant $TAILLE(TABA)^{1/2}$. Ici le premier 0 signifie qu'il n'existe pas de lien avec comme symbole A allant de 0 vers 0. Connaissant le nombre d'état de l'automate, nous savons donc que les nbEtat premiers du tableau seront donc les liens ayant pour prédécesseur 0. Il est plus facile de visualiser ce tableau sous cette forme.

	0	1	2
0	0	1	0
1	1	0	0
1	0	0	0

Tous les 1 se trouvant dans le rectangle rouge nous permettent de savoir quels sont les prédécesseurs de 0. Tous les 1 se trouvant dans le rectangle vert nous permettent de savoir quels sont les successeurs de 0.

```
for i in range(nbEtat):
    #Pour accéder au successeur d'un état il faut
    if(listeA[nbEtat*numeroEtat+i] == 1):
        #récupère le num de l'état
        tab.append(i)
```

Ici, le programme parcourt les éléments du rectangle vert (sur le schéma ci-dessus) et si cet élément est à 1 alors on récupère son successeur.

Et voici le code pour accéder au prédécesseur :

```
for i in range(nbEtat):
    if(listeA[numeroEtat+(i*nbEtat)] == 1):
        #récupère le num de l'état
        tab.append(i)
```

Concernant la conversion en AFD, le programme ajoute, dans la liste des nouveaux états, l'état initial.

Tant que tous les états de la liste des nouveaux états ne sont pas traités.

On récupère la liste pour le symbole « A » des états prédécesseurs de l'état courant

On récupère la liste pour le symbole « B » des états prédécesseurs de l'état courant

Si cette liste n'est pas un ensemble d'état correspondant à un super état dans la liste des nouveaux états.

Alors on ajoute cette liste à la liste des nouveaux états.

```
tableau1 = list(set(sum(tableau1, [])))
```

Dans le code, est utilisé la fonction sum. Celle-ci permet de passer un tableau 2D à un tableau 1D.

Par exemple : [1,2, [1,2]] retourne [1,2,1,2]

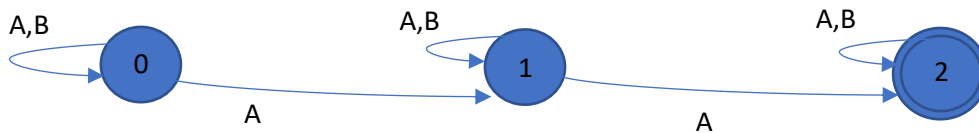
La fonction set () permet quant à elle de détruire les doublons.

Par exemple [1,2,1,2] retourne [1,2]

Exemples

Exemple1 :

Prenons, l'automate suivant :



L'état 0 est l'état initial.

A partir de cette automate, il suffit de saisir tous les liens entre les différents états. 1 si le lien existe, 0 sinon. Il faut aussi saisir le nombre d'état de l'automates ainsi que son état final.

```
Saisir le nombre d'etat : 3
L'etat initiale doit etre 0
A : Successeur 0      Prédécesseur : 0 ? 1/0 :1
A : Successeur 0      Prédécesseur : 1 ? 1/0 :1
A : Successeur 0      Prédécesseur : 2 ? 1/0 :0
B : Successeur0       Prédécesseur : 0 ? 1/0 :1
B : Successeur0       Prédécesseur : 1 ? 1/0 :0
B : Successeur0       Prédécesseur : 2 ? 1/0 :0
A : Successeur 1      Prédécesseur : 0 ? 1/0 :0
A : Successeur 1      Prédécesseur : 1 ? 1/0 :1
A : Successeur 1      Prédécesseur : 2 ? 1/0 :1
B : Successeur1       Prédécesseur : 0 ? 1/0 :0
B : Successeur1       Prédécesseur : 1 ? 1/0 :1
B : Successeur1       Prédécesseur : 2 ? 1/0 :0
A : Successeur 2      Prédécesseur : 0 ? 1/0 :0
A : Successeur 2      Prédécesseur : 1 ? 1/0 :0
A : Successeur 2      Prédécesseur : 2 ? 1/0 :1
B : Successeur2       Prédécesseur : 0 ? 1/0 :0
B : Successeur2       Prédécesseur : 1 ? 1/0 :0
B : Successeur2       Prédécesseur : 2 ? 1/0 :1
Saisir l'etat finale entre 0 et 2 2
```

Voici le résultat :

```
Nouveau super etat :[0]
Successeur avec symbole 'A' : [0, 1]
Successeur avec symbole 'B' : [0]
Cet etat est initial
|
Nouveau super etat :[0, 1]
Successeur avec symbole 'A' : [0, 1, 2]
Successeur avec symbole 'B' : [0, 1]

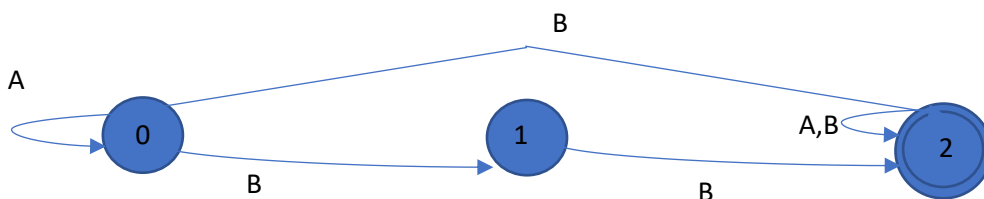
Nouveau super etat :[0, 1, 2]
Successeur avec symbole 'A' : [0, 1, 2]
Successeur avec symbole 'B' : [0, 1, 2]
Cet état est terminal

>>>
```

Le programme nous retourne, dans ce cas, 3 super-états, avec pour chaque état leur successeur en « A » et en « B » ainsi que l'état final et initial. Le programme nous retourne bien un AFD. Après vérification sur papier, nous obtenons le même automate.

Exemple 2 :

Prenons, un nouvel automate :



Ici, l'état 0 est l'état initial.

De même que précédemment, nous indiquons au programme les liens entre les états :

```

Saisir le nombre d'etat : 3
L'etat initiale doit etre 0
A : Successeur 0      Prédécesseur : 0 ? 1/0 :1
A : Successeur 0      Prédécesseur : 1 ? 1/0 :0
A : Successeur 0      Prédécesseur : 2 ? 1/0 :0
B : Successeur0       Prédécesseur : 0 ? 1/0 :0
B : Successeur0       Prédécesseur : 1 ? 1/0 :1
B : Successeur0       Prédécesseur : 2 ? 1/0 :1
A : Successeur 1      Prédécesseur : 0 ? 1/0 :0
A : Successeur 1      Prédécesseur : 1 ? 1/0 :1
A : Successeur 1      Prédécesseur : 2 ? 1/0 :0
B : Successeur1       Prédécesseur : 0 ? 1/0 :0
B : Successeur1       Prédécesseur : 1 ? 1/0 :1
B : Successeur1       Prédécesseur : 2 ? 1/0 :0
A : Successeur 2      Prédécesseur : 0 ? 1/0 :0
A : Successeur 2      Prédécesseur : 1 ? 1/0 :0
A : Successeur 2      Prédécesseur : 2 ? 1/0 :0
B : Successeur2       Prédécesseur : 0 ? 1/0 :0
B : Successeur2       Prédécesseur : 1 ? 1/0 :1
B : Successeur2       Prédécesseur : 2 ? 1/0 :0
Saisir l'état finale entre 0 et 2 1

```

Le programme, nous retourne bien un AFD équivalent à l'AFN saisi :

```

Nouveau super etat :[0]
Successeur avec symbole 'A' : [0]
Successeur avec symbole 'B' : [1, 2]
Cet etat est initial

Nouveau super etat :[1, 2]
Successeur avec symbole 'A' : [1]
Successeur avec symbole 'B' : [1]
Cet état est terminal

Nouveau super etat :[1]
Successeur avec symbole 'A' : [1]
Successeur avec symbole 'B' : [1]
Cet état est terminal

```

Minimisation d'un automate

Explications du code

Tout d'abord, nous avons commencé par programmer 4 fonctions utiles :

La première effectue juste un OU Logique entre deux nombres binaires de taille 1. La deuxième, « estNouveauEtat » permet de vérifier si un super état est déjà dans la pile des nouveaux états. La 3^{ème} fonction permet, à partir du symbole et du numéro de l'état de retrouver la liste de ces successeurs.

