Algorithmes sur les graphes

Dernière mise à jour le : 25/03/2024

Introduction

Un des premiers algorithmes qu'on doit savoir utiliser sur un graphe est celui de son parcours. Parcourir un graphe, c'est visiter ses différents sommets, afin de pouvoir opérer une action tour à tour sur eux.

Les deux algorithmes fondamentaux permettant de parcourir un graphe s'appellent :

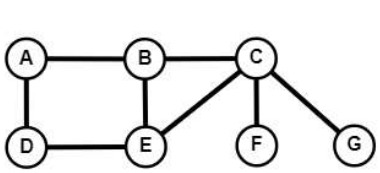
- le *parcours en profondeur* d'abord ;
- le *parcours en largeur* d'abord.

Selon les actions opérées au cours d'un parcours, on peut détecter des cycles dans le graphe, trouver le chemin le plus court entre deux sommets, calculer la distance entre deux sommets, etc.

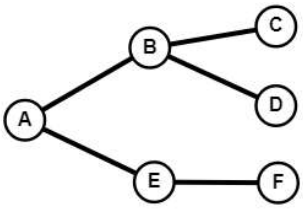
Les algorithmes sur les graphes sont très utilisés dans la vie courante, ils permettent par exemple :

- le routage des paquets de données dans un réseau ;
- de trouver le chemin le plus court entre deux villes (utilisé par les GPS) ;
- de sortir d'un labyrinthe ;
- etc.

Dans la suite, on considérera les deux graphes `g1` et `g2` suivants.



Graphe `g1`



Graphe `g2`

Ils seront représentés par listes de successeurs et implémentés par des dictionnaires.

```
g1 = {
    "A": ["B", "D"],
    "B": ["A", "C", "E"],
    "C": ["B", "E", "F", "G"],
    "D": ["A", "E"],
    "E": ["B", "C", "D"],
    "F": ["C"],
    "G": ["C"]
}

g2 = {
    "A": ["B", "E"],
    "B": ["A", "C", "D"],
    "C": ["B"],
    "D": ["B"],
    "E": ["A", "F"],
    "F": ["E"]
}
```

Les sommets d'un graphe sont les clés du dictionnaire. En particulier, on peut accéder aux voisins/successeurs d'un sommet en utilisant sa clé.

```
>>> g1["A"]  # voisins du sommet A dans g1
['B', 'D']
```

> Ces deux graphes sont disponibles sur le site *graphonline.ru* : graphe 1 et graphe 2 . On pourra en particulier leur appliquer des algorithmes.

Parcours en profondeur et en largeur

Comparaison des deux algorithmes

Ces deux algorithmes ont le même but : explorer tous les sommets atteignables d'un graphe à partir d'un sommet de départ. L'idée est d'explorer les voisins (ou successeurs) rencontrés au fur et à mesure en marquant les sommets visités pour ne pas tourner en rond.

Parcours en profondeur d'abord : à partir d'un sommet, on explore un de ses voisins (ou successeurs), et ainsi de suite. S'il n'y a plus de voisins, on revient au sommet précédent et on passe à un autre de ses voisins. Cette façon de faire implique que chaque "branche" est explorée jusqu'au bout, avant de revenir sur nos pas, d'où le nom de parcours en *profondeur*.

Parcours en largeur d'abord : à partir d'un sommet, on explore tous ses voisins (ou successeurs), puis on explore tous les voisins de ces voisins, et ainsi de suite. Le parcours balaie ainsi chaque "branche" au même rythme, d'où le nom de parcours en *largeur*.

La seule différence entre ces deux algorithmes est donc l'ordre dans lequel les voisins sont traités. Cela permet d'écrire le même algorithme pour les deux parcours, en changeant juste la collection qui stocke les sommets à visiter : une *pile* pour le parcours en profondeur et une *file* pour le parcours en largeur.

