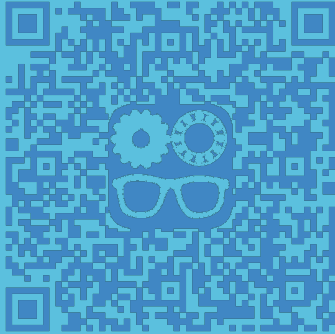




Les graphes



Renaud Costadoat
Lycée Dorian



Présentation

Caractéristiques

Implémentation

Parcours

Introduction

La théorie des graphes un domaine très important en informatique et dans les sciences en général. L'origine des graphes remonte au problème de Königsberg étudié par Euler en 1736.

La ville de *Königsberg* est construite autour de deux îles situées sur le fleuve *Pregel* et reliées entre elles par un pont.

6 autres ponts relient les rives de la rivière à l'une ou l'autre des deux îles.

Le problème que s'est posée Euler consiste à déterminer s'il existe ou non une promenade dans les rues de Königsberg permettant, à partir d'un point de départ au choix, de passer une et une seule fois par chaque pont, et de revenir à son point de départ, étant entendu qu'on ne peut traverser le fleuve qu'en passant sur les ponts.

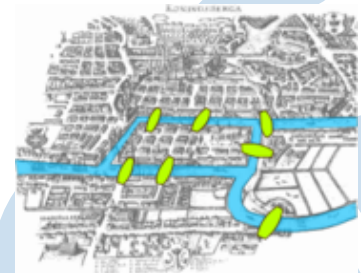


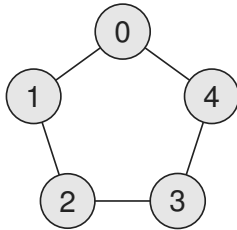
Figure – Ponts de *Königsberg*

Définitions

Définition

Un **graphe** $G = (S, A)$ est un couple composé :

- d'un ensemble S de points appelés **sommets** (ou vertex en anglais),
- d'un ensemble A de liens appelés **arêtes** (ou edge en anglais).



Ce graphe $G = (S, A)$ est défini par :

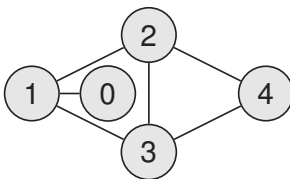
- $S = 0, 1, 2, 3, 4$
- $A = '01', '12', '23', '34', '40'$

- Les sommets reliés par une arête sont ses **extrémités**,
- Une arête forme une **boucle** si ces extrémités sont identiques,
- Les sommets sont **voisins** s'ils sont reliés par une arête,
- Les arêtes peuvent être **orientées** (*flèches*) et donc imposer un sens de parcours des sommets.

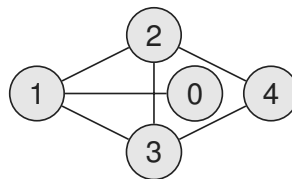


Définitions

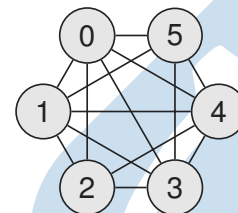
Un graphe est dit **planaire** s'il existe une représentation dans laquelle aucune de ces arêtes ne se croisent.



Graphe planaire

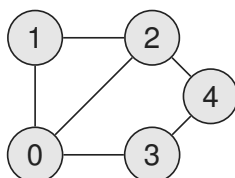


Graphe planaire

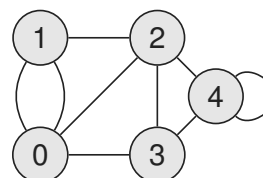


Graphe **non** planaire

Un graphe est dit **simple** si au plus une arête relie deux sommets et s'il n'y a pas de **boucle** sur un sommet. Sinon, c'est un **multigraphe**.