L'utilisateur commence par saisir le nombre d'état et renseigne les différents liens de l'automate. L'ensemble des liens possibles de l'automate est enregistré dans un tableau 1D pour le symbole A et un autre pour le symbole B. Il est cependant très facile de retrouver un prédécesseur ou un successeur dans ces tableaux. La taille du tableau sera de taille $NbEtat^2$ ce qui permettra de visualiser tous les liens possibles entre les états.

Par exemple :

Voici un automate tel qu'il pourrait être enregistré :

TABA = [0,1,0,1,0,0,0,0,0]

Nous pouvons facilement obtenir le nombre d'état en faisant $TAILLE(TABA)^{1/2}$. Ici le premier 0 signifie qu'il n'existe pas de lien avec comme symbole A allant de 0 vers 0. Connaissant le nombre d'état de l'automate, nous savons donc que les nbEtat premiers du tableau seront donc les liens ayant pour prédécesseur 0. Il est plus facile de visualiser ce tableau sous cette forme.

	0	1	2
0	0	1	0
1	1	0	0
1	0	0	0

Concernant la minimisation de l'automate, nous avons repris l'algorithme vu en cours consistant à isoler les couples distinguables. Tout d'abord nous ajoutons tous les couples composés d'un état final.

Tant qu'il reste des couples à traiter dans la liste

Si les deux états ont tous les deux, deux état successeur en a ou deux état successeur en b

Alors on empile ces deux états successeurs dans la liste

Une fois ceci fait, le programme regarde les couples qui ne se trouve pas dans la liste et réunit les liens des deux états de ces couples grâce à un OU logique.

Par exemple si 0 et 1 doivent être réunis :

	0	1	2
0 :	0	1	0
1	1	0	0

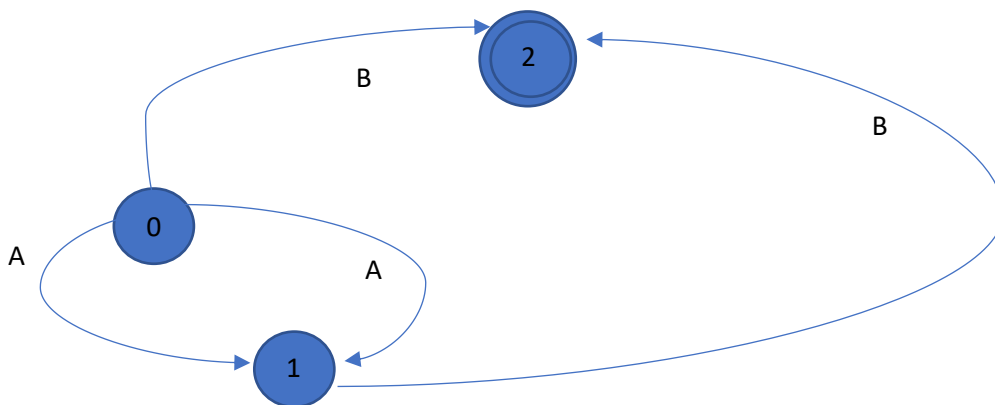
Cela donnera :

	0	1	2
0,1	1	1	0

Une fois ceci fait nous disposons des nouveaux super-état ainsi que leurs différents liens vers les états.

Exemples

Prenons l'automate suivant :



Dans cette automate, l'état 0 est l'état initial. De même que pour la partie 1 nous indiquons les liens de l'automate ainsi que son état final.

```

Saisir le nombre d'etat : 3
L'etat initiale doit etre 0
A : Successeur 0    Prédécesseur : 0 ? 1/0 :0
A : Successeur 0    Prédécesseur : 1 ? 1/0 :1
A : Successeur 0    Prédécesseur : 2 ? 1/0 :0
B : Successeur0     Prédécesseur : 0 ? 1/0 :0
B : Successeur0     Prédécesseur : 1 ? 1/0 :0
B : Successeur0     Prédécesseur : 2 ? 1/0 :1
A : Successeur 1    Prédécesseur : 0 ? 1/0 :1
A : Successeur 1    Prédécesseur : 1 ? 1/0 :0
A : Successeur 1    Prédécesseur : 2 ? 1/0 :0
B : Successeur1     Prédécesseur : 0 ? 1/0 :0
B : Successeur1     Prédécesseur : 1 ? 1/0 :0
B : Successeur1     Prédécesseur : 2 ? 1/0 :1
A : Successeur 2    Prédécesseur : 0 ? 1/0 :0
A : Successeur 2    Prédécesseur : 1 ? 1/0 :0
A : Successeur 2    Prédécesseur : 2 ? 1/0 :0
B : Successeur2     Prédécesseur : 0 ? 1/0 :0
B : Successeur2     Prédécesseur : 1 ? 1/0 :0
B : Successeur2     Prédécesseur : 2 ? 1/0 :0
Saisir l'état finale entre 0 et 2 2

```

Le programme nous retourne le nouveau super-etat créé ainsi que ces successeurs en A et en B. Les autres états reste inchangés.

```

Nouveau Super Etat :
[0, 1]
Cet etat est initial
  Successeur en A :[0, 1]
  Successeur en A :[0, 1]
  Successeur en B :2

```

En effet, sur cet automate si les états 1 et 0 sont réunis, on obtient un automate équivalent mais avec moins d'états. De plus le programme nous indique si le nouveau super-état est initial et/ou final.

Si l'automate ne peut pas être minimisé alors le programme s'arrête et affiche un message.