Principe de l'algorithme de parcours en profondeur

Algorithme de parcours en profondeur

- On choisit un sommet de départ
- On l'empile
- Tant que la pile n'est pas vide :
  - On dépile son sommet
  - S'il n'a pas encore été visité on le marque et on empile tous ses voisins non encore visités
  - Sinon, on ne fait rien (on passe donc directement à l'itération suivante)

En stockant les sommets encore à visiter dans une **pile**, on s'assure que ce sont les derniers sommets découverts qui vont être visités en premier (LIFO, *Last In First Out*), cela correspond au parcours en profondeur :

```
def parcours_prof(graphe, debut):
    visites = {}
    pile = [debut]
    while len(pile) > 0:
        s = pile.pop()
        if s in visites:   # si s a déjà été visité
            continue       # on passe à l'itération suivante
        visites[s] = True  # sinon l'itération en cours se poursuit
        for voisin in graphe[s]:
            if voisin not in visites:
                pile.append(voisin)
    return visites
```

On a utilisé ici un dictionnaire pour marquer les sommets (en les ajoutant dans le dictionnaire). La valeur d'un sommet visité (ici `True`) n'a pas d'importance. On aurait donc pu utiliser ici une structure de données abstraite plus simple : l'*ensemble*.

Principe de l'algorithme de parcours en largeur

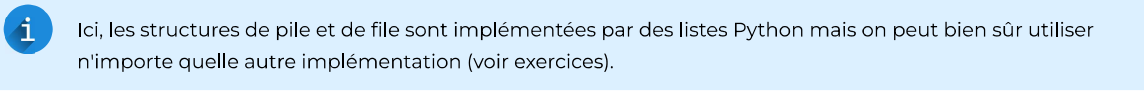
C'est simple, il suffit de remplace la pile par une file !

Algorithme de parcours en largeur

- On choisit un sommet de départ
- On l'enfile
- Tant que la file n'est pas vide :
  - On défile son premier élément
  - S'il n'a pas encore été visité on le marque et on enfile tous ses voisins non encore visités
  - Sinon, on ne fait rien (on passe donc directement à l'itération suivante)

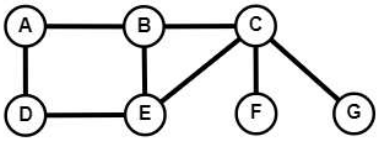
En stockant les sommets encore à visiter dans une **file**, on s'assure que ce sont les premiers sommets découverts qui vont être visités en premier (FIFO, *First In First Out*), cela correspond au parcours en largeur :

```
def parcours_larg(graphe, debut):
    visites = {}
    file = [debut]
    while len(file) > 0:
        s = file.pop(0)
        if s in visites:   # si s a déjà été visité
            continue       # on passe à l'itération suivante
        visites[s] = True  # sinon l'itération en cours se poursuit
        for voisin in graphe[s]:
            if voisin not in visites:
                file.append(voisin)
    return visites
```



Ici, les structures de pile et de file sont implémentées par des listes Python mais on peut bien sûr utiliser n'importe quelle autre implémentation (voir exercices).

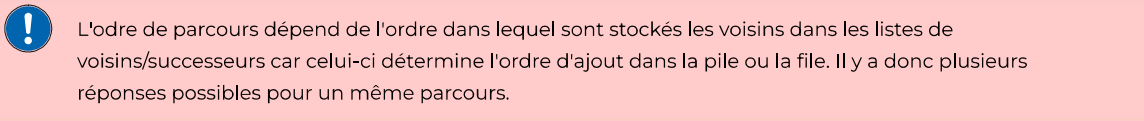
En observant l'ordre d'ajout des clés dans le dictionnaire (*valable à partir de Python 3.7 !*), on peut voir l'ordre des sommets visités par chaque parcours.




```
>>> parcours_prof(g1, "A")
{'A': True, 'D': True, 'E': True, 'C': True, 'G': True, 'F': True, 'B': True}
>>> parcours_larg(g1, "A")
{'A': True, 'B': True, 'D': True, 'C': True, 'E': True, 'F': True, 'G': True}
```

Le parcours en profondeur donne l'ordre de parcours A -> D -> E -> C -> G -> F -> B.

Le parcours en largeur donne l'ordre de parcours A -> B -> D -> C -> E -> F -> G.



L'odre de parcours dépend de l'ordre dans lequel sont stockés les voisins dans les listes de voisins/successeurs car celui-ci détermine l'ordre d'ajout dans la pile ou la file. Il y a donc plusieurs réponses possibles pour un même parcours.

Version récursive du parcours en profondeur

En réalité, le parcours en profondeur est naturellement *récursif*. En effet, un part du sommet de départ et on explore l'un de ses voisins, puis on explore l'un des voisins du voisin, ainsi de suite jusqu'à ce qu'on ne trouve plus de voisins (on arrive à un "cul-de-sac"), auquel cas on revient au sommet précédent.

On peut traduire l'algorithme de parcours en profondeur de la façon très simple suivante : si un sommet n'est pas visité, on le marque et on parcourt récursivement tous ses voisins.