Graphe simple

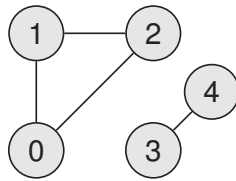


Multigraphe

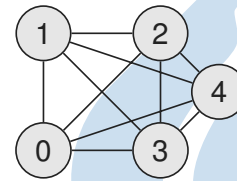


Définitions

Un graphe est **connexe** s'il est possible, à partir de n'importe quel sommet de rejoindre tous les autres sommets en suivant les arêtes. Un graphe non connexe se décompose en plusieurs sous graphes connexes. Un graphe est **complet** si tous les sommets sont des voisins directs.

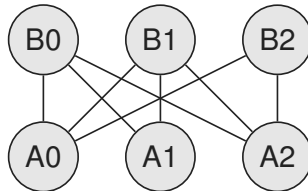


Graphe non connexe

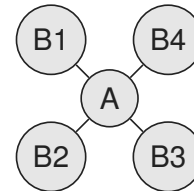


Graphe complet

Un graphe est dit **biparti** si son ensemble de sommets S peut être partitionné en deux sous ensembles de sommets A et B de façon à ce que toute arête ait une extrémité dans A et une extrémité dans B . Une **étoile** est donc un graphe biparti dont l'une des bipartitions n'a qu'un sommet.



Graphe biparti



Graphe étoile

Navigation icons: back, forward, search, etc.

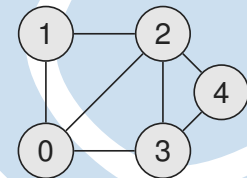
Définitions

Definition

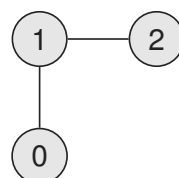
- Dans un graphe non orienté, un cycle correspond à une chaîne simple dont les extrémités sont confondues et contenant au moins une arête,
- Un arbre est un graphe connexe sans cycle.

Par exemple, sur le graphe précédent, la chaîne 0-1-2-0 forme un cycle. Celui-ci est d'ailleurs dit **élémentaire** car hormis, les deux extrémités de la chaîne les autres nœuds sont tous uniques.

Le cycle 4-3-2-1-0-3-4 n'est pas élémentaire car le nœud 3 est présent deux fois.



Graphe avec cycle



Arbre

Navigation icons: back, forward, search, etc.

Définitions

Dans un graphe non orienté, la détection de cycles est relativement simple. Il faut tout d'abord effectuer un parcours du graphe et en découvrant au fur et à mesure les sommets, il suffit de vérifier si on découvre un nouveau sommet ou si c'est un sommet déjà visité et qui n'est pas le parent du sommet courant. Dans ce dernier cas, c'est donc qu'il existe un cycle.

Exemple

À partir du graphe utilisé pour illustrer les algorithmes de parcours en largeur et profondeur. Déterminer le nombre de cycles présent dans le graphe proposé.

Dans le graphe précédent, on détecte 3 cycles élémentaires.

- A-B-E-A ;
- A-B-C-D-A ;
- A-E-F-G-D-A.

On constate, que ce nombre de cycles correspond aux nombres d'arêtes qui ont été considérés comme « Arête (ou lien) découvert mais inutile » lors des illustrations des parcours de graphe effectués précédemment.

Ainsi, la détection de cycles n'est pas si « dure » que cela et n'a pas de réels impact sur la complexité des algorithmes de parcours.

DORJAN

Renaud Costadoat

S02 - C16

 $\frac{7}{34}$

Caractéristiques

L'**ordre** d'un graphe est le nombre n de sommets du graphe.

Si (u, v) est une arête du graphe orienté $G = (S, A)$, on dit que l'arête (u, v) est **incidente** aux sommets u et v ou encore que l'arête (u, v) quitte le sommet u et arrive dans le sommet v . Le sommet v est **adjacent** au sommet u . Si le graphe est non-orienté, la relation d'adjacence est symétrique.