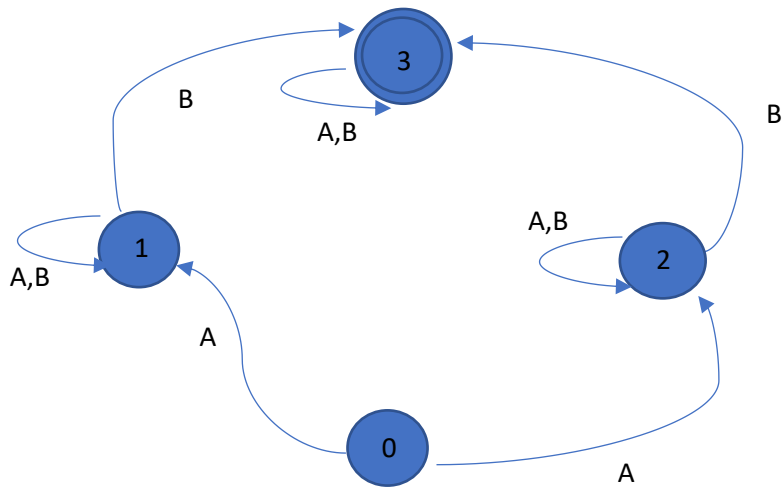
```

Saisir le nombre d'etat : 2
L'etat initiale doit etre 0
A : Successeur 0    Prédécesseur : 0 ? 1/0 :0
A : Successeur 0    Prédécesseur : 1 ? 1/0 :1
B : Successeur0     Prédécesseur : 0 ? 1/0 :1
B : Successeur0     Prédécesseur : 1 ? 1/0 :0
A : Successeur 1    Prédécesseur : 0 ? 1/0 :1
A : Successeur 1    Prédécesseur : 1 ? 1/0 :0
B : Successeur1     Prédécesseur : 0 ? 1/0 :0
B : Successeur1     Prédécesseur : 1 ? 1/0 :1
Saisir l'état finale entre 0 et 1 1
Impossible de plus minimiser cet automate
>>> |

```


Exemple 2

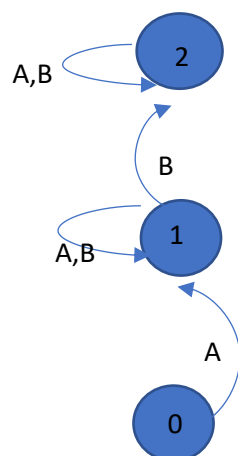
L'état 0 est l'état initial



Le programme nous retourne les lignes suivantes :

```
Nouveau Super Etat :  
[1, 2]  
Successeur en A : [1, 2]  
Successeur en A : [1, 2]  
Successeur en B : [1, 2]  
Successeur en B : 3
```

Les état 1 et 2 doivent être réunis. Voici l'automate équivalent minimisé :



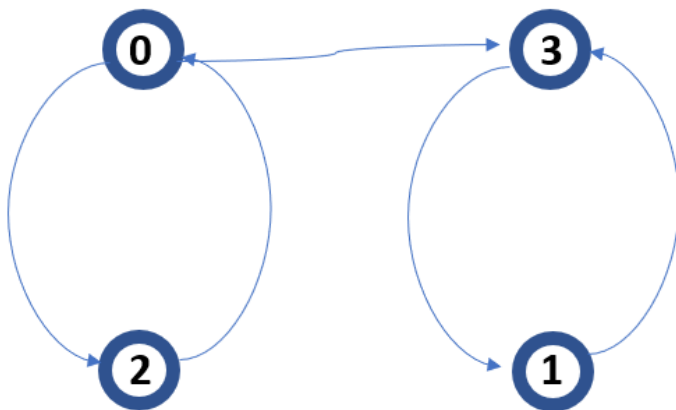
Compte rendu du projet de Graphe

Composantes fortement connexes

Rappel du principe de l'algorithme

Le but de cet algorithme est de déterminer les composantes fortement connexes. Pour tout couple d'une composante, le couple doit avoir un chemin de l'un vers l'autre et de l'autre vers l'un.

Pour trouver les composantes fortement connexes d'un graphe, on cherche, pour chaque successeur d'un état, si un de ses successeurs est ce même état. Si cette condition est réalisée alors les deux points forme un sous-graphe fortement connexe. Si ce successeur remplit cette condition, on fait de même qu'avec avec le premier état en ignorant le sommet déjà traité et ainsi de suite. Une fois ceci fait, on passe à autre point non déjà traité et on recommence jusqu'à ce que tous les sommets soient traités. En effet, un graphe peut comporter plusieurs sous graphes fortement connexes



Par exemple, ce graphe comporte deux composantes fortement connexes : Les états 0 et 2 d'une part et 1 et 3 d'autre part.

Structure utilisée et explications

L'utilisateur commence par saisir le nombre d'états du graphe et s'il est orienté ou non. Puis, il renseigne les différents liens du graphe. L'ensemble des liens possibles du graphe est enregistré dans un tableau 1D. Il est cependant très facile de retrouver un prédécesseur ou un successeur dans ces tableaux. La taille du tableau sera de taille $NbEtat^2$ ce qui permettra de visualiser tous les liens possibles entre les états.

Par exemple :

Voici un graphe orienté tel qu'il pourrait être enregistré :

TABA = [0,1,0,1,0,0,0,0]

Nous pouvons facilement obtenir le nombre d'états en faisant $TAILLE(TABA)_{1/2}$. Ici le premier 0 signifie qu'il n'existe pas de lien allant de 0 vers 0. Connaissant le nombre d'états de l'automate, nous savons donc que les nbEtat premières valeurs du tableau seront donc les liens ayant pour prédécesseur 0. Il est plus facile de visualiser ce tableau sous cette forme.

	0	1	2
0	0	1	0
1	1	0	0
2	0	0	0

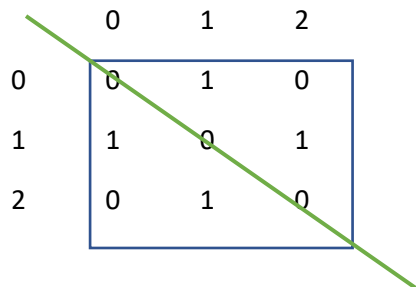
Le tableau est encadré en bleu et est disposé de façon plus compréhensible

Tous les 1 se trouvant dans le rectangle rouge nous permettent de savoir quels sont les **prédécesseurs** de 0. Tous les 1 se trouvant dans le rectangle vert nous permettent de savoir quels sont les **successeurs** de 0.

Dans le cas d'un graphe non orienté la structure est la même, il est demandé à l'utilisateur de saisir seulement la partie droite (encadré en rouge) par rapport à la diagonale du tableau :

	0	1	2
0	0	1	0
1	0	0	1
2	0	0	0

En effet, on considère qu'un lien non orienté de 1 vers 0 équivaut à un lien orienté de 0 à 1 et un autre de 1 à 0. Il suffit, par la suite, de remplir les autres cases par symétrie avec la diagonale.



	0	1	2
0	0	1	0
1	1	0	1
2	0	1	0

La méthode `getSuccesseur()` permet de récupérer une liste des successeurs de l'état passé en paramètre

```
def getSuccesseurs(numeroEtat):
    tab = []
    for i in range(nbEtat):
        #Pour acceder au predeceseur d'un etat il faut
        if (grapheInitial[nbEtat*numeroEtat+i] == 1):
            #recupere le num de l'etat
            tab.append(i)
    return tab
```

Méthode permettant de trouver une composante fortement connexe grâce à la récursivité :

