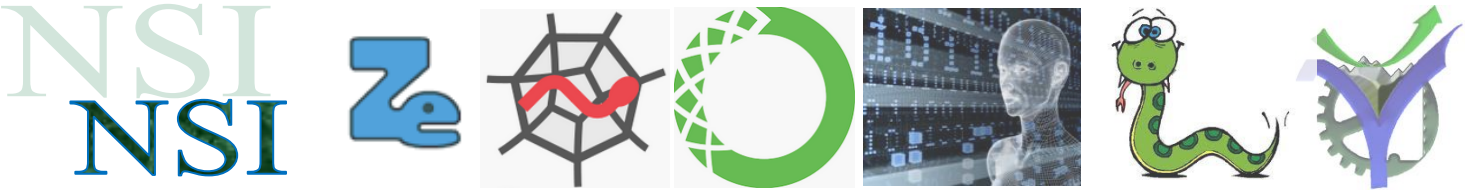


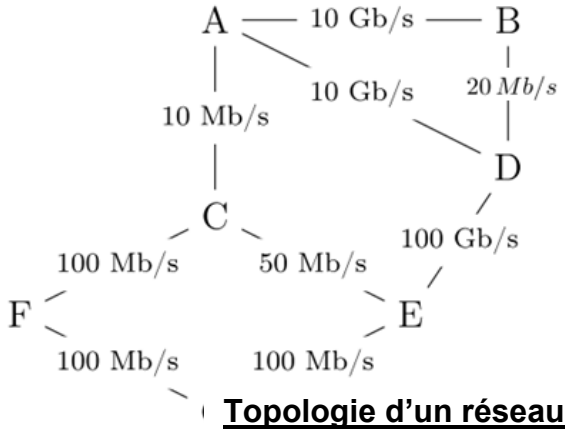
Les graphes

Résumé

Les graphes permettent de caractériser une représentation d'une solution à un grand nombre de problèmes de la vie courante.

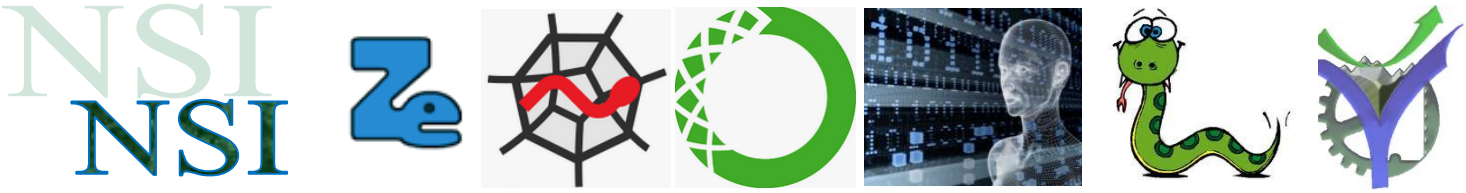
Savoir utiliser un graphe pour en déduire l'existence de relations entre deux entités, l'existence d'un chemin, le coût de transit comparatif entre deux chemins est alors possible.

Il existe un très grand nombre d'applications industrielles tel que : le routage d'informations dans les réseaux informatique, l'organisation des tâches dans un atelier de production.



Topologie d'un réseau

Sommaire



1 Les graphes quesaco ?

1.1 Présentation

Les graphes sont partout dans notre vie quotidienne, quelques exemples :

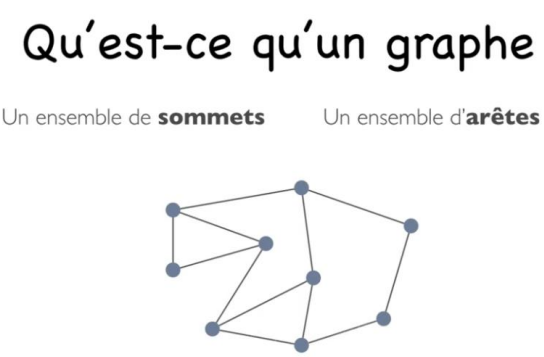
1

Réseau autoroutier

2

Le réseau d'amis de Facebook

1.2 Notions de base autour des graphes



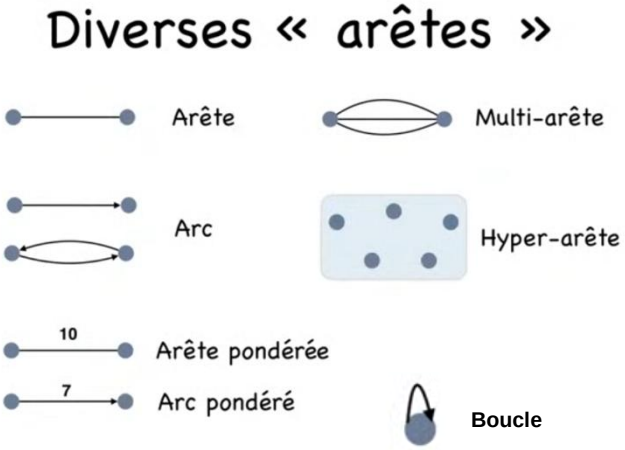
La vidéo ci-dessous introduit les notions de bases sur les graphes.

1.3 Divers types de graphes

Graphe orienté, non orienté, pondéré avec boucles …. tout est là :

https://www.youtube.com/watch?v=O1YdAP-tF2U&list=RDCMUCHtJVeNLyR1yuJ1_xCK1WRg&index=12

https://www.youtube.com/watch?v=YYv2R1cCTa0



[1] https://static.wikia.nocookie.net/routes/images/1/10/Carte_autoroutes_Concessions.jpg/revision/latest?cb=20101115200414

[2] https://www.blogdumoderateur.com/analyser-son-profil-facebook-avec-wolfram-alpha/

1.4 Vocabulaire :

Sommets ou Nœuds

Arêtes, Arcs : graphe non orienté, graphe orienté

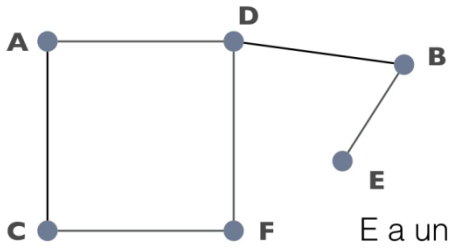
Graphe complet : il existe toutes les arêtes possibles entre tous ses sommets.

Sommets voisins : sommets reliés par une arête.

Degré d'un sommet : nombre de ses voisins.

Voisins de A : D et C

Degré de D = 3



E a un seul voisin : B

Degré de E = 1

E et F ne sont **pas** voisins

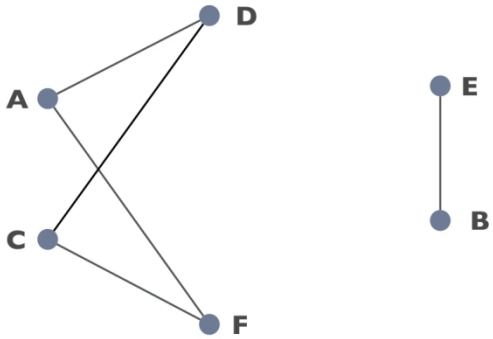
Chemin : succession d'arêtes permettant de relier deux sommets.

Longueur de chemin : le nombre d'arêtes.

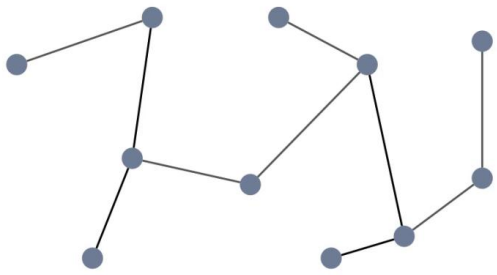
Cycle : un chemin qui part et arrive depuis le même nœud.

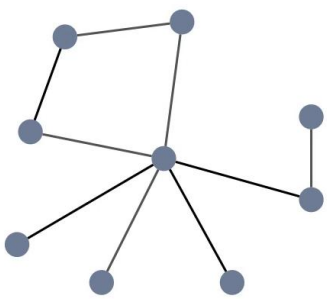
Graphe connexe : toute paire de sommets du graphe peut être reliée par un chemin.

Composante connexe : graphe en plusieurs parties non reliées ensembles.



Arbre : graphe connexe et sans cycle.





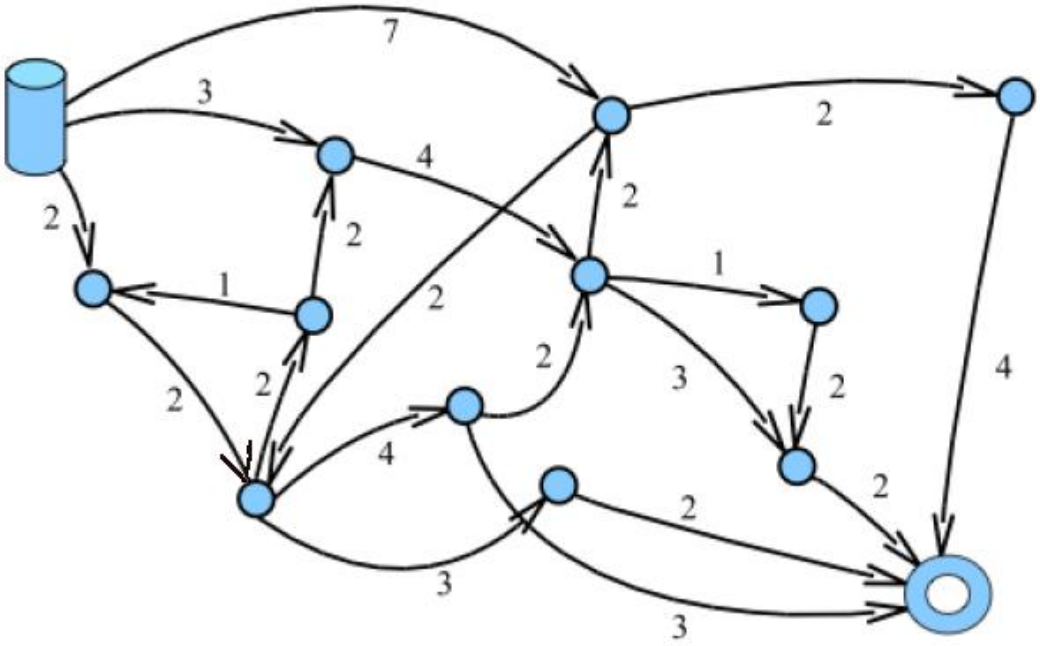
Propriété : somme des degrés = 2 • nombre de sommets