Definition

On appelle **degré** de v , noté $d(v)$, le nombre d'arêtes incidentes à ce sommet

Le **degré d'un graphe** est le degré maximum de tous ses sommets.

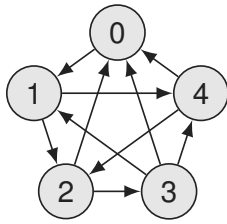
Si m est le nombre d'arêtes, pour un graphe non orienté on aura : $\sum_{v \in V} d(v) = 2.m$

On peut en déduire que le nombre de sommet de degré impaire est forcément pair.

On désigne par **demi-degré extérieur** d'un sommet v d'un graphe orienté, noté $d_+(v)$, le nombre d'arcs qui le quittent. On désigne par **demi-degré intérieur**, noté $d_-(v)$, le nombre d'arcs qui arrivent, avec $d(v) = d_+(v) + d_-(v)$.

Chaînes

Dans un graphe orienté, on appelle **chemin** entre les sommets A et B , noté $\chi(A, B)$, une suite alternant sommets et arêtes reliant le sommet de départ A au sommet d'arrivée B . Dans un graphe non orienté, cette suite est appelée **chaîne** du graphe G .



Par exemple pour ce graphe on a :

- $d_+(3) = 3$
- $d_-(3) = 1$
- $\chi(3, 2) = 3, 0, 1, 2$

Une **chaîne** du graphe G est une suite alternant sommets et arêtes. Cette suite commence par un sommet et se termine par un sommet.

- On appelle **distance** entre deux sommets la longueur de la plus petite chaîne les reliant.
- On appelle **diamètre** d'un graphe la plus longue distance entre deux sommets.
- Une chaîne est **élémentaire** si chaque sommet du graphe apparaît au plus une fois.
- Une chaîne est **simple** si chaque arête du graphe apparaît au plus une fois.
- Une chaîne est **fermée** si le sommet de départ et de fin sont identiques.
- On appelle **cycle** une chaîne fermée simple.



Chaînes

Pour un graphe G ayant m arêtes, n sommets et p composantes connexes, on définit le nombre cyclomatiques $\mu(G)$ du graphe par :

$$\mu(G) = m - n + p$$

Le nombre cyclomatique est positif et représente le nombre de cycles indépendants.

Definition

- On appelle **cycle eulérien** d'un graphe G , un cycle passant une unique fois par chacune des arêtes de G . Un graphe est dit **eulérien** s'il possède un cycle eulérien.
- On appelle **chaîne eulérienne** d'un graphe G , une chaîne passant une unique fois par chacune des arêtes de G . Un graphe est dit **semi-eulérien** s'il ne possède que des chaînes eulériennes.
- On appelle **cycle hamiltonien** d'un graphe G un cycle passant une unique fois par chacun des sommets de G . Un graphe est dit **hamiltonien** s'il possède un cycle hamiltonien.
- On appelle **chaîne hamiltonienne** d'un graphe G une chaîne passant une unique fois par chacun des sommets de G . Un graphe est dit **semi-hamiltonien** s'il ne possède que des chaînes hamiltoniennes.



Implémentation

Afin d'implémenter un graphe dans le langage python, plusieurs solution sont possibles. Soit G un graphe d'ensemble de sommets S et d'ensembles d'arêtes A .

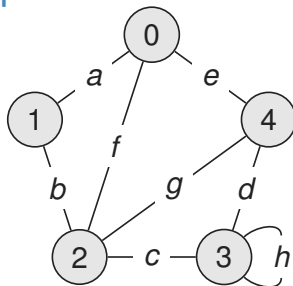
Definition

La **matrice d'incidence** $\mathbb{M}_G = (M_{ve})$ de G est la matrice $n \times m$ où les coefficients (M_{ve}) représentent le nombre d'incidence entre le sommet v et l'arête e .