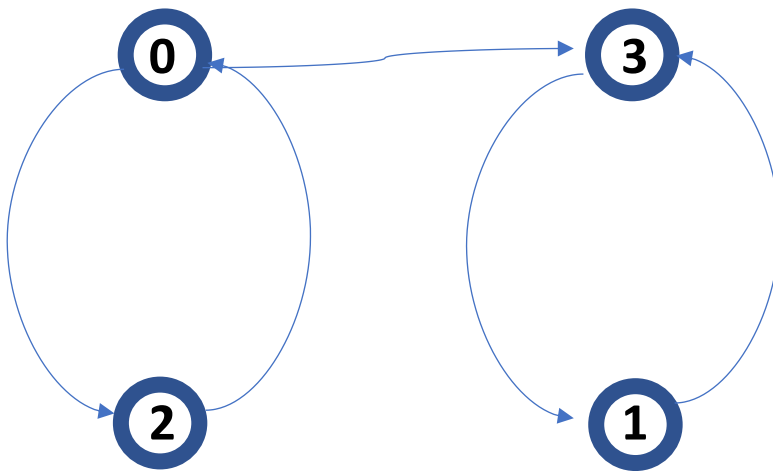
```
#Puisqu'on utilise la reecusrivite, la liste doit etre declarer en global
elementtraite = []
#Permet de trouver les etat fortement connexe avec l'etat passé en parametre
def findCompConnexe(elem):
    #l'element est traité
    elementtraite.append(elem)
    #Pour un sommet on cherche les successeur
    listeSuccesseur = getSuccesseurs(elem)
    #puis si un des successeur de elem a pour successeur elem alors il sont for
    for successeur in listeSuccesseur:
        if elem in getSuccesseurs(successeur) :
            #Alors les deux sommet sont connexes
            grapheConnexe.append(successeur)
            grapheConnexe.append(elem)
            #Cette condition evite la boucle infini
            #Si un etat a deja été traité alors on ne le retraite pas
            if successeur not in elementtraite:
                elementtraite.append(successeur)
                #Pour chaque etat fortement connexe à l'etat initiale on cher
                findCompConnexe(successeur)
```

Pour chaque élément non traité, c'est-à-dire ne faisant pas déjà partie d'une composante fortement connexe, on cherche s'il fait partie d'une composante fortement connexe grâce à la méthode ci-dessus. Pour chaque composante fortement connexe on l'affiche :

```
grapheConnexe = []
#pour chaque sommet du graphes
for y in range(nbEtat):
    #Si l'on a deja cherché les etats fortement connexes de l'
    if y not in elementtraite:
        findCompConnexe(y)
        if len(grapheConnexe) > 0:
            #On affiche la liste en suprimant les doublon
            print("Nouvelle composante fortement connexe : ")
            print(list(dict.fromkeys(grapheConnexe)))
            #on vide la liste jusqu'elle reset après
            grapheConnexe = []
```

Cas d'usages

Voici un graphe orienté pour lequel on cherche ses composantes fortement connexes :



Voici l'entrée du programme (1 signifie que le lien existe et 0 qu'il n'existe pas) :

```

Saisir le nombre d'etat : 4
Le graphe est il orienté ? : o/n o
Graphe : Successeur 0    Prédécesseur : 0 ? 1/0 :0
Graphe : Successeur 0    Prédécesseur : 1 ? 1/0 :0
Graphe : Successeur 0    Prédécesseur : 2 ? 1/0 :1
Graphe : Successeur 0    Prédécesseur : 3 ? 1/0 :1
Graphe : Successeur 1    Prédécesseur : 0 ? 1/0 :0
Graphe : Successeur 1    Prédécesseur : 1 ? 1/0 :0
Graphe : Successeur 1    Prédécesseur : 2 ? 1/0 :0
Graphe : Successeur 1    Prédécesseur : 3 ? 1/0 :1
Graphe : Successeur 2    Prédécesseur : 0 ? 1/0 :1
Graphe : Successeur 2    Prédécesseur : 1 ? 1/0 :0
Graphe : Successeur 2    Prédécesseur : 2 ? 1/0 :0
Graphe : Successeur 2    Prédécesseur : 3 ? 1/0 :0
Graphe : Successeur 3    Prédécesseur : 0 ? 1/0 :0
Graphe : Successeur 3    Prédécesseur : 1 ? 1/0 :1
Graphe : Successeur 3    Prédécesseur : 2 ? 1/0 :0
Graphe : Successeur 3    Prédécesseur : 3 ? 1/0 :0

```

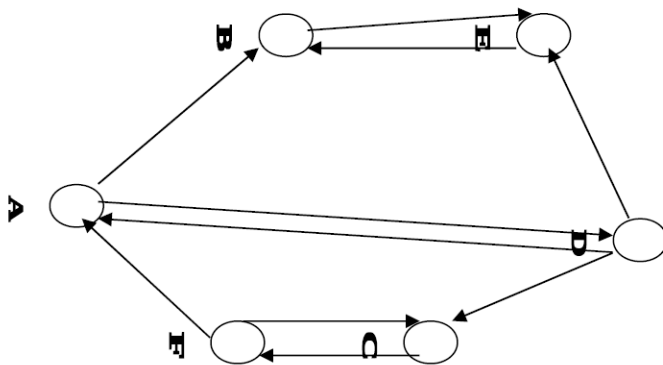
Et voici la sortie du programme. Celui-ci trouve deux composantes fortement connexes :

```

Nouvelle composante fortement connexe :
[2, 0]
Nouvelle composante fortement connexe :
[3, 1]

```

Voici un autre graphe vu en TD :



Voici les informations du graphe saisie :

```
Saisir le nombre d'etat : 6
Le graphe est il orienté ? : o/n o
Graphe : Successeur 0    Prédécesseur : 0 ? 1/0 :0
Graphe : Successeur 0    Prédécesseur : 1 ? 1/0 :1
Graphe : Successeur 0    Prédécesseur : 2 ? 1/0 :0
Graphe : Successeur 0    Prédécesseur : 3 ? 1/0 :1
Graphe : Successeur 0    Prédécesseur : 4 ? 1/0 :0
Graphe : Successeur 0    Prédécesseur : 5 ? 1/0 :0
Graphe : Successeur 1    Prédécesseur : 0 ? 1/0 :0
Graphe : Successeur 1    Prédécesseur : 1 ? 1/0 :0
Graphe : Successeur 1    Prédécesseur : 2 ? 1/0 :0
Graphe : Successeur 1    Prédécesseur : 3 ? 1/0 :0
Graphe : Successeur 1    Prédécesseur : 4 ? 1/0 :1
Graphe : Successeur 1    Prédécesseur : 5 ? 1/0 :0
Graphe : Successeur 2    Prédécesseur : 0 ? 1/0 :0
Graphe : Successeur 2    Prédécesseur : 1 ? 1/0 :0
Graphe : Successeur 2    Prédécesseur : 2 ? 1/0 :0
Graphe : Successeur 2    Prédécesseur : 3 ? 1/0 :0
Graphe : Successeur 2    Prédécesseur : 4 ? 1/0 :0
Graphe : Successeur 2    Prédécesseur : 5 ? 1/0 :1
Graphe : Successeur 3    Prédécesseur : 0 ? 1/0 :1
Graphe : Successeur 3    Prédécesseur : 1 ? 1/0 :0
Graphe : Successeur 3    Prédécesseur : 2 ? 1/0 :1
Graphe : Successeur 3    Prédécesseur : 3 ? 1/0 :0
Graphe : Successeur 3    Prédécesseur : 4 ? 1/0 :1
Graphe : Successeur 3    Prédécesseur : 5 ? 1/0 :0
Graphe : Successeur 4    Prédécesseur : 0 ? 1/0 :0
Graphe : Successeur 4    Prédécesseur : 1 ? 1/0 :1
Graphe : Successeur 4    Prédécesseur : 2 ? 1/0 :0
Graphe : Successeur 4    Prédécesseur : 3 ? 1/0 :0
Graphe : Successeur 4    Prédécesseur : 4 ? 1/0 :0
Graphe : Successeur 4    Prédécesseur : 5 ? 1/0 :0
Graphe : Successeur 5    Prédécesseur : 0 ? 1/0 :1
Graphe : Successeur 5    Prédécesseur : 1 ? 1/0 :0
Graphe : Successeur 5    Prédécesseur : 2 ? 1/0 :1
Graphe : Successeur 5    Prédécesseur : 3 ? 1/0 :0
Graphe : Successeur 5    Prédécesseur : 4 ? 1/0 :0
Graphe : Successeur 5    Prédécesseur : 5 ? 1/0 :0
Nouvelle composante fortement connexe :
```