1.5 Quelques questions

Acheminement de pétrole[3]

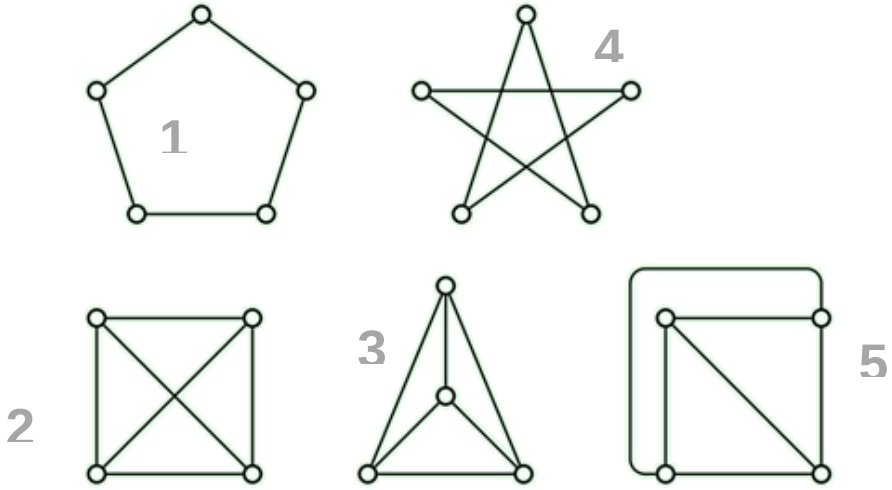
La compagnie pétrolière Inti T'schouff souhaite acheminer du pétrole par oléoduc vers un pays client. Le réseau d'oléoduc comporte plusieurs tronçons, chacun ayant une capacité maximale (en débit) à ne pas dépasser. Les tronçons sont directionnels. Sur le graphe suivant, la compagnie pétrolière est représentée par le cylindre, le client par le jeton. La capacité maximale de chaque arc est indiquée.

Q1. Quel est le débit maximum que la compagnie pétrolière peut envoyer vers le client via le réseau?



Quels sont les graphes identiques[4]

Q2. Déterminer les graphes identiques.



[3] Exercices de graphes, Nadia Brauner, UGA.
[4] Ibid.

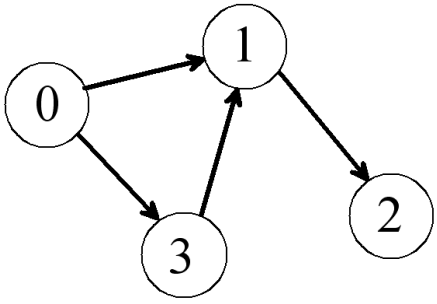
Tracé de graphe

Q3. Construire un graphe orienté dont les sommets sont les entiers compris entre 1 et 12 et dont les arcs représentent la relation « être diviseur de »[5].

2 Représentation en python[6]

2.1 Représentation par matrice d'adjacence

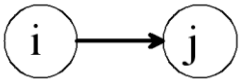
Graphes orientés



Voilà un graphe orienté :

Pour le représenter informatiquement on peut utiliser un tableau de booléens de dimension N x N ou N est le nombre de nœuds ou sommets du graphe. Chaque valeur de ce tableau représente l'existence d'un arc orienté entre le nœud i et le nœud j :

- adjacence[i][j] = True il existe un arc entre les nœuds i et j

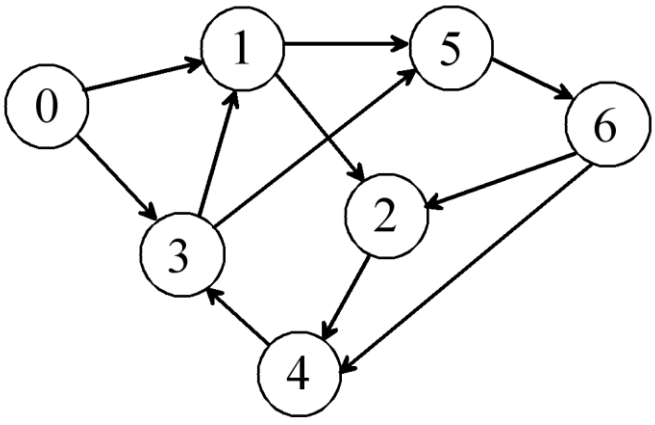


- adjacence[i][j] = False il existe aucun arc entre les nœuds i et j

Matrice d'adjacence du graphe ci-dessus ⇨

	0	1	2	3
0	F	T	F	T
1	F	F	T	F
2	F	F	F	F
3	F	T	F	F

Q4. Donner la matrice d'adjacence du graphe ci-dessous :



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

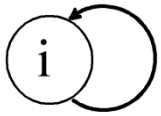
⇦

[5] D'après exo_graphes_sopena_tout.pdf Éric SOPENA - sopena@labri.fr

[6] Les scripts de cette section sont inspirés du livre 24 leçons et exercices corrigés TNSI ellipses.

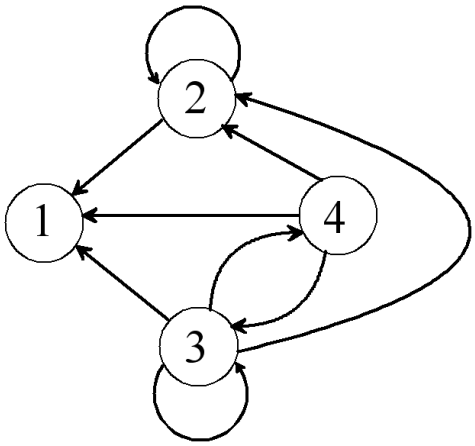
Q5. Cas des graphes non orientés : indiquez comment est renseignée dans ce cas la matrice d'adjacence.

Q6. Cas des boucles : indiquez comment est renseignée dans ce cas la matrice d'adjacence.



Q7. Donner la matrice d'adjacence du graphe ci-contre puis répondre aux questions :

	1	2	3	4
1				
2				
3				
4				



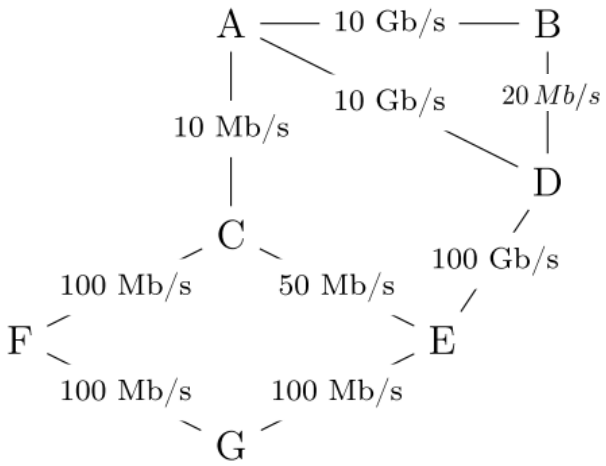
a) Comment peut-on identifier un nœud puits, c'est-à-dire un nœud d'où aucun arc ne part vers un autre nœud dans la matrice d'adjacence ?

b) Comment identifier un nœud qui est relié à tous les autres dans la matrice d'adjacence ?

Graphes pondérés

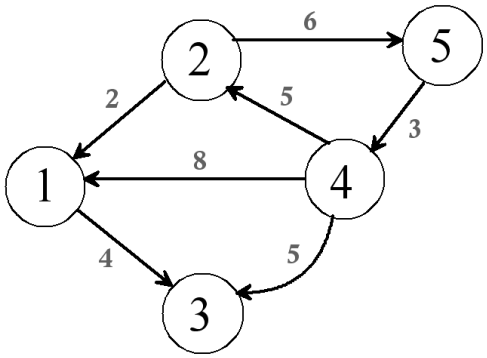
Si l'on veut indiquer un coût de transit entre deux nœuds reliés par un arc, par exemple un temps de transit, un coût en carburant, une capacité en débit d'une liaison alors on indique la valeur à coté de l'arc.

Voir l'illustration[7] ci-contre d'un réseau numérique interconnectant plusieurs routeurs avec les débits correspondants.



[7] Issue du sujet0 NSI

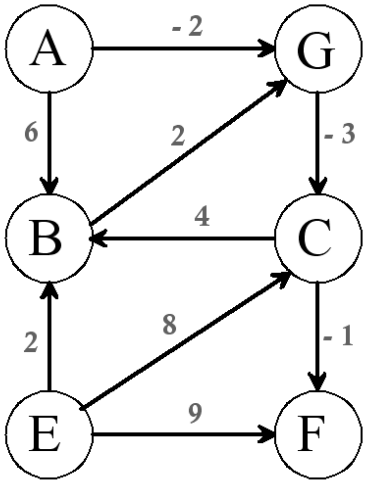
Représentation avec une matrice d'adjacence :



	1	2	3	4	5
1	∞	∞	4	∞	∞
2	2	∞	∞	∞	6
3	∞	∞	∞	∞	∞
4	8	5	5	∞	∞
5	∞	∞	∞	3	∞

Nous y voyons que l'absence de liaison entre deux nœuds à un poids d'une valeur ∞.

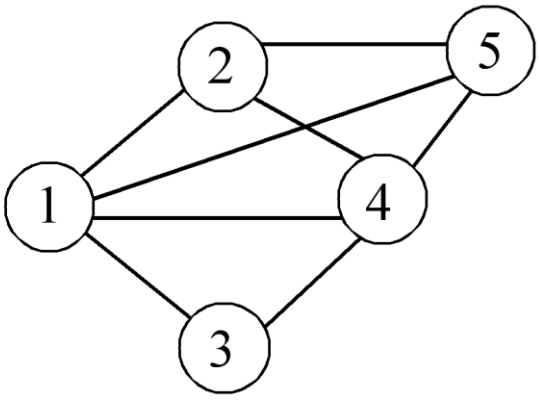
Q8. Remplir la matrice d'adjacence du graphe ci-dessous :



	A	B	C	E	F	G
A						
B						
C						
E						
F						
G						

Graphes non orientés

La représentation par une matrice d'adjacence de ces graphes ne pose pas de problèmes particuliers. Chaque liaison occupe deux cases dans la matrice car les liaisons sont symétriques.