La **matrice d'adjacence** $\mathbb{A}_G = (A_{uv})$ de G est la matrice carrée $n \times n$ où les coefficients (A_{uv}) représentent le nombre d'arêtes entre les sommets u et v .

Comme la plupart des graphes ont beaucoup plus d'arêtes que de sommets, la matrice d'adjacence est généralement bien plus petite que sa matrice d'incidence et donc nécessite moins d'espace mémoire. Quand il s'agit de graphes simples, une représentation encore plus compacte est possible. Pour chaque sommet v , les voisins de v sont listés selon un ordre quelconque. Une liste $(N(v) : v \in S)$ de ces listes est appelée liste d'adjacence du graphe.

Implémentation



Ce graphe $G = (S, A)$ est défini par :

- $S = 0, 1, 2, 3, 4$
- $A = a, b, c, d, e, f, g, h$

La liste $G = [[1, 2, 4], [0, 2], [0, 1, 3, 4], [2, 3, 4], [0, 2, 3]]$ est une liste d'adjacence.

$G[0]$ renvoie la liste des voisins du sommet 0.

Le dictionnaire $\text{graphe} = \{0: [1, 2, 4], 1: [0, 2], 2: [0, 3, 4], 3: [2, 3, 4], 4: [0, 2, 3]\}$ a la même fonction.

La matrice d'adjacence est alors :

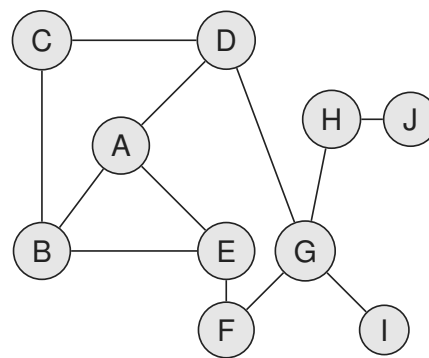
$$\mathbb{A}_G = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Parcours d'un graphe

Le parcours de graphe va permettre d'en déterminer quelques caractéristiques. Celui-ci va donc tout naturellement nous être utile pour vérifier si d'un sommet i , on peut atteindre un autre sommet j . Différentes stratégies sont possibles mais trois sont à connaître :

- Algorithme du parcours en largeur d'un graphe
- Algorithme du parcours en profondeur d'un graphe
- Algorithme de Dijkstra

Soit le graphe suivant:



Graphe exemple



Parcours en largeur

Le parcours en largeur d'un graphe consiste par explorer un nœud départ, puis ses successeurs, puis les successeurs non explorés des successeurs, etc. Ainsi, à partir d'un nœud départ *dep*, on liste d'abord les nœuds voisins de *dep* pour ensuite les explorer un par un et ainsi de suite. Mais une illustration sera sans doute plus représentative :

Légende



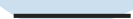
Nœud non
découvert



Nœud exploré



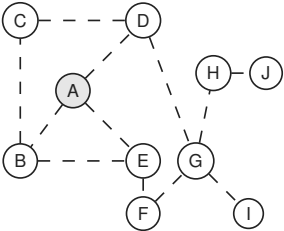
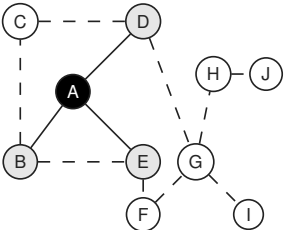
Nœud découvert



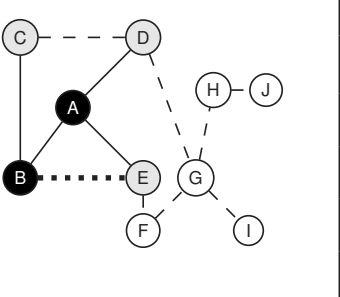
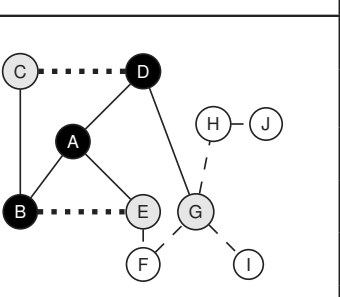
Lien exploré

Nota : Un nœud exploré est un nœud dont on a découvert tous les voisins (ou successeurs).