Comme souvent, l'énoncé d'un programme récursif est plus concis (et plus "logique"). Voici une implémentation récursive de cet algorithme :

```
def parcours(graphe, visites, s):
    """parcours en profondeur depuis le sommet s"""
    if s not in visites:
        visites[s] = True
        for voisin in graphe[s]:
            parcours(graphe, visites, voisin)
    return visites
```

Il suffit alors de lancer le premier appel avec un dictionnaire `visites` vide, ce que fait la fonction d'interface `parcours_prof_rec` suivante.

```
def parcours_prof_rec(graphe, debut):
    return parcours(graphe, {}, debut)
```

On peut vérifier que cela fonctionne tout autant.

```
>>> parcours_prof_rec(g1, "A")
{'A': True, 'B': True, 'C': True, 'E': True, 'D': True, 'F': True, 'G': True}
```

> L'ordre des sommets visités n'est pas le même car ici c'est le premier voisin écrit dans la liste des sucesseurs (et pas encore visité) qui est exploré en premier. C'était le contraire avec la pile car les sommets étant empilés l'un après l'autre, celui en haut de la pile était le dernier écrit dans la liste de successeurs. On pourrait obtenir le même résultat si on empilait les voisins/successeurs dans l'ordre inverse.

Les algorithmes de parcours en profondeur ou en largeur permettent, selon les actions opérées sur les sommets découverts, à écrire de nouveaux algorithmes essentiels comme ceux permettant de :

- repérer la présence d'un cycle dans un graphe ;
- chercher un chemin dans un graphe

Repérer la présence d'un cycle dans un graphe

Rappelons la définition d'un *cycle* : dans un graphe non orienté, un **cycle** est une suite d'arêtes consécutives (chaîne) dont les deux sommets extrémités sont identiques.

> Le terme *cycle* n'a de sens que dans un graphe *non orienté*, c'est pourquoi on ne considère dans ce paragraphe que des **graphes non orientés**.

Principe de l'algorithme de détection de cycle

Il suffit d'adapter légèrement, au choix, l'un des deux algorithmes de parcours du graphe. Si lors du parcours on rencontre (en dépilant ou en défilant) un sommet déjà visité (marqué grâce au dictionnaire `visites`), on a trouvé un cycle ! En effet, cela signifie que ce sommet a été ajouté au moins deux fois dans la pile ou dans la file, ce qui veut dire que l'on peut l'atteindre par au moins deux sommets différents. Ces deux chemins ayant pour origine le sommet de départ du parcours, on a nécessairement un cycle.

L'algorithme est identique à celui d'un parcours en stoppant le parcours si un cycle est trouvé :

Algorithme de détection de cycle

- On choisit un sommet de départ
- On l'empile
- Tant que la pile n'est pas vide :
  - On dépile son sommet
  - S'il n'a pas encore été visité on le marque et on empile tous ses voisins non encore visités
  - Sinon, **on a trouvé un cycle et on renvoie Vrai**

Si le parcours se termine sans trouver de cycle, on renvoie Faux.

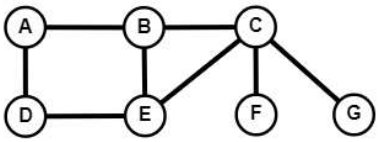
L'écriture d'une fonction repérant un cycle est donc quasiment identique à celle d'un parcours en profondeur (ou en largeur) : si on rencontre un sommet déjà visité, au lieu de passer au suivant (avec le mot-clé `continue`), il faut renvoyer `True`. Si à l'issue du parcours on n'a pas trouvé de cycle, on renvoie `False`.

La fonction `parcours_prof_cycle` qui suit renvoie `True` si et seulement si on trouve un cycle en partant du sommet `debut`.

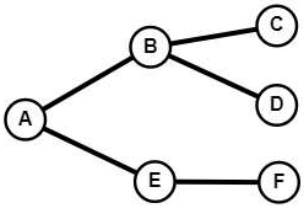
```
def parcours_prof_cycle(graphe, debut):
    """Renvoie True ssi un cycle est détecté
    dans le parcours à partir du sommet debut."""
    visites = {}
    pile = [debut]
    while len(pile) > 0:
        s = pile.pop()
        if s in visites:
            return True  # on a remplacé continue
        visites[s] = True
        for voisin in graphe[s]:
            if voisin not in visites:
                pile.append(voisin)
    return False  # on renvoie False et non plus le dictionnaire visites
```

> Le principe serait le même en utilisant un parcours en largeur.