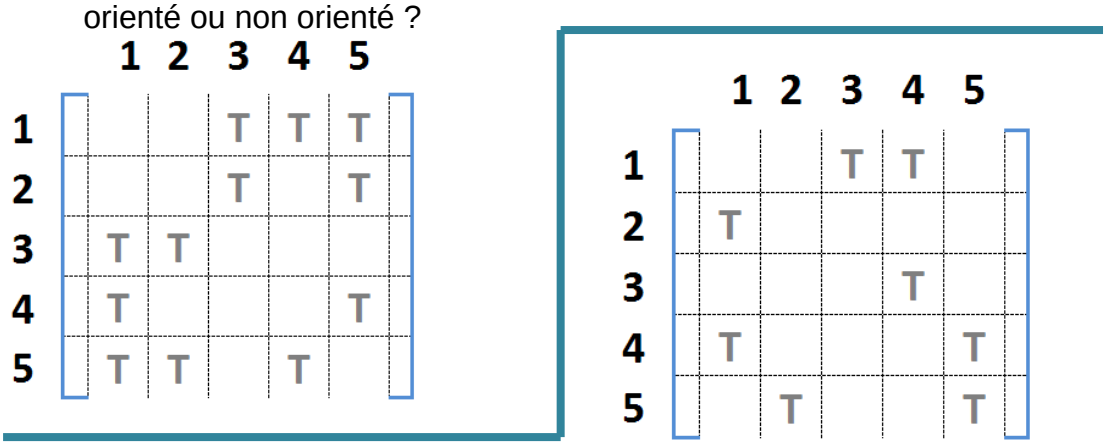
	1	2	3	4	5
1	F	T	T	T	T
2	T	F	F	T	T
3	T	F	F	T	F
4	T	T	T	F	T
5	T	T	F	T	F

Symétrie du graphe non orienté.

La symétrie illustre le fait que les arcs ne sont pas orientés et donc que chaque liaison est bidirectionnelle.

Q9. Retracez le graphe à partir de la matrice d'adjacence ci-dessous, le graphe est-il orienté ou non orienté ?

	1	2	3	4	5
1			T	T	T
2			T		T
3	T	T			
4	T				T
5	T	T		T	



Q10. Représentez le graphe qui a la matrice d'adjacence ci-dessus.

2.2 Les scripts Python[8]

Une classe Graphe présentation

La classe graphe ci-dessous permet de décrire les graphes avec une matrice d'adjacence. Elle est basée sur le paradigme de programmation objet.

```
5  class Graphe:
6      """un graphe représenté par une matrice d'adjacence,
7         où les sommets sont les entiers 0,1,...,n-1"""
8
9      def __init__(self, n):
10         self.n = n
11         self.adj = [[False] * n for _ in range(n)]
12
13     def ajouter_arc(self, s1, s2):
14         self.adj[s1][s2] = True
15
16     def arc(self, s1, s2):
17         return self.adj[s1][s2]
18
19     def voisins(self, s):
20         v = []
21         for i in range(self.n):
22             if self.adj[s][i]:
23                 v.append(i)
24         return v
```

[8] Les scripts sont inspirés de NSI 24 leçons avec exercices corrigés, ellipses.

Pour réviser nos connaissances en Python et POO répondre aux questions ci-dessous :

POO

Q11. Quelle est l'utilité de définir une classe Graphe ?

Q12. Comment s'appellent les fonctions décrites à l'intérieur de la classe ?

Q13. Comment s'appellent les structures de données qui contiennent les données des entités instanciées par la Classe ?

Q14. Quel est le nom de la fonction __init__ ?

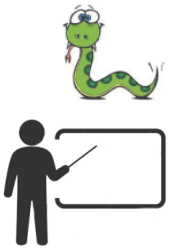
Q15. Quel est son rôle ?

Python

Q16. Comment s'appelle la construction Python qui permet d'initialiser la matrice d'adjacence, ligne 11 du code ?

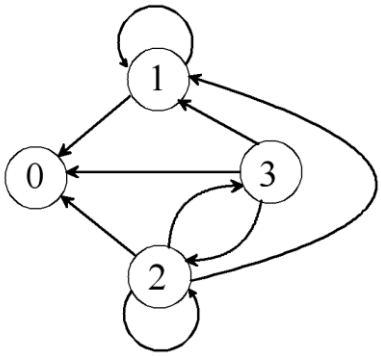
Q17. Que fait l'instruction **[False]*n**

Mise en œuvre codage d'un graphe non orienté



Script_Graphe_1. Compléter le script pour décrire le graphe ci-dessous. (*Graphe identique à celui de la question Q7 mais avec les nœuds numérotés à partir de 0*)

Script_Graphe_1.py



Résultat attendus :

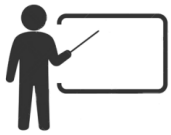
```
In [4]: g.adj
Out[4]:
[[False, False, False, False],
 [True, True, False, False],
 [True, True, True, True],
 [True, True, True, False]]
```

	0	1	2	3
0	.	.	.	.
1	T	T	.	.
2	T	T	T	T
3	T	T	T	.

Amélioration impression de la matrice d'adjacence

Script_Graphe_2. Compléter le script 1 en ajoutant une méthode pour imprimer la matrice d'adjacence. Le format de l'impression est donné ci-dessus.

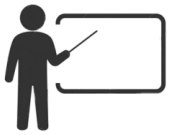
Script_Graphe_2.py



Ajout de la possibilité d'impression directement par print(objet)

Objet est un objet de la classe Graphe

Script_Graphe_3. Compléter le script précédent pour pouvoir imprimer la matrice d'adjacence directement en faisant print(g) où g est un graphe instancié par la Classe Graphe. Script_Graphe_3.py



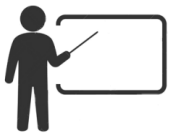
Exemple de résultat attendus :

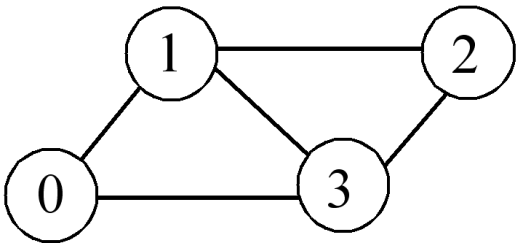
```
Matrice d'adjacence
    0   1   2   3
0   .   .   .   .
1   T   T   .   .
2   T   T   T   T
3   T   T   T   .
```

Q18. Qu'est-ce qui différencient les deux fonctions d'impressions questions Q17 et Q18 ?

Ajout d'une arête (arc non orienté)

Script_Graphe_4. Ajouter une méthode permettant l'existence d'arêtes dans les graphes. Script_Graphe_4.py






```
Matrice d'adjacence
    0   1   2   3
0   .   T   .   T
1   T   .   T   T
2   .   T   .   T
3   T   T   T   .
```

Graphe pondéré

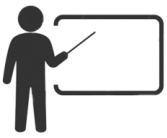
Pour représenter cette catégorie de graphe il nous faut pouvoir utiliser la valeur infini ∞ pour indiquer l'absence de liaison entre deux nœuds.

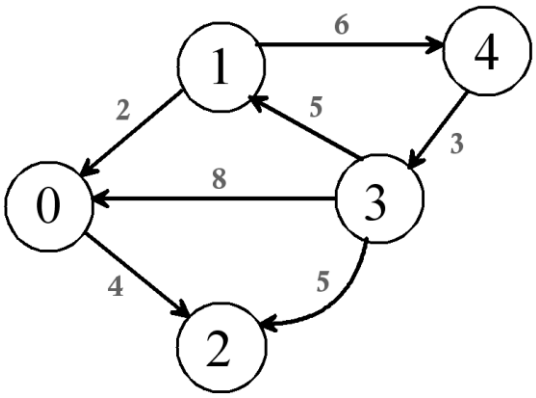
La valeur infini existe en Python :

```
In [7]: infini = float('inf')

In [8]: infini
Out[8]: inf
```

Script_Graphe_5. Créer une classe GraphePondere qui reprend toutes les méthodes précédemment utilisées mais adaptées à ce type de graphe. Créer le graphe ci-dessous.

Script_Graphe_5.py





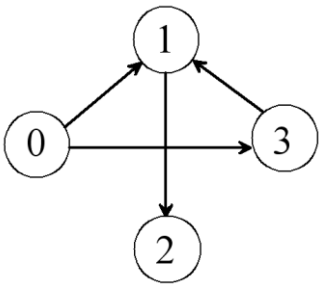
	0	1	2	3	4
0	inf	inf	T	inf	inf
1	T	inf	inf	inf	T
2	inf	inf	inf	inf	inf
3	T	T	T	inf	inf
4	inf	inf	inf	T	inf

```
In [20]: g.arc(1,2)
Out[20]: inf

In [21]: g.arc(3,1)
Out[21]: 5
```

2.3 Représentation par dictionnaire d'adjacence

Principe de la représentation

Une autre solution pour représenter les graphes consiste à dresser une liste de tous les nœuds, avec pour chacun d'entre eux la liste des nœuds vers lesquels ils sont reliés. Prenons un exemple en analysant le graphe ci-dessous :



Le nœud n°0 est relié aux nœuds 1 et 3,
Le nœud n°1 est relié au nœud 2,
Le nœud n°3 est relié an nœud 1.

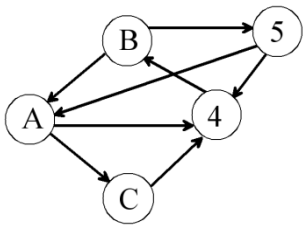
En python nous utiliserons un dictionnaire pour décrire la liste de tous les nœuds du graphe, et un ensemble contenant la liste des nœuds destination des arcs comme valeur. Pour le graphe pris comme exemple voilà ce que cela donne :

```
In [35]: g.adj
Out[35]: {0: {1, 3}, 1: {2}, 2: set(), 3: {1}}
```

Avantage de cette représentation

La représentation de tous les nœuds du graphe par un dictionnaire permet de ne pas se restreindre à identifier ceux-ci uniquement par des numéros commençant par 0. Il est maintenant possible d'élargir les identifications.

Un exemple :




```
Table d'adjacence
B -> A 5
C -> 4
A -> C 4
4 -> B
5 -> A 4
```

2.4 Les scripts Python

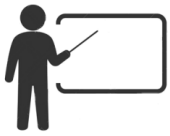
```
5  class Graphe:
6      """un graphe comme un dictionnaire d'adjacence"""
7
8      def __init__(self):
9          self.adj = {}
10
11     def ajouter_sommet(self, s):
12         if s not in self.adj:
13             self.adj[s] = set()
14
15     def ajouter_arc(self, s1, s2):
16         self.ajouter_sommet(s1)
17         self.ajouter_sommet(s2)
18         self.adj[s1].add(s2)
```

```
20     def arc(self, s1, s2):
21         return s2 in self.adj[s1]
22
23     def sommets(self):
24         return list(self.adj)
25
26     def voisins(self, s):
27         return self.adj[s]
```

Mise en œuvre

Script_Graphe_6. A partir du script ci-dessous contenant la classe Graphe construite avec un dictionnaire d'adjacence compléter les méthodes d'impression. Les exemples de sorties sont indiquées page précédente.