Parcours en largeur

Illustration	États d'avancement	Commentaires
	<p>Liste des nœuds explorés</p> <p>\emptyset</p> <p>Liste des nœuds découverts</p> <p>A</p> <p>Liste des nœuds à explorer</p> <p>A</p>	<p>Le nœud A est en cours d'exploration.</p>
	<p>Liste des nœuds explorés</p> <p>A</p> <p>Liste des nœuds découverts</p> <p>A B D E</p> <p>Liste des nœuds à explorer</p> <p>B D E</p>	<p>Le nœud A a été exploré et les nœuds B, D et E ont été découverts. Ils sont ajoutés à la liste des nœuds à explorer.</p>

Parcours en largeur

	<p>Liste des nœuds explorés A B</p> <p>Liste des nœuds découverts A B D E C</p> <p>Liste des nœuds à explorer <input type="text" value="D"/> <input type="text" value="E"/> <input type="text" value="C"/></p>	<p>Le nœud B est exploré, on découvre alors le nœud C et de nouveau le nœud E. Mais, celui-ci a déjà été découvert, il n'est pas ajouté la liste des nœuds à explorer.</p>
	<p>Liste des nœuds explorés A B D</p> <p>Liste des nœuds découverts A B D E C G</p> <p>Liste des nœuds à explorer <input type="text" value="E"/> <input type="text" value="C"/> <input type="text" value="G"/></p>	<p>Le nœud D est exploré, on découvre alors de nouveau les nœuds A et C. Seul G qui le nouveau nœud découvert est ajouté à la liste des nœuds à explorer.</p>

Parcours en largeur

	<p>Liste des nœuds explorés A B D E</p> <p>Liste des nœuds découverts A B D E C G F</p> <p>Liste des nœuds à explorer C G F</p>	<p>Le nœud E est exploré, on découvre alors de nouveau les nœuds A et B. Seul F qui est le nouveau nœud découvert est ajouté à la liste des nœuds à explorer.</p>
	<p>Liste des nœuds explorés A B D E C</p> <p>Liste des nœuds découverts A B D E C G F</p> <p>Liste des nœuds à explorer G F</p>	<p>Le nœud C est exploré. Rien ne se passe car ses nœuds voisins ont déjà été découverts.</p>

Parcours en largeur

	<p>Liste des nœuds explorés A B D E C G</p> <p>Liste des nœuds découverts A B D E C G F H I</p> <p>Liste des nœuds à explorer F H I</p>	<p>Le nœud G est exploré. On découvre deux nouveaux nœuds (H et I) qu'on ajoute à la liste de ceux à explorer.</p>
	<p>Liste des nœuds explorés A B D E C G F</p> <p>Liste des nœuds découverts A B D E C G F H I</p> <p>Liste des nœuds à explorer H I</p>	<p>Le nœud F est exploré. Rien ne se passe.</p>

Parcours en largeur

	<p>Liste des nœuds explorés A B D E C G F H</p> <p>Liste des nœuds découverts A B D E C G F H I J</p> <p>Liste des nœuds à explorer I J</p>	<p>Le nœud H est exploré. On découvre un nouveau nœud : le nœud J.</p>
	<p>Liste des nœuds explorés A B D E C G F H I</p> <p>Liste des nœuds découverts A B D E C G F H I J</p> <p>Liste des nœuds à explorer J</p>	<p>Le nœud I est exploré. Rien ne se passe.</p>

Parcours en largeur

	<p>Liste des nœuds explorés A B D E C G F H I J</p> <p>Liste des nœuds découverts A B D E C G F H I J</p> <p>Liste des nœuds à explorer \emptyset</p>	<p>Le nœud J est exploré. L'algorithme s'arrête car la liste des nœuds à explorer est vide.</p>
--	--	--

File

A
BDE
DEC
ECG
CGF
GF
FHI
HI
IJ
J
∅

Remarquez comment évolue la liste des **nœuds à explorer** :

- Les **nouveaux nœuds** sont ajoutés en fin de liste ;
- Le **nœud courant** à explorer est le premier de cette liste,

Les nœuds explorés dès qu'ils sont ajoutés à la liste sont en **orange**.