Et voici les trois composantes fortement connexes trouvé :

```
Nouvelle composante fortement connexe :
[3, 0]
Nouvelle composante fortement connexe :
[4, 1]
Nouvelle composante fortement connexe :
[5, 2]
```

On peut facilement vérifier la véracité des composantes fortement connexes trouvées par le programme en regardant le graphe initial

Plus court chemin

Rappel du principe de l'algorithme

L'algorithme de Dijkstra permet d'obtenir le plus court chemin possible entre deux états dans un graphe pondéré sans lien de poids négatif. L'algorithme de Dijkstra consiste en la construction d'un tableau permettant de savoir la distance minimale des états par rapport à un état de départ.

On cherche, tout d'abord, le plus court chemin reliant l'état initial à un autre sommet. Ensuite, pour ce sommet, on fait de même en ajoutant le poids de l'arête entre cet état et l'état initial, au poids des états menant aux successeurs. On construit de proche en proche le chemin cherché en choisissant à chaque itération de l'algorithme, un sommet parmi ceux qui n'ont pas encore été traité, tel que la longueur connue provisoirement du plus court chemin allant de l'état initial à l'état final soit la plus courte possible.

Structure utilisée et explications

Pour l'algorithme de Dijkstra, l'utilisateur doit renseigner un graphe non orienté, j'ai donc réutilisé la même structure de données que pour l'exercice précédent. Cette fois-ci, l'utilisateur n'a la possibilité de choisir un graphe non orienté seulement. Le graphe est par la suite converti en graphe orienté grâce à la méthode vue dans l'exercice précédent. Cependant, l'utilisateur doit cette fois-ci, renseigner le poids du lien entre les deux états. S'il n'y a pas de lien entre deux états proposés alors l'utilisateur doit saisir -1.

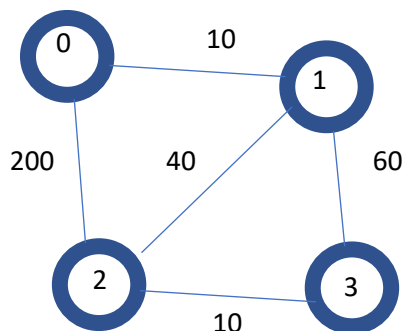
But du traitement

```
#Traitement

#On initialise le tableau avec -1 dans chaque case, ce tableau permet d'obtenir la distance m
#Si la distance est -1 alors la distance minimale est inconnue
listeDistDepart = [ -1 for j in range(nbEtat*nbEtat)]
#On recupere la liste des successeur du point de depart
listSuccesseur = getSuccesseurs(depart)
etatVisite = []
#On ajoute l'etat de depart à la liste des etats visites
etatVisite.append(depart)
#On ajoute dans le tableau les distances du point de départ a chaque successeur
for i in range(0,len(listSuccesseur)):
    listeDistDepart[(nbEtat*depart)+listSuccesseur[i][0]] = listSuccesseur[i][1]
#On cherche le plus petit chemin
plusPetitChemin = 100000 ;
for i in listeDistDepart:
    if i < plusPetitChemin and i > 0:
        plusPetitChemin = i
#On recupere le point le plus proche grace a la distance la plus proche
pointplusproche = (listeDistDepart.index(plusPetitChemin)) %nbEtat
etatVisite.append(pointplusproche)

#Puis tant que l'etat d'arrive n'a pas ete visité on répete le code precedent
while arrivee not in etatVisite :
    #On cherche le point le plus proche
    etatVisite.append(pointplusproche)
    #On recupere les successeur du poin
    tabSuccesseurs = getSuccesseurs(pointplusproche)
    #Pour chacun des successeurs du point on rajoute le poids du point de depart a ce point
    for i in range(0,len(tabSuccesseurs)):
        tabSuccesseurs[i][1] = tabSuccesseurs[i][1]+plusPetitChemin
    for i in range(0,len(tabSuccesseurs)):
        listeDistDepart[(nbEtat*pointplusproche)+tabSuccesseurs[i][0]] = tabSuccesseurs[i][1]
    #Condition de sortie des que l'état d'arrive est traité
    print(etatVisite)
    if arrivee in etatVisite :
        break
    #On cherche le plus petit chemin
    plusPetitChemin = 100000
    for i in range(0,nbEtat*nbEtat):
        if (i%nbEtat) not in etatVisite:
            if listeDistDepart[i] < plusPetitChemin and listeDistDepart[i] > 0:
                plusPetitChemin = listeDistDepart[i]
                pointplusproche = i%nbEtat
```

Le but du code ci-dessus est convertir le tableau initial (nommé `grapheInitial` dans le code) représentant le poids de chaque lien en un tableau indiquant le poids du chemin minimum pour chaque état depuis l'état de départ choisi (nommé `listeDistDepart` dans le code).



	0	1	2	3
0	-1	10	200	-1
1	10	-1	40	60
2	200	40	-1	10
3	-1	60	10	-1

Par exemple si l'état de départ choisi est 1, alors on obtiendra ce tableau :

	0	1	2	3
0	-1	20	210	-1
1	10	-1	40	60
2	240	80	-1	50
3	-1	140	90	-1

Pour obtenir le plus court chemin d'un état n jusqu'au point de départ, il suffit de regarder la plus petite valeur de la colonne n. Par exemple pour obtenir le plus court chemin de 1 jusqu'à l'état trois il suffit de regarder dans la colonne 3 (4eme colonne (en orange)) la plus petite valeur (ici 50). Une fois qu'on a ce tableau, il est donc très facile de trouver pour un état le plus court chemin jusqu'à l'état initial.

Affichage du plus court chemin :

```
#On affiche le poinds du plus petit chemin de depart a arrivee
print("Le plus petit chemin est de taille : "+str(plusPetitChemin))
print("Chemin le plus court : ")
plusPetitChemin = 10000
#De arrive a depart on affiche le chemin le plus court
print("Etat :"+str(arrivee))
while arrivee != depart:
    for i in range (0,nbEtat):
        if listeDistDepart[arrivee+(nbEtat*i)] < plusPetitChemin and listeDistDepart[arrivee+(nbEtat*i)] > 0:
            plusPetitChemin = listeDistDepart[arrivee+(nbEtat*i)]
            tmp = i
    arrivee = tmp
    print(",")
    print("Etat :"+str(arrivee))
```

Pour afficher le plus court chemin, on parcourt tout le chemin de l'arrivé au départ dans le sens inverse.