Script_Graphe_6.py

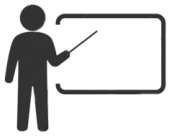


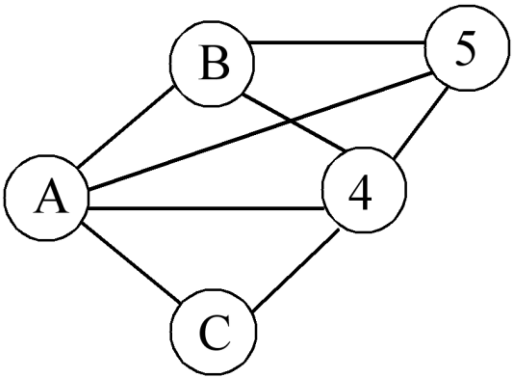
Nous pouvons reprendre le même travail de complément que celui fait dans le paragraphe 1 concernant la représentation par matrice d'adjacence.

Ajout d'une arête

Script_Graphe_7. Ajouter la possibilité de renseigner des arêtes entre deux nœuds.

Script_Graphe_7.py

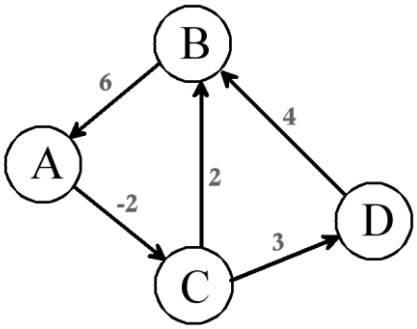





```
Table d'adjacence
B -> A 4 5
C -> A 4
A -> C 4 B 5
4 -> C A B 5
5 -> A B 4
```

Graphe pondéré

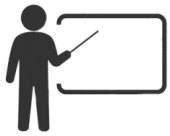
Pour pouvoir représenter la valeur des arcs ou arêtes il nous faut améliorer notre représentation et utiliser maintenant un dictionnaire de dictionnaire :




```
In [4]: g.adj
Out[4]: {'A': {'C': -2}, 'D': {'B': 4}, 'C': {'D': 3, 'B': 2}, 'B': {'A': 6}}
```

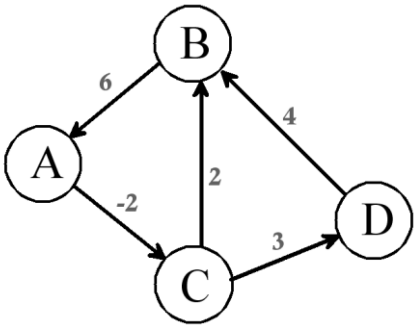
Script_Graphe_8. Compléter le script pour représenter des graphes pondérés.

Script_Graphe_8.py



Quelques résultats attendus :

```
In [8]: g.arc('A','C')
Out[8]: True

In [9]: g.arc('A','D')
Out[9]: False

In [10]: g.sommets()
Out[10]: ['A', 'D', 'C', 'B']

In [22]: g.poids('A','C')
Out[22]: -2

In [23]: g.poids('A','D')
Out[23]: inf
```




```
In [15]: g.voisins('C')
Out[15]: {'B', 'D'}

In [13]: g.voisins('B')
Out[13]: {'A'}
```

2.5 Comparaison des deux implantations

Nous avons illustré nos implantations de graphe en Python avec deux procédés différents.

- Une représentation par matrice d'adjacence
- Une représentation par dictionnaire d'adjacence

Ces deux représentations permettent de décrire tous les types de graphes orientés, non orientés ou bien pondérés. Elles ne sont pas totalement équivalentes.

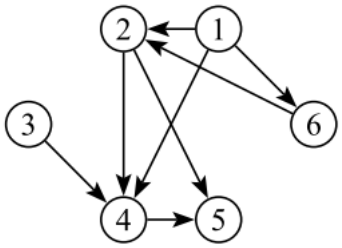
La représentation avec matrice d'adjacence

Cette représentation est basée sur une matrice M de dimension [n, n] où n est le nombre de nœuds du graphe, le remplissage de cette matrice avec des valeurs non nulles dépend du graphe. Elle est le plus souvent remplie de 0 ou ∞ les différents nœuds n'étant en général reliés qu'à quelques autres nœuds du graphe. L'occupation en mémoire peut donc être très grande.

Cette représentation permet de déterminer simplement l'existence de chemins de longueur m entre deux nœuds en calculant M^2 pour des chemins de longueur 2, M^3 pour des chemins de longueur 3 etc …

Illustration[9]

Voilà un graphe avec sa matrice d'adjacence, (*les nœuds sont numérotés à partir de 1*)



$$M = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$M^2 = \begin{pmatrix} 0 & 1 & 0 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Vérifions l'existence de chemins de longueur 2 ou 3 :

$$M^3 = \begin{pmatrix} 0 & 0 & 0 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Q19. Donner le ou les chemins de longueur 2 entre les nœuds
1 et 2 :
6 et 4 :
2 et 5 :
1 et 5 :

Q20. Donner le ou les chemins de longueur 3 entre les nœuds
1 et 4 :
6 et 5 :
1 et 5 :

[9] CAHIERS DE LA CRM, Introduction à la théorie des graphes, Didier Müller source internet.

Excursus :

A titre indicatif si A, B sont deux matrices carrées de dimension [n, n] alors on note a_{ij} un élément de A, (*i numéro de la ligne et j numéro de la colonne*), B_{ij} un élément de B. Alors C = A•B est également une matrice de dimension [n, n].

C est défini par : $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$ Pour réaliser ce calcul il nous faut :

n^3 multiplications, $n^2(n-1)$ additions.

La complexité du calcul sur la base de cette définition est en $O(n^3)$. Des algorithmes avancés existent actuellement la complexité[10] la plus performante est en $n^{2,3727}$ en 2011.

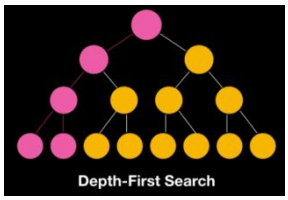
La représentation avec dictionnaire d'adjacence

Elle permet de représenter de très gros graphes en optimisant l'occupation en mémoire de celle-ci. La recherche dans les dictionnaires est en temps constant $O(1)$ en Python. Par contre le calcul des chemins comme précédemment n'est pas possible.

3 Parcours de graphes[11]

3.1 Présentation

Parcours en profondeur (DFS : Depth-First Search)



GIF

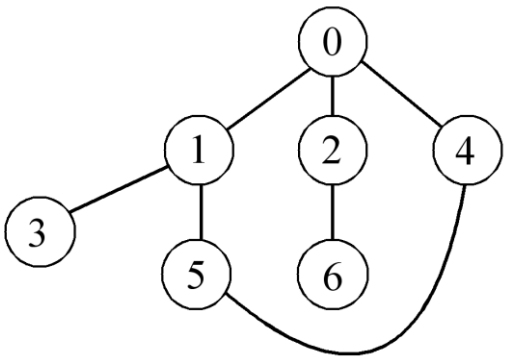
L'exploration d'un parcours en profondeur depuis un sommet S fonctionne comme suit :

A partir d'un nœud de départ l'algorithme d'exploration poursuit un chemin dans le graphe jusqu'à un cul-de-sac ou alors jusqu'à atteindre un sommet déjà visité. Il revient alors sur le dernier sommet où on pouvait suivre un autre chemin puis reprend l'exploration.

Les sommets visités sont donc marqués au fur et à mesure de la progression.

L'exploration s'arrête quand tous les sommets depuis S ont été visités.

https://www.youtube.com/watch?v=kcedjJOjDpg



Q21. Donner le parcours en profondeur à partir du nœud
0 :
2 :
5 :

Note : Quand il y a plusieurs fils possibles à partir d'un nœud le chemin suivit dépend du choix de l'algorithme.

⇦

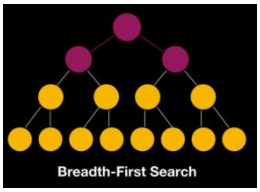
[10] http://www.dptinfo.ens-cachan.fr/Conferences/Koiran13.pdf
[11] Les sources diverses utilisées dans ce paragraphe sont indiquées dans la bibliographie.

Parcours en largeur (Breadth First Search)

Le parcours en largeur explore le graphe couche par couche. Cela revient à s'éloigner progressivement du point de départ. Par rapport à ce point de départ la progression est 'horizontale'.

Pour réaliser ce fonctionnement les nœuds découverts sont mis dans une file au fur et à mesure et traités dans l'ordre de cette file ensuite.

https://www.youtube.com/watch?v=NrQGxfFMYzs



Q22. Donner le parcours en largeur à partir du nœud **GIF**

0 :

2 :

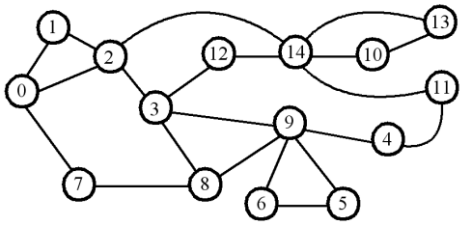
5 :

3.2 Implémentation en Python

Le graphe d'étude

Le graphe ci-dessous sera utilisé pour tester nos algorithmes

(Voir feuille jointe en annexe)

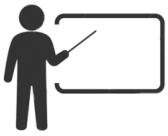


Version récursive du parcours en profondeur DFS

L'implémentation est basée sur l'utilisation de deux listes, une liste dénommée visited qui contient la liste des nœuds déjà visité du graphe. A la fin de l'algorithme cette liste contiendra les nœuds dans l'ordre du parcours. Une deuxième liste unvisited contient la liste des nœuds non encore visités à partir du nœud courant.

```
78  def recursive_dfs(graph, node, visited=None):
79      '''
80      Parcours en profondeur DFS
81      Implémentation récursive
82      Le fonctionnement est réalisé avec deux listes :
83      visited   : contient la liste des noeuds déjà explorés
84      unvisited : donne la liste des noeuds non encore visité à partir
85                  du noeud courant.
86      '''
87      # Initialisation de la liste des sommets
88      # visités
89      if visited is None:
90          visited = []
91
92      # Si le noeud courant n'est pas dans la liste
93      # des noeuds visité on l'ajoute
94      if node not in visited:
95          visited.append(node)
96
97      # Liste des noeuds non encore visités à partir
98      # du noeud courant
99      unvisited = [n for n in g.voisins(node) if n not in visited]
100
101     # On poursuit en profondeur pour tous les noeuds
102     # restants à découvrir
103     for node in unvisited:
104         recursive_dfs(graph, node, visited)
105
106     # Retour du résultat
107     return visited
```

Script_Graphe_9. Complétez le script ci-dessous pour réaliser le parcours en profondeur du graphe en implémentation récursive. Script_Graphe_9.py



Notez les résultats de vos parcours avec comme sommet de départ :
0 :
9 :
13 :

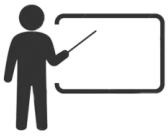
Version itérative du parcours en profondeur DFS

Le graphe est parcouru à partir du nœud de départ. Les nœuds visités sont stockés dans la liste visited. Pour chaque nœud visité la liste de tous ses fils non encore explorés est inséré dans une Pile de type LIFO. (Last In First Out le dernier enregistré est le premier à sortir).