Cette structure de données est appelée **file**. Elle est composée de quatre grandes manipulations élémentaires :

- « Créer » : Créer la file, au départ celle-ci est vide ;
- « Enfiler » : ajouter un élément dans la file (à la fin de celle-ci) ;
- « Défiler » : renvoyer le prochain élément de la file (premier de celle-ci), et le retire de celle-ci ;
- « Tester si la file est vide » ;

Cette structure s'appuie sur le principe du premier arrivé, premier sorti (FIFO : First in, First out). C'est un principe utilisé dans :

- les files d'attentes (aux caisses du supermarché par exemple) ;
- les serveurs d'impression, qui traitent les requêtes dans l'ordre où elles arrivent ;
- Création de mémoire tampons (buffer).



Pseudo code

Algorithme 1 : Algorithme de parcours de graphe en largeur

Initialisation;

Initialiser la liste *file* avec le nœud départ ;

Initialiser la liste *noeuds_decouverts* avec le nœud départ ;

tant que *file* *n'est pas vide* **faire**

 noeud_courant ← Défiler(*file*) ;

pour *chacun des voisins v de noeud_courant* **faire**

si *v n'appartient pas à la liste noeuds_decouverts* **alors**

 Ajouter *v* à la liste *noeuds_decouverts* ;

 Enfiler *v* à la fin de la liste *file* ;

fin

fin

fin

return *noeuds_decouverts*



Parcours en profondeur

Illustration	États d'avancement	Commentaires
	<p>Liste des nœuds explorés \emptyset</p> <p>Liste des nœuds découverts A</p> <p>Liste des nœuds à explorer A</p>	<p>Le nœud A est le nœud de départ.</p>
	<p>Liste des nœuds explorés A</p> <p>Liste des nœuds découverts A B D E</p> <p>Liste des nœuds à explorer B D E</p>	<p>Le nœud A est exploré.</p> <p>Les nœuds B, D et E sont découverts. Ils sont ajoutés à la liste des nœuds à explorer.</p>

Parcours en profondeur

	<p>Liste des nœuds explorés A E</p> <p>Liste des nœuds découverts A B D E F</p> <p>Liste des nœuds à explorer B D F</p>	<p>Le nœud E est exploré et les nœuds A, B et F ont été découverts. On ajoute uniquement à la liste des nœuds à explorer le nœud F.</p>
	<p>Liste des nœuds explorés A E F</p> <p>Liste des nœuds découverts A B D E F G</p> <p>Liste des nœuds à explorer B D G</p>	<p>Le nœud F est exploré, seul le nœud G est un voisin encore non découvert.</p>

Parcours en profondeur

	<p>Liste des nœuds explorés A E F G</p> <p>Liste des nœuds découverts A B D E F G H I</p> <p>Liste des nœuds à explorer B D H I</p>	<p>Le nœud G est exploré, seuls le nœuds H et I sont ajoutés à la liste des nœuds à explorer car le nœud D a déjà été découvert.</p>
	<p>Liste des nœuds explorés A E F G I</p> <p>Liste des nœuds découverts A B D E F G H I</p> <p>Liste des nœuds à explorer B D H</p>	<p>Le nœud I est exploré. Rien ne se passe.</p>