1 est le point de départ

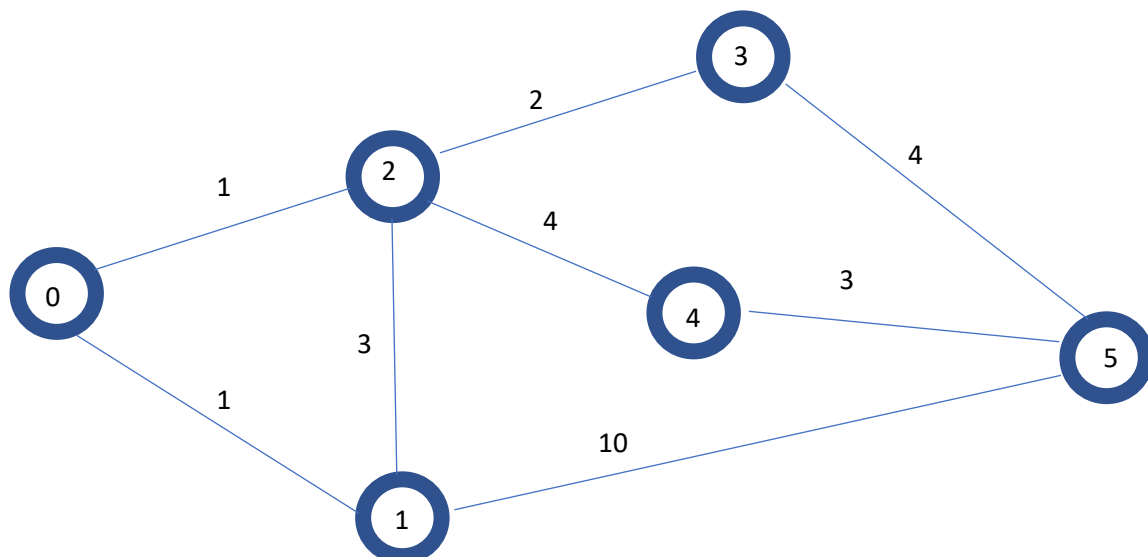
3 est le point d'arrivé

	0	1	2	3
0	-1	20	210	-1
1	10	-1	40	60
2	240	80	-1	50
3	-1	140	90	-1

Le chemin passe par 2 puisque la plus petite valeur dans la colonne numéro 3 se trouve à la ligne numéro 2. On doit donc, par la suite, chercher dans la colonne numéro 2 la plus petite valeur, qui est 40. 40 se trouve à la ligne numéro 1 et la ligne numéro 1 est l'état de départ, on peut donc s'arrêter. Le plus court chemin est donc 3-2-1.

Cas d'usages

Cherchons le plus court chemin entre le sommet 1 et le sommet 6 :



Voici le graphe tel qu'il est renseigné en entrée du programme

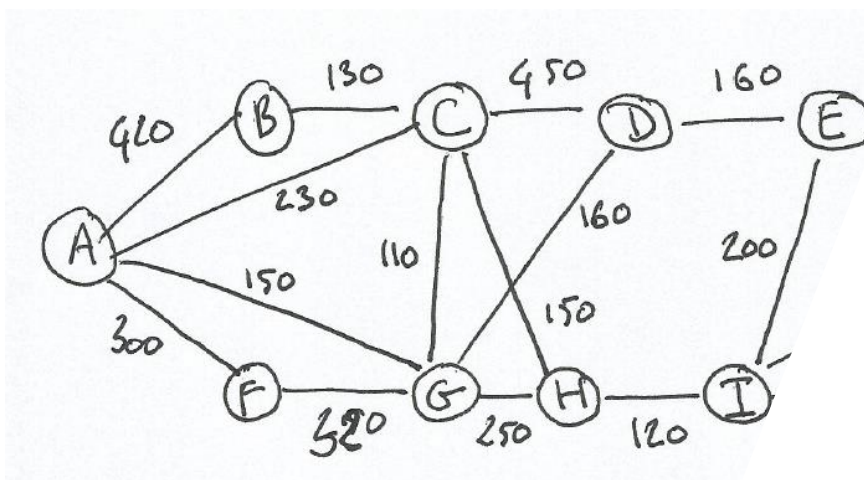
```
Saisir le nombre d'etat : 6
Sommet de départ (saisir val entre 0 et 5 ) : 0
Sommet d'arrivee(saisir val entre 0 et 5 ) : 5
Poids du lien entre 0 et : 0 ? Si le lien n'existe pas alors saisir -1 :-1
Poids du lien entre 0 et : 1 ? Si le lien n'existe pas alors saisir -1 :1
Poids du lien entre 0 et : 2 ? Si le lien n'existe pas alors saisir -1 :1
Poids du lien entre 0 et : 3 ? Si le lien n'existe pas alors saisir -1 :-1
Poids du lien entre 0 et : 4 ? Si le lien n'existe pas alors saisir -1 :-1
Poids du lien entre 0 et : 5 ? Si le lien n'existe pas alors saisir -1 :-1
Poids du lien entre 1 et : 1 ? Si le lien n'existe pas alors saisir -1 :-1
Poids du lien entre 1 et : 2 ? Si le lien n'existe pas alors saisir -1 :3
Poids du lien entre 1 et : 3 ? Si le lien n'existe pas alors saisir -1 :-1
Poids du lien entre 1 et : 4 ? Si le lien n'existe pas alors saisir -1 :-1
Poids du lien entre 1 et : 5 ? Si le lien n'existe pas alors saisir -1 :10
Poids du lien entre 2 et : 2 ? Si le lien n'existe pas alors saisir -1 :-1
Poids du lien entre 2 et : 3 ? Si le lien n'existe pas alors saisir -1 :2
Poids du lien entre 2 et : 4 ? Si le lien n'existe pas alors saisir -1 :4
Poids du lien entre 2 et : 5 ? Si le lien n'existe pas alors saisir -1 :-1
Poids du lien entre 3 et : 3 ? Si le lien n'existe pas alors saisir -1 :-1
Poids du lien entre 3 et : 4 ? Si le lien n'existe pas alors saisir -1 :-1
Poids du lien entre 3 et : 5 ? Si le lien n'existe pas alors saisir -1 :4
Poids du lien entre 4 et : 4 ? Si le lien n'existe pas alors saisir -1 :-1
Poids du lien entre 4 et : 5 ? Si le lien n'existe pas alors saisir -1 :3
Poids du lien entre 5 et : 5 ? Si le lien n'existe pas alors saisir -1 :-1
```

Voici la sortie du programme :

```
Le plus petit chemin est de taille : 7
Chemin le plus court :
Etat : 5
↓
Etat : 3
↓
Etat : 2
↓
Etat : 0
```

Le programme affiche la taille du plus court chemin ainsi que son chemin. En effet, on peut facilement voir grâce au schéma du graphe que le chemin le plus court est bien de taille 7 et le chemin est bien 5->3->2->0.