Le traitement se poursuit tant qu'il reste des nœuds dans la Pile. A noter que pendant le déroulement de l'exploration il est possible d'avoir plusieurs occurrences d'un même nœud dans la pile en fonction de la topologie du graphe et de l'ordre du parcours d'exploration réalisé. Cela n'a pas d'importance car quand l'élément au sommet de la Pile est prélevé pour être analysé, s'il est déjà visité à ce moment de l'algorithme de parcours, alors il est simplement ignoré et le processus de vidage de la Pile continue.

```
51  def iterative_dfs(graph, node):
52      '''
53      Parcours en profondeur DFS
54      Implémentation itérative.
55      Utilisation d'une Pile (stack en anglais) les successeurs du noeud en
56      cours d'analyse sont insérés dans la pile. La pile se vide au fur et à
57      mesure de la boucle non bornée while.
58      L'implantation de la pile repose sur le module Python collections.deque
59      Au fur et à mesure de leur découverte les noeuds sont ajoutés à la liste
60      visited qui est renvoyée en résultat
61      '''
62      # Initialisations
63      visited = []
64      stack = deque()
65      stack.append(node)
66
67      # Tant que la Pile n'est pas vide
68      while stack:
69          # On extrait l'élément de droite de la pile
70          node = stack.pop()
71          # Si le noeud extrait n'est pas déjà visité
72          if node not in visited:
73              # On l'ajoute à la liste des noeuds visités
74              visited.append(node)
75              # On établi la liste des neouds fils non encore
76              # visités
77              unvisited = [n for n in g.voisins(node)  if n not in visited]
78              # On insère cette liste dans la Pile
79              stack.extend(unvisited)
80
81      # Retour du résultat
82      return visited
```

Script_Graphe_10. Complétez le script ci-dessous pour réaliser le parcours en profondeur du graphe en implémentation itérative. Script_Graphe_10.py



Notez les résultats de vos parcours avec comme sommet de départ :
0 :
9 :
13 :

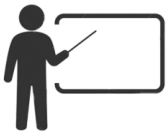
Q23. Expliquez pourquoi les deux algorithmes ne donnent pas les nœuds dans le même ordre.

3.3 Parcours en largeur ou BFS

Version itérative

```
143 def iterative_bfs(graph, start):
144     '''
145     Parcours en largeur du graphe.
146     Les noeuds découverts sont stockés dans une file (queue en anglais) FIFO
147     First In First Out et donc ils sont traités par couche
148     s'éloignant du noeud de départ de manière concentrique.
149     L'implantation de la file repose sur le module Python collections.deque
150     Le fonctionnement de la file est obtenu en prélevant les données du coté
151     gauche.
152     '''
153 
154     # Initialisation
155     visited = []
156     queue = deque()
157     queue.append(start)
158 
159     # Tant que la file n'est pas vide
160     while queue:
161         # On récupère un élément à gauche, coté sortie de la FIFO
162         node = queue.popleft()
163         # Si le noeud extrait n'est pas déjà visité
164         if node not in visited:
165             # On l'ajoute à la liste des noeuds visités
166             visited.append(node)
167             # On établi la liste des neouds fils non encore
168             # visités
169             unvisited = [n for n in g.voisins(node) if n not in visited]
170             # On insère cette liste dans la File
171             queue.extend(unvisited)
172 
173     # Retour du résultat
174     return visited
```

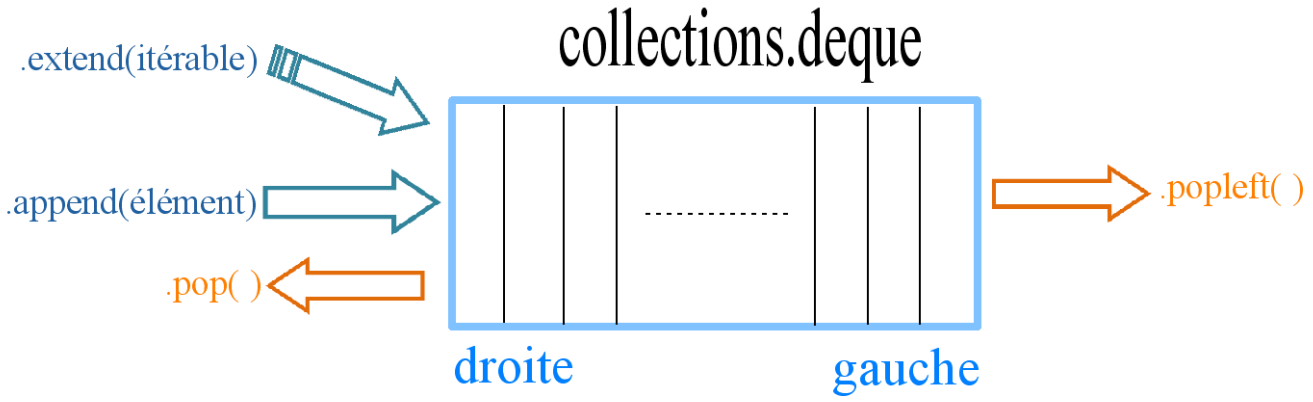
Script_Graphe_11. Complétez le script ci-dessous pour réaliser le parcours en largeur du graphe en implémentation itérative. Script_Graphe_11.py



⇦

Notez les résultats de vos parcours avec comme nœud de départ :
0 :
9 :
13 :

3.4 L'objet collections.deque



Fonctionnement en Pile LIFO (Parcours en Profondeur)

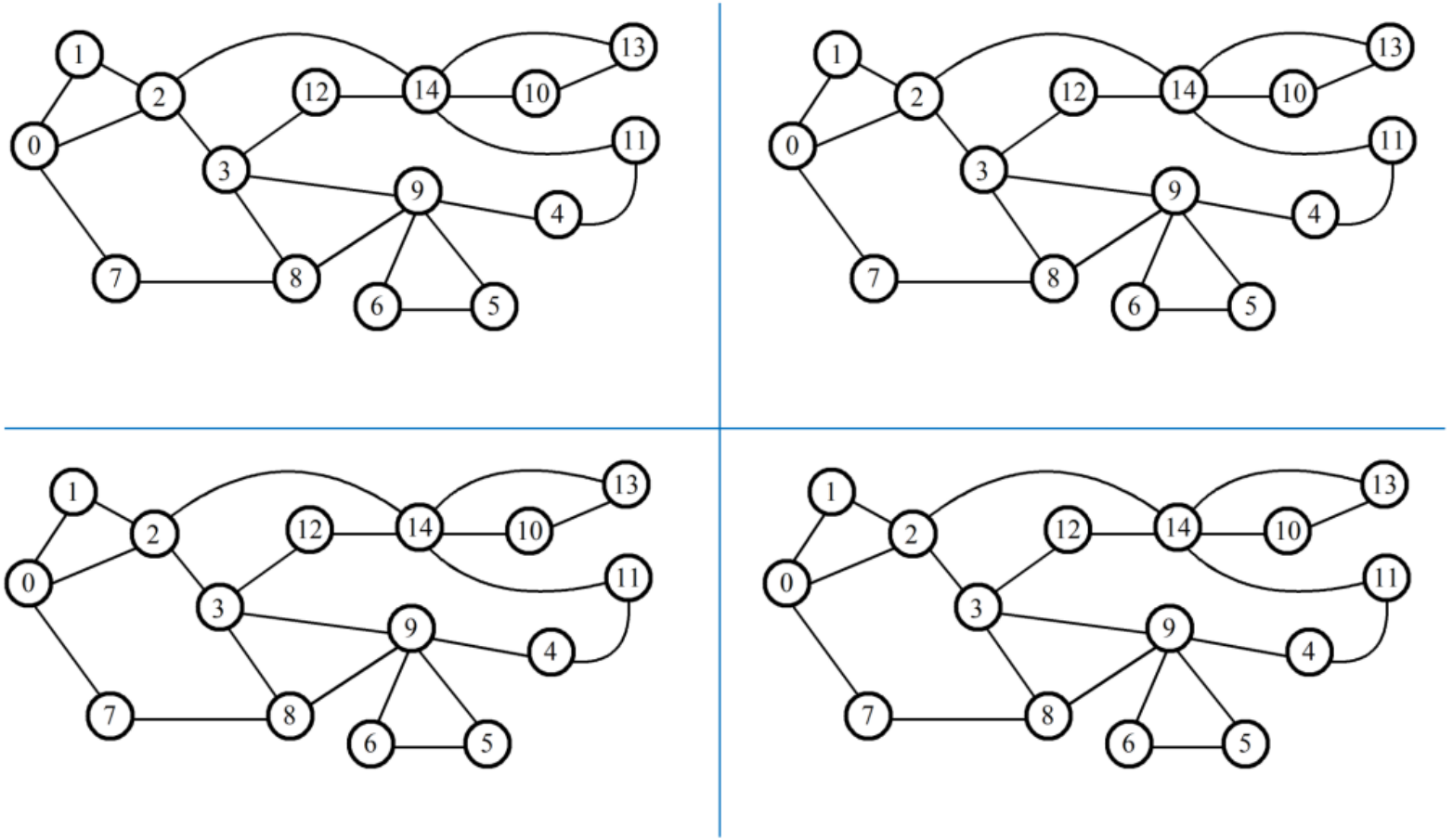
En entrée	En sortie le dernier rentré
Coté droit .extend( ) ou .append( )	Coté droit .pop( )

Fonctionnement en File FIFO (Parcours en Largeur)

En entrée	En sortie le premier rentré
Coté droit .extend( ) ou .append( )	Coté gauche .popleft( )

3.5 Exercices

Parcours de graphes

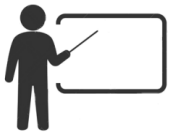


4 Mini projets

4.1 Détection un graphe connexe

Q24. Proposez une méthode qui permet de détecter qu'un graphe est connexe en utilisant un parcours en profondeur ou en largeur.