Parcours en profondeur

	<p>Liste des nœuds explorés A E F G I H</p> <p>Liste des nœuds découverts A B D E F G H I J</p> <p>Liste des nœuds à explorer B D J</p>	<p>Le nœud H est exploré. On ajoute le nœud J à la liste des nœuds à explorer.</p>
	<p>Liste des nœuds explorés A E F G I H J</p> <p>Liste des nœuds découverts A B D E F G H I J</p> <p>Liste des nœuds à explorer B D</p>	<p>Le nœud J est exploré. Rien ne se passe.</p>

Parcours en profondeur

	<p>Liste des nœuds explorés A E F G I H J D</p> <p>Liste des nœuds découverts A B D E F G H I J C</p> <p>Liste des nœuds à explorer B C</p>	<p>Le nœud D est exploré. On découvre le nœud C comme nouveau nœud à explorer.</p>
	<p>Liste des nœuds explorés A E F G I H J D C</p> <p>Liste des nœuds découverts A B D E F G H I J C</p> <p>Liste des nœuds à explorer B</p>	<p>Le nœud C est exploré. Aucun nouveau nœud n'est découvert.</p>

Parcours en profondeur

	<p>Liste des nœuds explorés A E F G I H J D C B</p> <p>Liste des nœuds découverts A B D E F G H I J C</p> <p>Liste des nœuds à explorer \emptyset</p>	<p>Le nœud B est exploré. Aucun nouveau nœud n'est découvert. L'algorithme s'arrête car la liste des nœuds à explorer est vide.</p>
--	--	---

Pile

A
BDE
BDF
BDG
BDHI
BDH
BDJ
BD
BC
B
∅

Remarquez comment évolue la liste des nœuds à explorer :

- Les **nouveaux nœuds** sont ajoutés en fin de liste ;
- Le **nœud courant** à explorer est le dernier de cette liste.

Les nœuds explorés dès qu'ils sont ajoutés à la liste sont en **orange**.

Cette structure de données, appelée **pile**, est composée de 4 manipulations :

- « Créer » : créer la pile, au départ celle-ci est vide ;
- « Empiler » : ajouter un élément dans la pile (à la fin de celle-ci) ;
- « Dépiler » : renvoyer le prochain élément de la pile (le dernier), et le retirer,
- « Tester si la pile est vide » ;

C'est le principe du dernier arrivé, premier sorti (LIFO : Last in, First out) utilisé dans :

- La gestion des piles d'assiette,
- La gestion des CTRL+Z qui annule la dernière commande tapée,
- La gestion des erreurs sous Python : Python renvoie à l'écran l'endroit de l'erreur, mais remonte dans les différentes fonctions dans lesquelles se trouve l'erreur.



Pseudo-code

Algorithme 2 : Algorithme de parcours de graphe en profondeur itératif

Initialisation;

Initialiser la pile d'appel avec le nœud de départ;

Initialiser la liste des nœuds découverts avec le nœud de départ;

tant que la pile n'est pas vide faire

 noeud_courant ← Dépiler(pile);

pour chacun des voisins de v de noeud_courant **faire**

si v n'appartient pas à la liste noeuds_decouverts **alors**

 Empiler v à la fin de la pile ;

 Ajouter noeud_courant à la liste des nœuds découverts ;

fin

fin

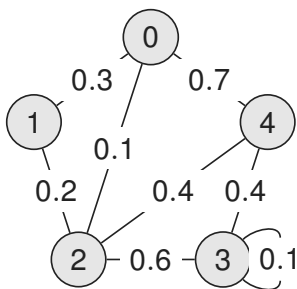
fin

return la liste des nœuds découverts



Étiquettes et pondération

- Un graphe **étiqueté** est un graphe (orienté ou non) dont les liaisons entre les sommets (arêtes ou arcs) sont affectées d'étiquettes (mot, lettre, symbole,...),
- Un graphe **pondéré** est un graphe étiqueté dont toutes les étiquettes sont des nombres réels positifs ou nuls. Ces nombres sont les poids des liaisons (arêtes ou arcs) entre les sommets,
- Le **poids** d'une chaîne (respectivement d'un chemin) est la somme des poids des arêtes (resp des arcs) qui constituent la chaîne (resp. le chemin),
- Une plus **courte** chaîne (resp. un plus court chemin) entre 2 sommets est, parmi les chaînes qui les relient (resp. les chemins qui les relient) celle (celui) qui a le poids minimum.