Voici un autre graphe, vu en TD. On cherche le plus court chemin de A à I :



On saisit donc le nombre d'état, ainsi que les deux sommets dont on veut trouver le plus court chemin.

```
Saisir le nombre d'etat : 9
Sommet de départ (saisir val entre 0 et 8 ) : 0
Sommet d'arrivee(saisir val entre 0 et 8 ) : 8
Poids du lien entre 0 et : 0 ? Si le lien n'existe pas alors saisir -1 :-1
Poids du lien entre 0 et : 1 2 Si le lien n'existe pas alors saisir -1 :420
```

Le programme nous retourne le chemin minimum de A à I :

L'état 8 étant I, l'état 7 étant H, l'état 2 étant C et l'état 0 étant A. En effet, le chemin I-H-C-A est bien le chemin le plus court et a bien un poids de 500

```
Le plus petit chemin est de taille : 500
Chemin le plus court :
Etat :8
↓
Etat :7
↓
Etat :2
↓
Etat :0
```

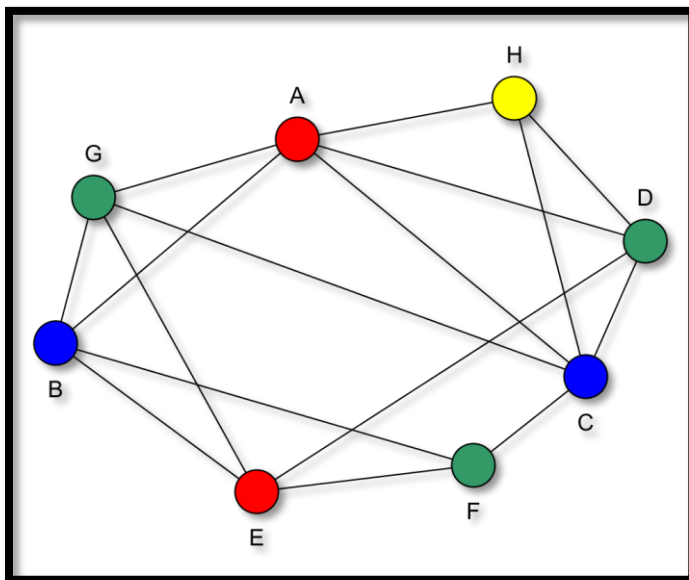
Algorithme de coloration

Rappel du principe de l'algorithme

L'algorithme de coloration consiste à attribuer différentes couleurs à chaque état du graphe de façon à ce que deux états ayant un lien, de l'un vers l'autre, ne soit pas de la même couleur.

Tout d'abord, les états doivent être triés selon le nombre d'arête(degré) dans l'ordre décroissant. On attribue une couleur au premier état de la liste puis la même couleur au premier état dans la liste qui est non adjacent au premier. On colore ensuite avec la même couleur, le premier sommet non adjacent aux deux premiers sommets coloriés et ainsi de suite jusqu'à ce qu'on ne trouve plus de sommet adjacent. On change de couleur puis on répète l'opération avec le premier état de liste non traité. On continue jusqu'à ce que tous les sommets soient coloriés.

Voici un exemple de coloration de graphe :



Structure utilisée et explications

Pour la structure de données, j'ai utilisé la même méthode que pour les exercices précédents.

```
Saisir le nombre d'etat : 5
Graphe : Existe t il un lien entre 0 et : 0 ? 1/0 :0
Graphe : Existe t il un lien entre 0 et : 1 ? 1/0 :1
Graphe : Existe t il un lien entre 0 et : 2 ? 1/0 :1
Graphe : Existe t il un lien entre 0 et : 3 ? 1/0 :0
Graphe : Existe t il un lien entre 0 et : 4 ? 1/0 :0
Graphe : Existe t il un lien entre 1 et : 1 ? 1/0 :0
Graphe : Existe t il un lien entre 1 et : 2 ? 1/0 :1
Graphe : Existe t il un lien entre 1 et : 3 ? 1/0 :1
Graphe : Existe t il un lien entre 1 et : 4 ? 1/0 :00
Graphe : Existe t il un lien entre 2 et : 2 ? 1/0 :0
Graphe : Existe t il un lien entre 2 et : 3 ? 1/0 :0
Graphe : Existe t il un lien entre 2 et : 4 ? 1/0 :0
Graphe : Existe t il un lien entre 3 et : 3 ? 1/0 :0
Graphe : Existe t il un lien entre 3 et : 4 ? 1/0 :1
Graphe : Existe t il un lien entre 4 et : 4 ? 1/0 :0
[[1, 'Couleur1'], [4, 'Couleur1'], [0, 'Couleur2'], [3, 'Couleur2'], [2, 'Couleur3']]
```

```

Saisir le nombre d'etat : 5
Graphe : Existe t il un lien entre 0 et : 0 ? 1/0 :0
Graphe : Existe t il un lien entre 0 et : 1 ? 1/0 :1
Graphe : Existe t il un lien entre 0 et : 2 ? 1/0 :0
Graphe : Existe t il un lien entre 0 et : 3 ? 1/0 :0
Graphe : Existe t il un lien entre 0 et : 4 ? 1/0 :0
Graphe : Existe t il un lien entre 1 et : 1 ? 1/0 :0
Graphe : Existe t il un lien entre 1 et : 2 ? 1/0 :1
Graphe : Existe t il un lien entre 1 et : 3 ? 1/0 :0
Graphe : Existe t il un lien entre 1 et : 4 ? 1/0 :0
Graphe : Existe t il un lien entre 2 et : 2 ? 1/0 :0
Graphe : Existe t il un lien entre 2 et : 3 ? 1/0 :1
Graphe : Existe t il un lien entre 2 et : 4 ? 1/0 :0
Graphe : Existe t il un lien entre 3 et : 3 ? 1/0 :0
Graphe : Existe t il un lien entre 3 et : 4 ? 1/0 :1
Graphe : Existe t il un lien entre 4 et : 4 ? 1/0 :0
[[1, 'Couleur1'], [3, 'Couleur1'], [2, 'Couleur2'], [0, 'Couleur2'], [4, 'Couleur2']]

```

```

tabEtatTrie = []
for i in range(0,nbEtat):
    tabEtatTrie.append([i,len(getSuccesseurs(i))])

#Trie du tableau selon le second element de chaque case qui coorespon au nombre de successeur
def takeSecond(elem):
    return elem[1]
tabEtatTrie.sort(key=takeSecond,reverse=True)
couleur = 1
tabCouleur = []
compteur = 0
while len(tabCouleur) < nbEtat:
    tabCouleur.append([tabEtatTrie[compteur][0],"Couleur"+str(couleur)])
    tmp = []
    for i in range(0,nbEtat):

        ok = True
        #Si l'element courant est adjacent a l'un des sommet possedant la couleur courante alors on ne realise pas la
        #condition suivantes(les element possedant la couleur courante se trouve dans tmp)
        for z in tmp:
            if i in getSuccesseurs(z):
                ok = False
        #Si i n'est pas adjacent au premier sommet non traite de la liste
        if i not in getSuccesseurs(tabEtatTrie[compteur][0]) and i != tabEtatTrie[compteur][0] and dejaTraite(i)==False and ok == True:
            tabCouleur.append([i,"Couleur"+str(couleur)])
            tmp.append(i)
            ok = True
        if i == nbEtat-1:
            break
    #On change de couleur pour passer a un nouveau groupe d'etats
    couleur = couleur + 1
    compteur = compteur +1

print(tabCouleur)

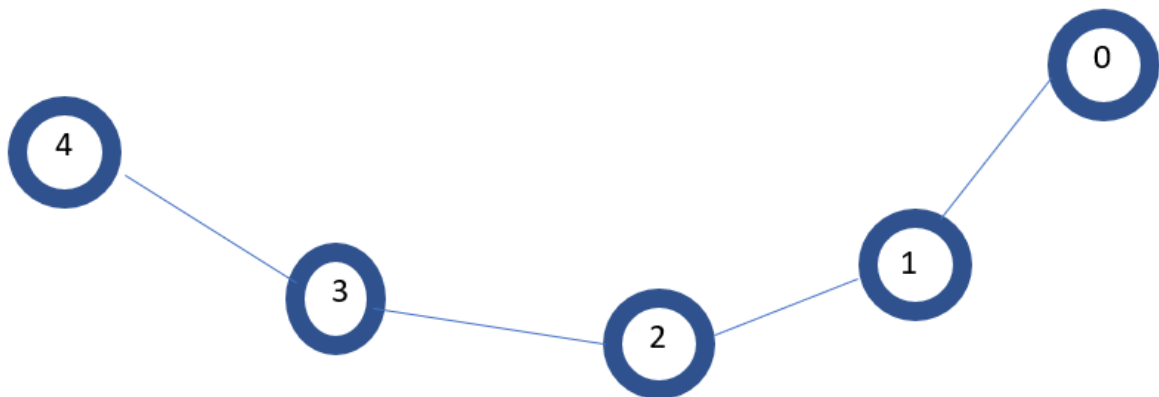
```