Script_Graphe_12. Complétez le script ci-dessous pour réaliser détecter la connexité d'un graphe. Script_Graphe_12.py

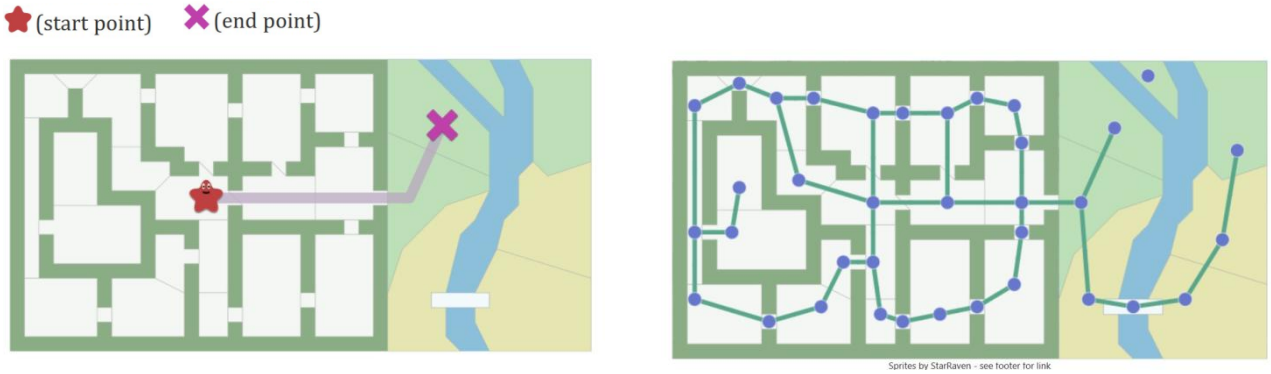


```
def graphe_connexe(graph):
    '''
    Test pour savoir si un graphe est connexe
    Pour cela on effectue un parcours en profondeur et si tous
    les sommets y sont représentés alors le graphe est connexe.
    '''
    # A compléter
```

4.2 Détection de chemin l'algorithme A* (prononcez A star)

La détection de chemin est intensivement utilisée dans les jeux par exemple. Nous exploitons ici l'excellent site[12] (en anglais) donnant les bases de la représentation des terrains et des calculs de trajectoire dans les jeux.

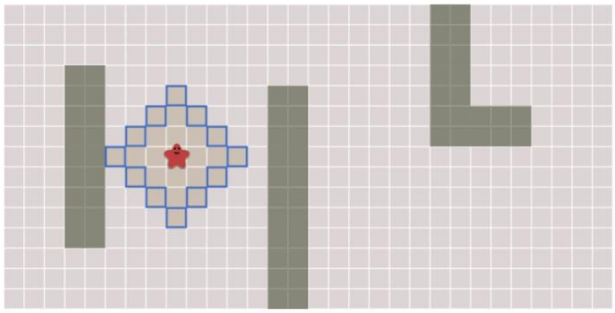
Voilà un exemple de représentation d'un plateau par un graphe :

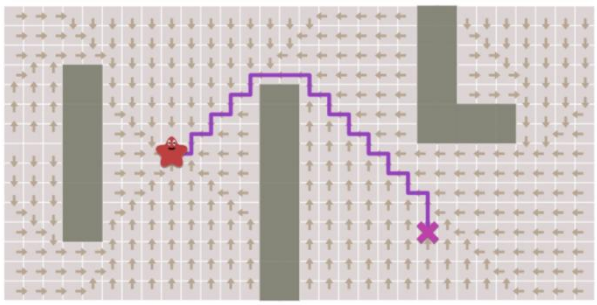


Le principe de l'algorithme A* consiste à parcourir le graphe en largeur de puis la position de départ. On parcourt le graphe en mémorisant le chemin par où on est arrivé. Comme les cailloux dans les bois utilisés par le petit Poucet cela nous permettra de parcourir le chemin 'à l'envers' pour obtenir notre résultat.

[12] https://www.redblobgames.com/pathfinding/a-star/introduction.html

Début de l'algorithme **Fin de l'algorithme**





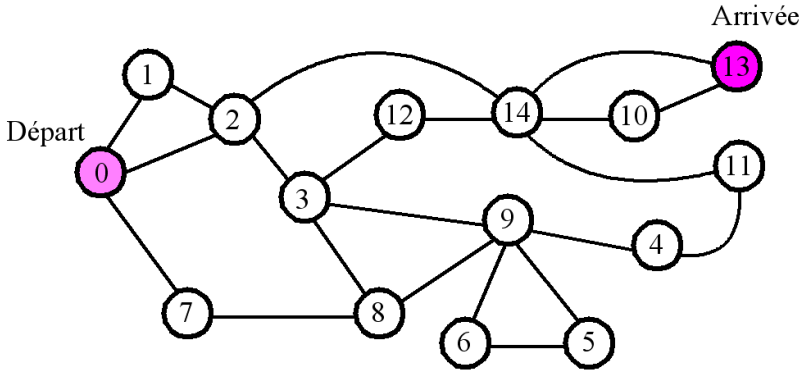
Les carrés bleus représentent la frontière qui progresse jusqu'à explorer toute la carte, (ou le graphe). Les flèches représentent la mémorisation permettant de retrouver de la direction de retour.

Algorithme A* parcours entre un nœud depart et un nœud arrivee

```
// Définition des deux types de données utilisées dans le parcours A*
la_frontiere ← File
la_frontiere ← nœud_de_depart

// Un dictionnaire pour mémoriser du chemin parcouru
came_from ← Dictionnaire
came_from [ nœud_de_depart ] ← None

// Première phase : Exploration de tout le graphe
Tant que la_frontiere n'est pas vide
    nœud_courant ← Sortie_File ( la_frontiere )
    liste_des_voisins ← voisins du nœud courant

    Pour tous les nœuds de la liste_des_voisins
        Si le noeud n'est pas dans la mémoire du chemin parcouru
        Alors
            la_frontiere ← Ajouter_File ( nœud )
            # Mémorise d'où on vient
            came_from [ noeud ] ← nœud_courant

// Deuxième phase : Reconstruction du chemin
nœud_courant ← nœud_arrivee
chemin ← Liste vide
Tant que nœud_courant <> nœud_de_depart
    // On mémorise le chemin en commençant par la fin
    ajouter nœud_courant à chemin
    // Récupérer le nœud d'où on vient
    nœud_courant ← came_from [ nœud_courant ]

// On retourne le chemin entre les nœuds depart et arrivee
ajouter nœud_de_depart à chemin
renverser ( chemin )

retourner chemin
```

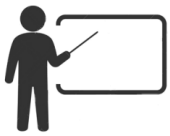
Exemple de résultats :

```
RECHERCHE DE CHEMIN
===================
Depart  :  0
Arrivée :  13

Chemin :
0  2  14  13
```



Script_Graphe_13. Complétez le script ci-dessous pour réaliser la recherche de parcours A* dans un graphe

Script_Graphe_13.py



```
def A_star_search(graphe,depart,arrivee):
    '''
    Recherche de chemin algorithme A*
    https://www.redblobgames.com/pathfinding/a-star/introduction.html
    '''
    # Préparation des données
    frontiere = deque()
    frontiere.append(depart)
    came_from = {}
    came_from[depart] = None

    # A compléter


    # Retour du chemin recherché
    return path
```

4.3 Calcul de chemin

Produit de matrices

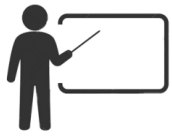
On peut souhaiter déterminer le nombre de chemins de longueur n existants entre deux nœuds d'un graphe. Nous avons vu plus haut que la matrice d'adjacence permettait de calculer ces résultats.

Ces calculs sont réalisés avec un produit de matrice si A, B sont deux matrices carrées de dimension [n, n] alors on note a_{ij} un élément de A, (*i numéro de la ligne et j numéro de la colonne*), B_{ij} un élément de B.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Alors C = A•B est également une matrice de dimension [n, n].

Script_Graphe_14. Complétez le script ci-dessous avec le produit de matrice. Les valeurs True et False de la matrice d'adjacence sont remplacées par 1 et 0 pour les calculs. Script_Graphe_14.py



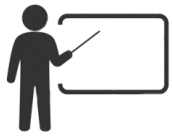
Exemple de résultats :

```
Puissance 1
0  1  0  1  0  1
0  0  0  1  1  0
0  0  0  1  0  0
0  0  0  0  1  0
0  0  0  0  0  0
0  1  0  0  0  0
```

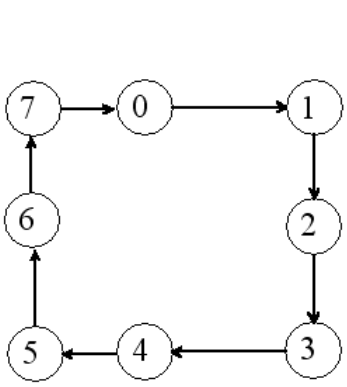
```
Puissance 2
0  1  0  1  2  0
0  0  0  0  1  0
0  0  0  0  1  0
0  0  0  0  0  0
0  0  0  0  0  0
0  0  0  1  1  0
```

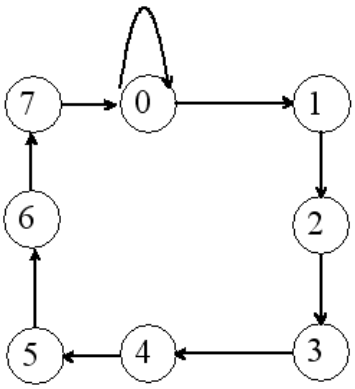
Applications au calcul de l'existence de chemins

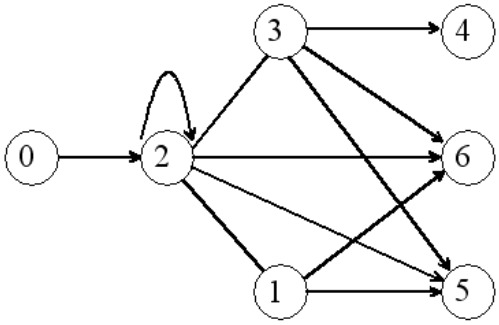
Script_Graphe_15. Vous allez utiliser le produit de matrices dans le script Script_Graphe_15.py pour calculer l'existence de chemins de longueur de 2 à n.