On peut vérifier qu'un cycle est bien détecté dans le graphe `g1` mais pas dans le graphe `g2`.




```
>>> parcours_prof_cycle(g1, "A")
True
```




```
>>> parcours_prof_cycle(g2, "A")
False
```

Cas d'un graphe non connexe

Si le graphe (non orienté) est connexe, on peut tester la présence d'un cycle à partir de n'importe quel sommet de départ. En revanche, pour un graphe non connexe (et toujours non orienté), il faut s'assurer de parcourir tous ses sommets. On peut par exemple lancer la détection à partir de chaque sommet. La fonction suivante permet de faire ce travail.

```
def possede_cycle(graphe):
    # on lance le parcours à partir de chaque sommet x du graphe
    for x in graphe:
        # si on trouve un cycle à partir d'un sommet x la réponse est vrai
        if parcours_prof_cycle(graphe, x):
            return True
    return False  # sinon il n'y a pas de cycle
```

Recherche d'un chemin (ou d'une chaîne) dans un graphe

Rappelons quelques définitions :

- Dans un graphe non orienté, une **chaîne** est une séquence ordonnée d'arêtes telle que chaque arête a une extrémité en commun avec l'arête suivante.
- Dans un graphe orienté, un **chemin** désigne une séquence ordonnée d'arcs consécutifs.

> Dans la suite, on ne parlera que de chaînes ou de chemins *simples*, c'est-à-dire n'empruntant pas deux fois la même arête (ou le même arc).

En utilisant un parcours en profondeur ou en largeur, on peut trouver tous les sommets accessibles à partir d'un sommet de départ. Cela permet d'écrire très facilement un algorithme qui renvoie Vrai s'il *existe un chemin* pour aller d'un point A à un point B. Il suffit de lancer l'un des deux parcours à partir du sommet A et de regarder à la fin du parcours si le sommet B a été atteint (s'il est dans le dictionnaire `visites`).

Cependant, cet algorithme ne permet pas d'exhiber un tel chemin. Pour cela, il faut travailler un peu plus.

Principe de l'algorithme de recherche d'un chemin

Une idée est d'**utiliser différemment le dictionnaire** `visites`. Celui-ci ne servira plus à marquer (à `True`) les sommets visités, mais associera à chaque sommet, le sommet qui permet de l'atteindre pour la première fois (le premier sommet duquel il est voisin dans le parcours).

Autrement dit, dès qu'on visite un sommet, il faut l'associer à tous ses voisins (non encore visités) dans le dictionnaire `visites`. On intialisera à `None` le sommet initial dans le dictionnaire `visites`. A la fin du parcours, il suffira de "remonter" le dictionnaire du sommet de fin au sommet de début.

Voici le principe plus en détail (en utilisant un parcours en profondeur) :

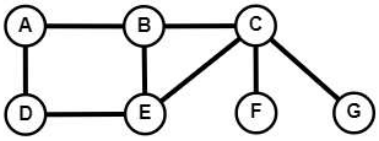
Algorithme de recherche d'un chemin

- On choisit le sommet de départ que l'on associe à `None`
- On l'empile
- Tant que la pile n'est pas vide :
  - On dépile son sommet `s`
  - (On ne le marque plus)
  - On empile tous ses voisins non encore visités et on les associe à la valeur `s` dans le dictionnaire `visites`.

La fonction `parcours_prof_ch` suivante permet de construire ce dictionnaire `visites`. Elle est basée sur un parcours en profondeur mais s'écrirait de la même manière avec un parcours en largeur (en remplaçant la pile par une file bien sûr).

```
def parcours_prof_ch(graphe, debut):
    visites = {debut: None}  # on associe le sommet de départ à None
    pile = [debut]
    while len(pile) > 0:
        s = pile.pop()
        # (on ne marque plus les sommets non visités)
        for voisin in graphe[s]:
            if voisin not in visites:
                pile.append(voisin)
                visites[voisin] = s  # on associe s à tous les voisins de s pas encore visités
    return visites
```

On peut lancer le parcours sur le graphe `g1` à partir du sommet A.




```
>>> parcours_prof_ch(g1, "A")
{'A': None, 'B': 'A', 'D': 'A', 'E': 'D', 'C': 'E', 'F': 'C', 'G': 'C'}
```

Pour trouver le chemin entre le sommet A et le sommet E, il faut "remonter" les sommets à partir de G :

- On cherche G : il est associé à la valeur C donc on a pu atteindre G à partir de C ;
- On cherche C : atteint à partir de E ;
- On cherche E : atteint à partir de D ;
- On cherche D : atteint à partir de A.