Le plus court chemin de 0 à 4 est 0–2–4.

Application au Q-Learning (Reinforcement)

L'objectif de cette étude est d'apprendre à une intelligence artificielle à choisir le chemin optimal afin de:

- récupérer les passagers et les déposer à leur destination,
- conduire en toute sécurité, ce qui signifie pas d'accidents,
- les conduire dans les plus brefs délais.

Nous utiliserons un environnement d'OpenAI Gym (Taxi-v3).

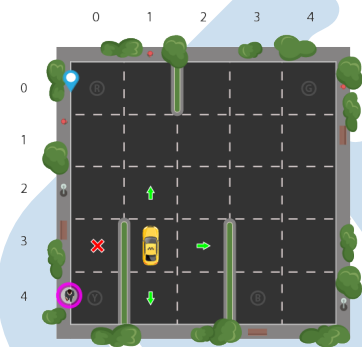
Un état est défini par (taxi_row, taxi_col, passenger_location, destination).

Il y en a $25 \times 5 \times 4 = 500$ et celui de l'image est (3,1,0,2), avec :

Passenger locations={0:'R(ed)', 1:'G(reen)', 2:'Y(ellow)', 3:'B(lue)', 4:'in taxi'}

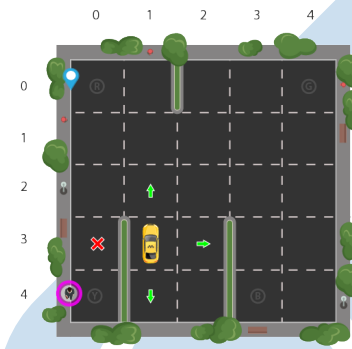
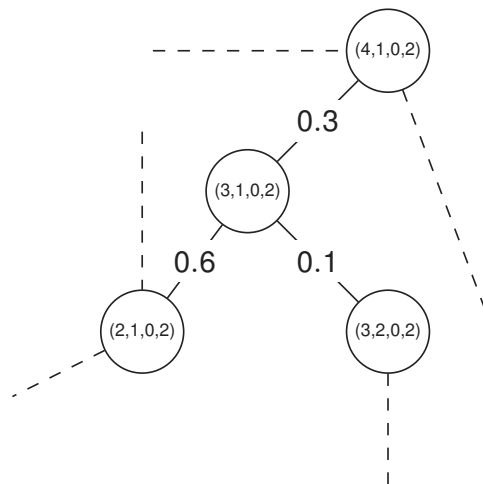
Destinations={0:'R(ed)', 1:'G(reen)', 2:'Y(ellow)', 3:'B(lue)'}
 Actions={0:'move south', 1:'move north', 2:'move east', 3:'move west',

4:'pickup passenger', 5:'drop off passenger'}



Application au Q-Learning (Reinforcement)

Un état peut alors être représenté par un sommet d'un graphe, dont les arêtes sont pondérées, et les états accessibles par des états voisins. Les coefficients de pondération sont déterminés durant la phase d'apprentissage et sont utilisés dans la phase d'utilisation afin de guider les choix du taxi.



Un extrait de ce graphe montre que le meilleur choix dans cette configuration est d'aller à l'état (2,1,0,2) c'est à dire que la voiture doit monter.

La simulation peut alors être effectuée grâce à un programme python.



Bibliographie

- Ressource UPSTI Réforme des programmes 2021 - bases des graphes
- Ressource UPSTI Réforme des programmes 2021 - parcours des graphes