Déterminer l'existence de chemins de longueur 2, 3, 4, …. Pour les graphes ci-dessous :





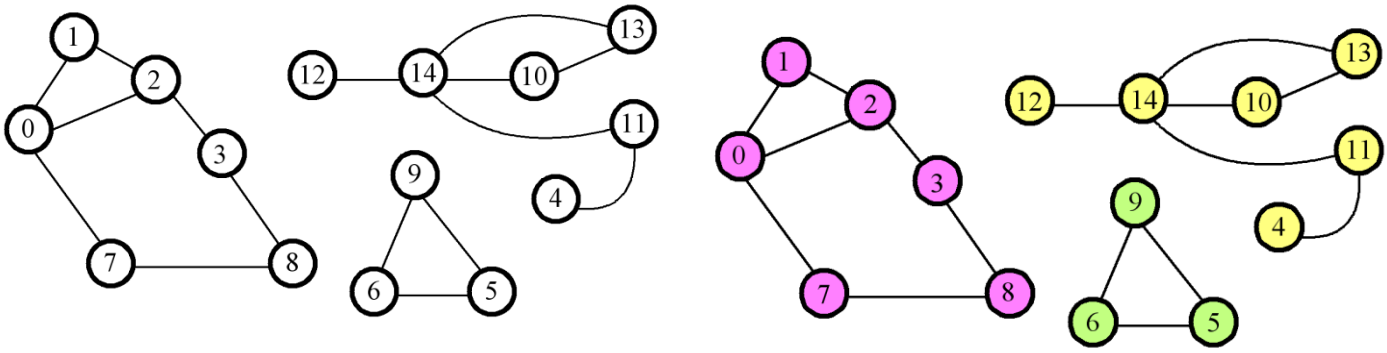


4.4 Identification de composantes connexes

Principe

Nous avons vu plus haut la notion de graphe connexe. Si tous les nœuds ne sont pas reliés les uns les autres mais qu'ils peuvent être regroupés par paquets on parle de composantes connexes.

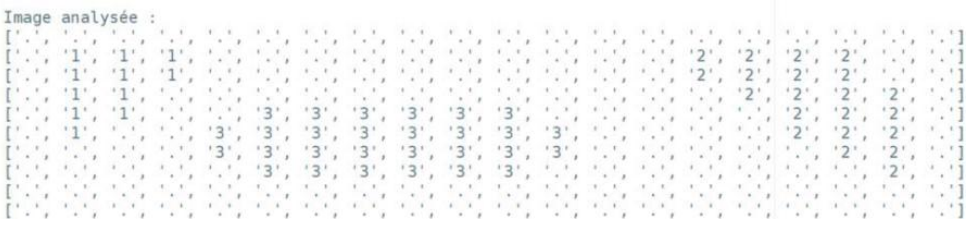
Un exemple sur le graphe ci-dessous où l'on peut identifier trois composantes connexes :



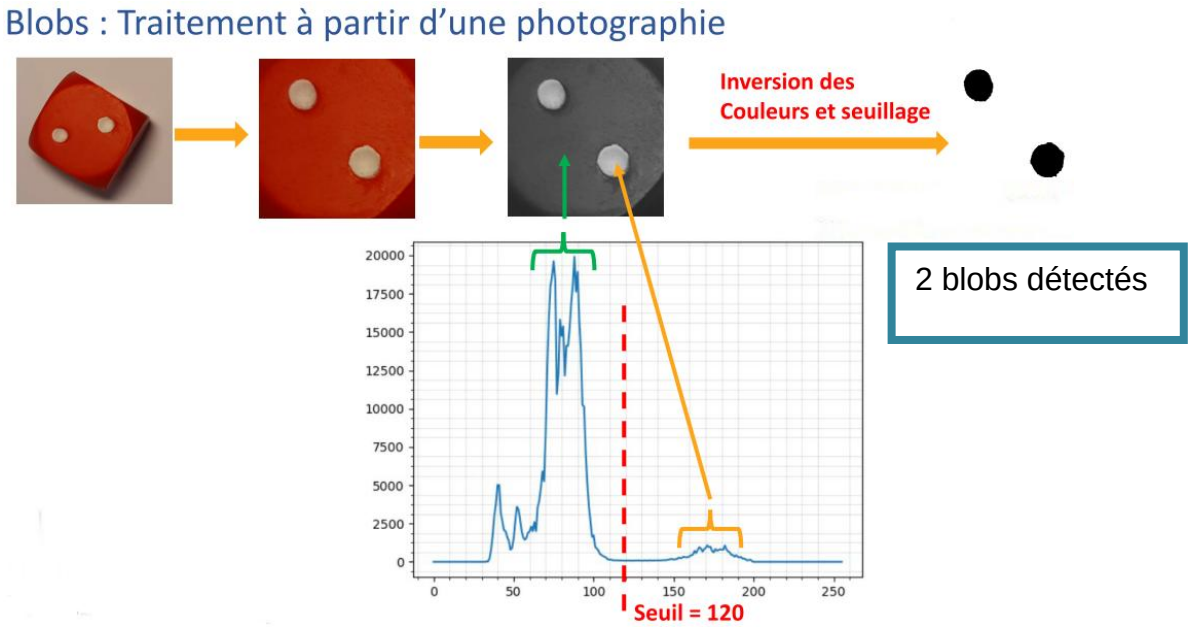
Application à l'imagerie

Cette technique de détection de composantes connexes ou 'blob' peut être utilisée dans l'analyse d'une image.

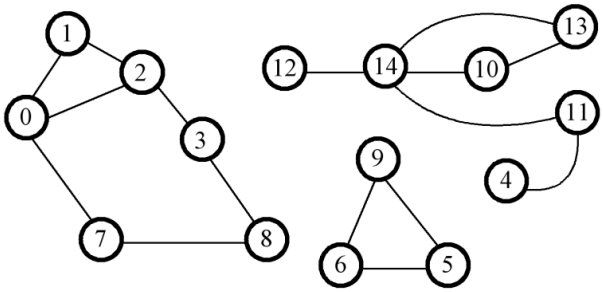
Avec une image de test :



Une image réelle :



Analyse des composantes connexes d'un graphe

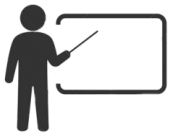


Q25. Avec le graphe ci-contre indiquer en langage courant quelle pourrait être la suite des opérations à réaliser pour détecter toutes les composantes connexes.

Q26. Proposer un algorithme de recherche de composantes connexes d'un graphe.

Script_Graphe_16. Compléter le script Script_Graphe_16.py

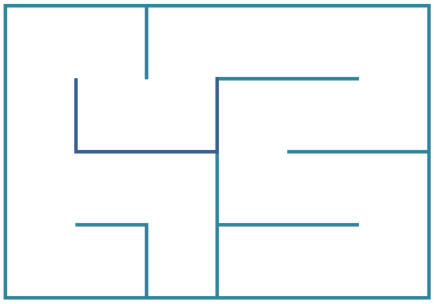
Pour détecter les composantes connexes du graphe.



Exemple de résultats attendus :

```
RECHERCHE DE COMPOSANTES CONNEXES
=================================

Composante  1  :  [0, 7, 8, 3, 2, 1]
Composante  2  :  [4, 11, 14, 13, 10, 12]
Composante  3  :  [5, 9, 6]
```

Mon premier labyrinthe

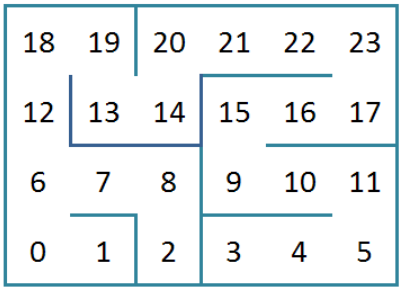


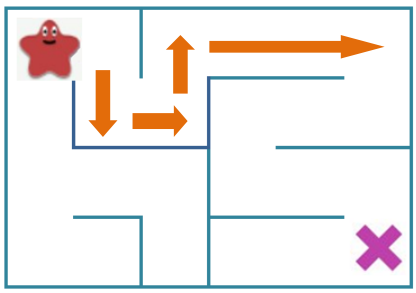
4.5 Graphes et labyrinthes

Introduction

Les graphes sont un outil approprié pour représenter les connexions entre deux objets donc entre les différentes cases d'un labyrinthe.

A partir du tracé d'un labyrinthe nous commençons par numéroter les cases comme ci-dessous :

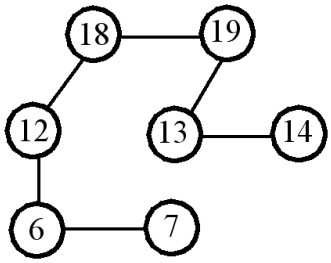




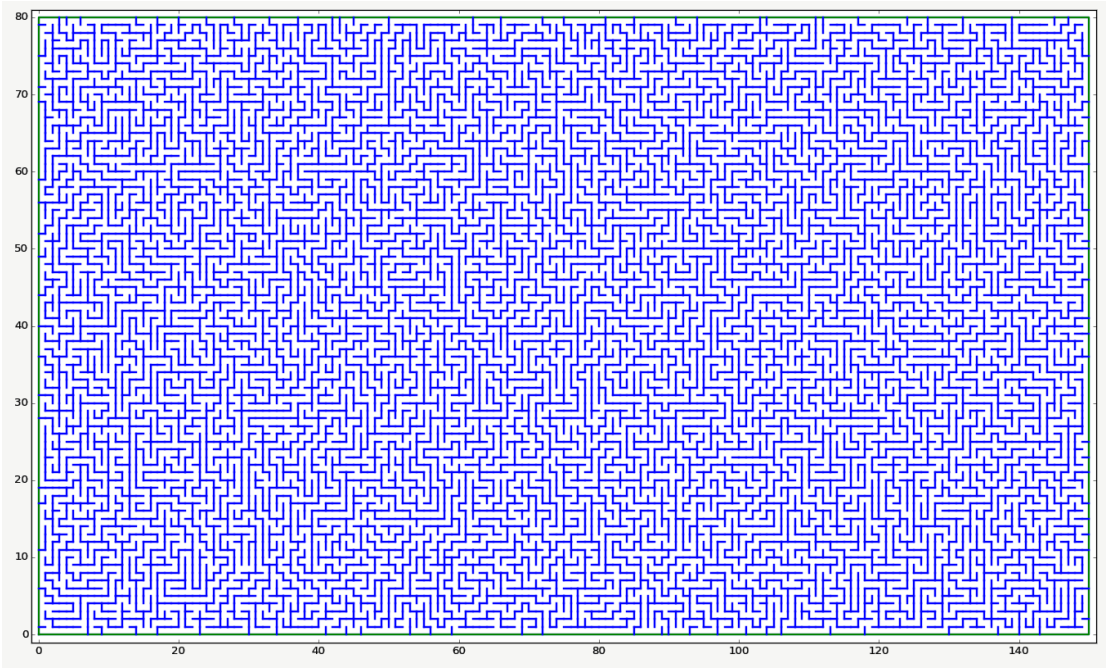
Représentation d'un labyrinthe par un graphe

Toutes les cases sont représentées par un nœud, si on peut passer d'une case à l'autre alors une arête relie les deux sommets du graphe correspondant aux numéros des cases concernées.