On a terminé puisqu'on a fini par tomber sur A. Un chemin possible entre A et G est donc : A -> D -> E -> C -> G.

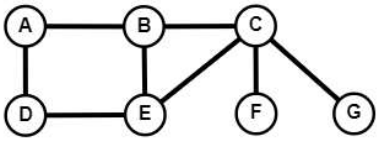
On était sûr de remonter jusqu'à A puisque G, se trouvant dans le dictionnaire `visites`, était nécessairement atteignable en partant de A. Si un sommet ne se trouve pas dans `visites`, on sait alors qu'il n'existe pas de chemin vers ce sommet en partant de A.

La fonction `chemin_prof(graphe, debut, fin)` permet d'effectuer ce travail en renvoyant une liste `ch` contenant les sommets du chemin trouvé entre les sommets `debut` et `fin`. Elle ajoute les sommets à `ch` au fur et à mesure de la remontée jusqu'à tomber sur celui de départ et renvoie ensuite cette liste qui a été préalablement renversée pour obtenir les sommets dans le bon ordre.

```
def chemin_prof(graphe, debut, fin):
    visites = parcours_prof_ch(graphe, debut)
    if fin not in visites:
        return None
    s = fin
    ch = [s]  # on ajoute le sommet de fin à partir duquel commence la "remontée"
    while s != debut:  # tant qu'on ne trouve pas le sommet de départ
        s = visites[s]  # on remonte en passant au sommet associé
        ch.append(s)  # qu'on ajoute au chemin
    ch.reverse()  # ne pas oublier de renverser la liste pour renvoyer le chemin dans le bon ordre
    return ch
```

On peut vérifier qu'elle renvoie le chemin trouvé à la main entre A et G.

```
>>> chemin_prof(g1, "A", "G")
['A', 'D', 'E', 'C', 'G']
```



On constate que le chemin n'est pas le plus court (en nombre d'arêtes) car on peut faire mieux : A -> B -> C -> G. Peut-on trouver le chemin le plus court ? La réponse est oui !

Recherche d'un plus court chemin

En faisant la même recherche à partir d'un parcours en largeur, on obtiendrait un plus court chemin (en nombre d'arêtes/arcs).

En effet, l'algorithme de recherche en largeur explore d'abord les sommets à une distance 1 du sommet de départ, puis ceux à distance 2 du sommet de départ, etc. Ainsi, chacun des autres sommets est atteint en passant par un nombre minimal d'arêtes (ou arcs), ce qui assure de trouver un plus court chemin (en nombre d'arêtes/arcs) vers chacun des autres sommets.

```
# on remplace la pile par une file
def parcours_larg_ch(graphe, debut):
    visites = {debut: None} # on associe le sommet de départ à None
    file = [debut]
    while len(file) > 0:
        s = file.pop(0)
        # (on ne marque plus les sommets non visités)
        for voisin in graphe[s]:
            if voisin not in visites:
                file.append(voisin)
                visites[voisin] = s  # on associe s à tous les voisins de s pas encore visités
    return visites

# exactement la même fonction que chemin_prof (en remplacant juste l'appel à la première ligne)
def chemin_larg(graphe, debut, fin):
    visites = parcours_larg_ch(graphe, debut)
    if fin not in visites:
        return None
    s = fin
    ch = [s]
    while s != debut:
        s = visites[s]
        ch.append(s)
    ch.reverse()
    return ch
```

```
>>> chemin_larg(g1, "A", "G")
['A', 'B', 'C', 'G']
```

Nous venons de voir comment utiliser un parcours en largeur pour trouver le plus court chemin, en nombre d'arêtes/arcs, entre deux sommets d'un graphe.

Distances entre les sommets

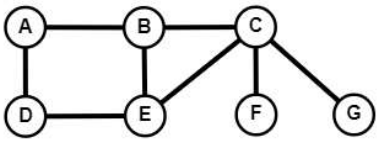
En modifiant le rôle du dictionnaire `visites` dans la recherche de chemin *du parcours en largeur*, on peut très facilement trouver la distance du sommet de départ à tous les autres. On va utiliser `visites` pour associer à chaque sommet la distance qui le sépare du sommet d'origine. La distance d'un sommet découvert est celle du sommet d'où on vient, plus 1 !