Tout d’abord, on crée un tableau. Chaque case du tableau est composée de son numéro ainsi que de son degré (on obtient le degré d’un état en comptant ses successeurs). Puis, on trie le tableau selon le degré de chaque état.

Pour chaque élément en privilégiant les degrés les plus faibles, on ajoute cet élément à un nouveau groupe, puis on cherche le premier sommet non adjacent dans la liste. On ajoute ce sommet à la liste. On cherche un autre état non adjacent à l’état se trouvant dans la liste et ainsi de suite jusqu’à ce que ce ne soit plus possible. Enfin on change de couleur et on ajoute un nouvel élément non traité à la liste. De même que précédemment on cherche les états qui ne sont pas adjacents. Une fois que l’on ne trouve plus de sommet non adjacent à tous ceux se trouvant dans la liste, on change de couleur et on recommence jusqu’à ce que tous les éléments soient traités.

Cas d'usages

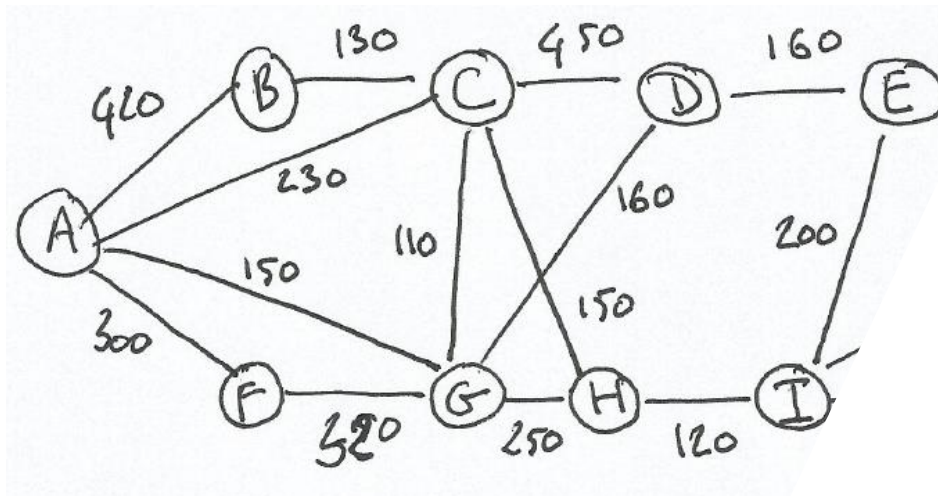
Voici un graphe à 5 états, 2 couleurs sont suffisantes pour colorer ce graphe. 0,2,4 dans une couleur et 1 et 3 dans une autre couleur.



```
Saisir le nombre d'etat : 5
Graphe : Existe t il un lien entre 0 et : 0 ? 1/0 :0
Graphe : Existe t il un lien entre 0 et : 1 ? 1/0 :1
Graphe : Existe t il un lien entre 0 et : 2 ? 1/0 :0
Graphe : Existe t il un lien entre 0 et : 3 ? 1/0 :0
Graphe : Existe t il un lien entre 0 et : 4 ? 1/0 :0
Graphe : Existe t il un lien entre 1 et : 1 ? 1/0 :0
Graphe : Existe t il un lien entre 1 et : 2 ? 1/0 :1
Graphe : Existe t il un lien entre 1 et : 3 ? 1/0 :0
Graphe : Existe t il un lien entre 1 et : 4 ? 1/0 :0
Graphe : Existe t il un lien entre 2 et : 2 ? 1/0 :0
Graphe : Existe t il un lien entre 2 et : 3 ? 1/0 :1
Graphe : Existe t il un lien entre 2 et : 4 ? 1/0 :0
Graphe : Existe t il un lien entre 3 et : 3 ? 1/0 :0
Graphe : Existe t il un lien entre 3 et : 4 ? 1/0 :1
Graphe : Existe t il un lien entre 4 et : 4 ? 1/0 :0
[[1, 'Couleur1'], [3, 'Couleur1'], [2, 'Couleur2'], [0, 'Couleur2'], [4, 'Couleur2']]
```

Le programme nous renvoi, pour chaque état, sa couleur associée

Cas d'un graphe vu en TD



Voici le résultat :

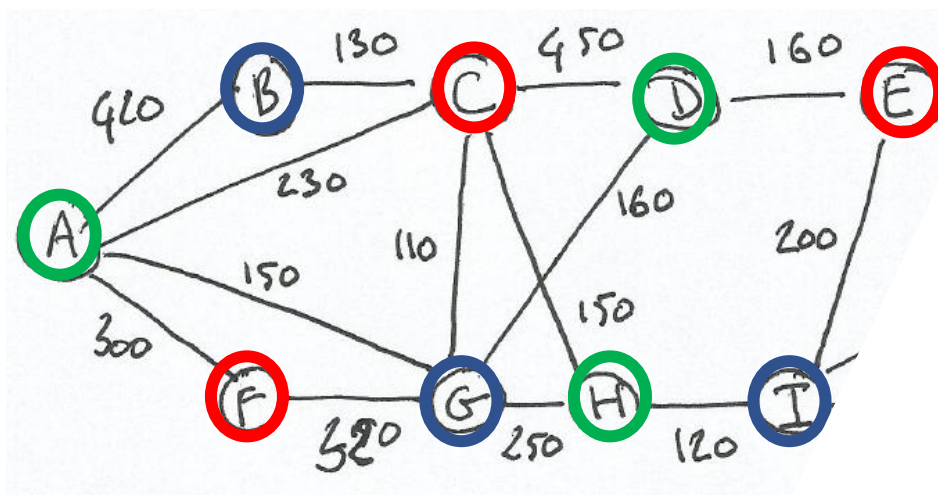
```
[[2, 'Couleur1'], [4, 'Couleur1'], [5, 'Couleur1'], [6, 'Couleur2'],  
1, 'Couleur2'], [8, 'Couleur2'], [0, 'Couleur3'], [3, 'Couleur3'], [7, 'Couleur3']]
```

Attention A correspond à 0

Couleur1 : 2,4,5 (C, E, F)

Couleur2 : 6,1,8 (G, B, I)

Couleur3 : 0,3,7 (A, D, H)



Le résultat retourné par le programme est correct, aucune paire d'état aillant la même couleur n'est adjacente. De plus, 3 couleurs ont été utilisé pour colorer le graphe. C'est bien le nombre minimal de couleur pour colorer le graphe.