Voir ci-contre un début de représentation du labyrinthe précédent.

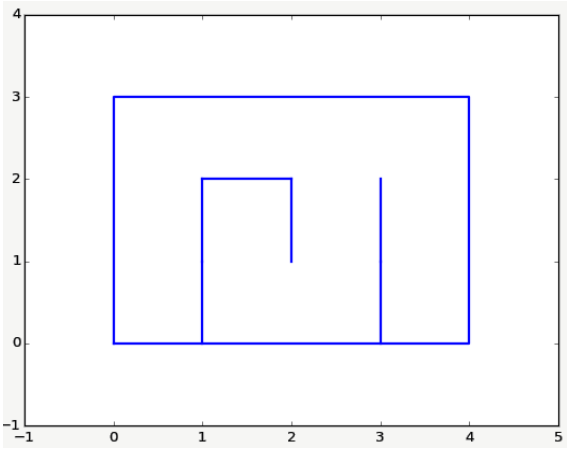


Évidemment les labyrinthes peuvent être un peu plus complexes



Exercices de représentations

A partir du labyrinthe ci-contre :

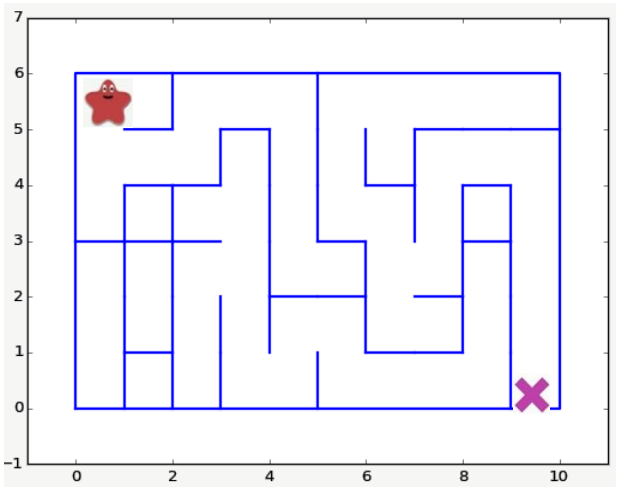


Q27. Numéroter les cases comme plus haut en démarrant par la valeur 0 en bas à gauche et en remontant jusqu'en haut à droite.

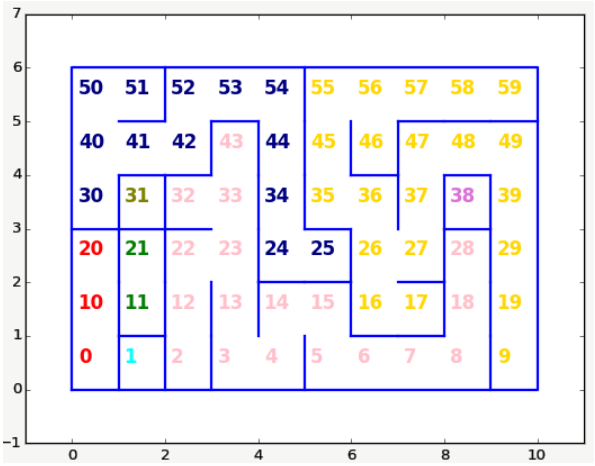
Q28. Représenter le graphe correspondant.

Recherche de composantes connexes

Il est possible de détecter les composantes connexes dans un labyrinthe. Pour cela l'algorithme vu en 4.4 pourra être utilisé.



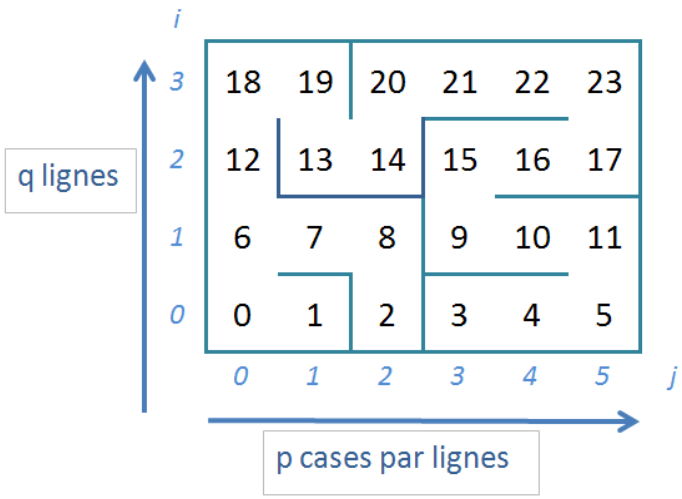
Q29. Combien y a-t-il de composantes dans ce graphe ?



Q30. Y a-t-il un chemin vers la sortie, du Red Blob[13] vers la croix violette ?

4.6 Implémentation en Python des labyrinthes

Les grandes lignes de l'implémentation ont été données plus haut. Il faut représenter le labyrinthe par un graphe. Donc passer d'un système de coordonnées ligne x colonne du labyrinthe à une représentation par numéro de case pour le graphe.



Calcul des coordonnées

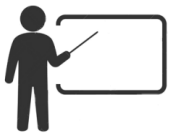
La case de coordonnées i x j = 2 x 3 pour le labyrinthe a comme valeur 15 pour le graphe.

Q31. Donner la relation permettant de calculer le numéro de la case en fonction de la valeur de la ligne de la colonne et de p.

Q32. Donner les deux relations permettant de calculer les coordonnées ligne x colonne à partir du numéro de la case et de p.

[13] Ces deux symboles sont empruntés à l'excellent site https://www.redblobgames.com/

Script_Graphe_17. Compléter les fonctions ci-dessous dans le script Labyrinthe_Et_Graphe_Depart_Eleve.py

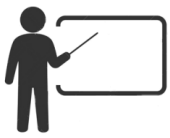


```
def calcul_numero_de_noeud(self, ligne, colonne):
    '''
    Calcul le numéro de noeud du graphe associé
    à la case (ligne,colonne) du labyrinthe.
    '''
    # A completer

    return numero


def calcul_coordonnee_case(self,numero):
    '''
    Calcul les coordonnées ligne x colonne à partir du numéro de
    la case
    '''
    # A completer

    return ligne,colonne
```

Calcul du chemin de sortie algorithme A*

Détermination des composantes connexes

Script_Graphe_18. Compléter le script ci-dessous pour y ajouter la détection du chemin et les composantes connexes. Labyrinthe_Et_Graphe_Depart_Eleve.py



```
def A_star_search(graphe,depart,arrivee):
def composantes_connexes(graphe):
```

Mise en œuvre finale

Tout est maintenant en place vous pouvez utiliser votre programme pour tester différents labyrinthes.

Pour cela une fois le labyrinthe et le graphe créé vous pouvez appeler directement via la console Python les méthodes pour exploiter graphiquement les résultats. Il suffit de les copier/coller directement.

Les données disponibles sont :

- liste_parcours_bfs : contient le parcours en largeur à partir de la case de départ.
- liste_parcours_Astar : trace le trajet vers la sortie s'il existe, i.e. le nœud de sortie appartient au parcours bfs depuis le nœud de départ.
- liste_des_composantes : liste des listes de numéro de nœuds des différentes composantes connexes. Si le graphe est complètement connex il n'y a qu'une seule composante.

Les appels de fonctions disponibles sont :

mon_labyrinthe.show_chemin(liste_parcours_bfs,'cyan)
mon_labyrinthe.show_chemin(liste_parcours_Astar,'green')
mon_labyrinthe.show_numero_cases()
mon_labyrinthe.efface_trace()
mon_labyrinthe.show()
mon_labyrinthe.show_composantes(liste_des_composantes)

```
## INSTRUCTION A TESTER DANS LA CONSOLE
'''

mon_labyrinthe.show_chemin(liste_parcours_bfs,'cyan)
mon_labyrinthe.show_chemin(liste_parcours_Astar,'green')
mon_labyrinthe.show_numero_cases()
mon_labyrinthe.efface_trace()
mon_labyrinthe.show()
mon_labyrinthe.show_composantes(liste_des_composantes)

'''
```

5 Autres ressources

5.1 Sur la théorie des graphes

http://www.bibmath.net/ressources/index.php?action=affiche&quoi=bde/graphes/graphes_pratiques&type=fexo

https://www.jesuisundev.com/comprendre-les-algorithmes-de-parcours-en-8-minutes/

https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_profondeur

5.2 Implémentation Python

Source de départ des implémentations des parcours BFS et DFS de ce cours

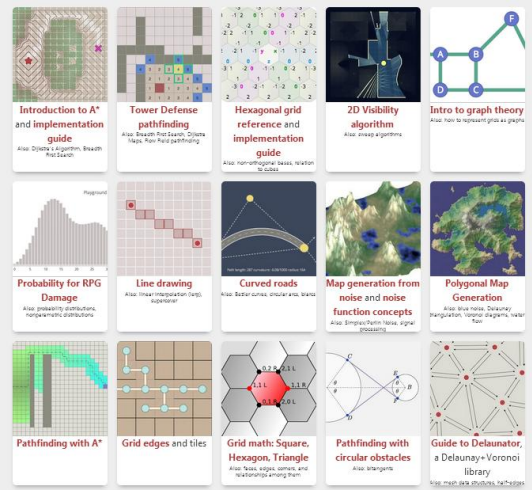
https://marcarea.com/weblog/2019/02/17/parcours-de-graphes-en-python

https://www.jesuisundev.com/comprendre-les-algorithmes-de-parcours-en-8-minutes/

https://developpement-informatique.com/article/329/detecter-un-cycle-dans-un-graphe-oriente-non-oriente

5.3 Sur la théorie des graphes appliquée aux jeux

https://www.redblobgames.com/



5.4 Sur les labyrinthes

http://www.mathsoup.xyz/mathsoup.xyz/content/Informatique/Fiche%20d'activit%C3%A9%204%20-%20g%C3%A9n%C3%A9ration-labyrinthe/g%C3%A9n%C3%A9ration-labyrinthes%20-%20%C3%A9l%C3%A8ves.html