En initialisant une distance égale à 0 pour le sommet de départ on obtient, en changeant uniquement une ligne, les distances entre chaque sommet et le sommet de départ.

```
def parcours_larg_distance(graphe, debut):
    visites = {debut: 0} # debut est à distance 0 de lui-même
    file = [debut]
    while len(file) > 0:
        s = file.pop(0)
        # (on ne marque plus les sommets non visités)
        for voisin in graphe[s]:
```

```
            if voisin not in visites:
                file.append(voisin)
                visites[voisin] = visites[s] + 1  # la distance est celle de s (d'où l'on vient) + 1
    return visites
```

On peut vérifier les distances entre le sommet A et les autres dans le graphe `g1`.




```
>>> parcours_larg_distance(g1, "A")
{'A': 0, 'B': 1, 'D': 1, 'C': 2, 'E': 2, 'F': 3, 'G': 3}
```

Les notions de plus *court* chemin ou de *distance* que l'on vient de voir, correspondent au nombre d'arêtes/arcs séparant les sommets. On suppose donc que chaque arête à le même coût dans ce calcul. Autrement dit, nos algorithmes s'appliquent sur des graphes *non pondérés* (ou des graphes dans lesquels toutes les arêtes ont le même poids).

Avec des graphes *pondérés*, les arêtes n'ont pas toutes le même coût, ce qui redéfinit cette notion de *distance*. C'est le cas de la plupart des graphes rencontrés dans la vie courante. On ne peut donc plus appliquer l'algorithme de plus court chemin étudié. Il en existe heureusement d'autres : le plus connu d'entre eux est l'*algorithme de Dijskstra*.

Algorithme de Dijskstra : pour trouver le plus court chemin *pondéré*

C'est l'algorithme qui permet de trouver le chemin le plus court (en km par exemple) entre deux villes ou pour router les paquets sur Internet, ou encore pour sortir d'un labyrinthe.

> Il ne s'applique que sur des graphes pondérés avec des poids positifs.

Cet algorithme n'est pas au programme (a priori), il est un peu plus compliqué à écrire mais il est très simple à comprendre et à appliquer. La vidéo suivante présente l'algorithme de Dijkstra.

Bilan

- Les algorithmes de parcours de graphe permettent de visiter tous les sommets d'un graphe. On a le choix entre un **parcours en profondeur** d'abord ou **un parcours en largeur** d'abord.
- La différence entre ces deux algorithmes est l'ordre dans lequel on visite tous les sommets. On peut donc les écrire de manière similaire en changeant juste la structure de données qui stocke les sommets à visiter : une *pile* pour poursuivre le parcours avec les derniers sommets rencontrés (parcours en profondeur) ou une *file* pour poursuivre le parcours avec les premiers sommets rencontrés (parcours en largeur).
- Pour ne pas tourner en rond, il faut *marquer les sommets visités* au cours du parcours. On a utilisé un dictionnaire pour faire ce travail.
- En modifiant le rôle et la façon d'utiliser ce dictionnaire, on peut facilement adapter les algorithmes de parcours pour repérer la présence d'un cycle, pour trouver un chemin entre deux sommets, voire le plus court d'entre eux.
- L'algorithme de parcours en largeur assure de trouver un plus court chemin dans un graphe *non pondéré* car il explore les différents sommets dans l'ordre de leur distance à celui de départ (ceux à distance 1, puis ceux à distance 2, etc.).
- Pour des graphes non pondérés, l'algorithme de Dijkstra (hors programme) permet de trouver le plus court chemin entre deux sommets dans un graphe *pondéré* : c'est celui utilisé pour router les paquets dans un réseau, pour nous guider avec un GPS, pour sortir d'un labyrinthe...

Références :

- Equipe pédagogique DIU EIL, Université de Nantes.
- Ressource Eduscol : Structures de données, paragraphe 1.
- Livre *Spécialité Numérique et sciences informatiques : 24 leçons avec exercices corrigés - Terminale*, éditions Ellipses, T. Balabonski, S. Conchon, J.-C. Filliâtre, K. Nguyen pour l'exemple du graphe `g1` et pour la version récursive du parcours en profondeur : http://www.nsi-terminale.fr/.

Germain BECKER, Lycée Mounier, ANGERS

Voir en ligne : info-mounier.fr/terminale_nsi/algorithmique/algorithmes-graphes