NSI Terminale - Algorithmes

Les graphes - Parcourir, chemins, cycles

qkzk

2020/05/02

Algorithmes sur les graphes

Nous allons présenter différents algorithmes sur les graphes :

- **parcours en largeur d'abord,**
- **parcours en profondeur d'abord,**

Et deux algorithmes qui utilisent les parcours :

- **recherche d'un chemin entre deux sommets,**
- **détection de la présence d'un cycle dans un graphe.**

Les applications sont nombreuses :

- Si un problème s'exprime avec un graphe, le *parcourir* permet de trouver une solution.
- Déterminer un chemin est ce qu'on fait pour trouver la sortie d'un labyrinthe.
- Détecter un cycle dans un graphe est une étape préalable à de nombreux algorithmes (choix d'une route optimale sur une carte, par exemple) qui exigent parfois *qu'il n'y ait pas de cycle* dans le graphe.

Parcourir un graphe simple

Nous avons déjà vu comment parcourir un arbre. Le principe est identique.

Une nuance importe toutefois, dans un arbre, il est impossible de passer plusieurs fois par le même noeud, dans un graphe ce risque est majeur.

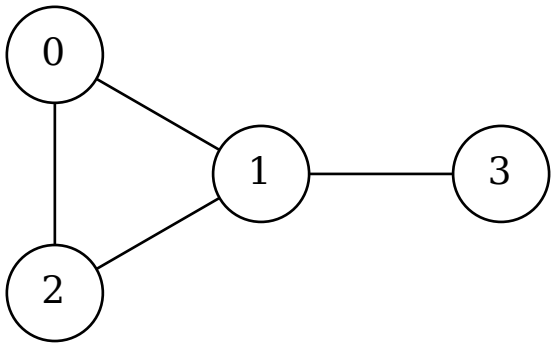
Il faut trouver un moyen de *marquer* les sommets rencontrés afin de ne plus les explorer.

Autre nuance importante, plusieurs arêtes peuvent mener à un même sommet, lors d'un parcours, on cherche à visiter *chaque sommet*, pas chaque arête.

Parcourir un graphe :

> C'est visiter chaque sommet du graphe et, généralement, faire quelque chose. On choisit généralement un point de départ, appelé **Source** et on s'arrête quand on a atteint la **Destination** ou quand on a visité tous les sommets.

Exemple



Considérons le graphe précédent.

Parcours en largeur d'abord :

on choisit de visiter **les voisins** d'un sommet avant d'aller plus profondément dans le graphe.

Partant de 0 l'ordre est :

0, 1, **2, 3**

Lors d'un parcours en *largeur* on utilise une *file* qu'on remplit au fur et à mesure.

Parcours en profondeur d'abord :

on choisit de visiter en premier **le dernier voisin rencontré**

Partant de 0 l'ordre est :

0, 1, **3, 2**

Lors d'un parcours en *profondeur* on utilise une *pile* qu'on remplit au fur et à mesure.

Algorithme : parcours en largeur dans un graphe simple

```
source                              (un noeud du graphe)
file     : [Source]                 (une file)
drapeaux : [-1, -1, etc., -1]       (un tableau avec -1 pour chaque indice de
                                    sommet)
```

Dans le tableau `drapeaux`, si un sommet est d'indice 2,

`drapeaux[2] = -1` signifie qu'on ne l'a **pas encore ajouté** à la file.
`drapeaux[2] = 0` signifie qu'on l'a **déjà ajouté** à la file.

Parcours en largeur :

```
Changer le drapeau de la source à 0.

Tant que la file n'est pas vide faire :
    courant = défiler()
    Pour chaque voisin de courant qui n'a pas déjà été visité,
        l'ajouter à la file.
        Changer leurs drapeaux à 0.
    visiter courant. # c'est ici qu'on fera généralement un travail.
```

Exemple

Disons que notre action “visiter” est d’afficher le numéro du sommet courant.

Sur le graphe précédent :

```
0. File = [0], drapeaux = [0, -1, -1, -1]
Début de la boucle.
1. On défile : courant = 0, File = []
    Voisins de 0 : 1 et 2 qu'on ajoute à la file. File = [1, 2]
      On change leurs drapeaux : drapeaux = [0, 0, 0, 1]
    On affiche 0
2. On défile, courant = 1. File = [2]
    voisins de 1 : 0, 2, 3. On ajoute 3 à la file (son drapeau vaut -1)
      File = [2, 3]
      On change le drapeau de 3 : drapeaux = [0, 0, 0, 0]
    On affiche 1
3. On défile, courant = 2, File = [3]
    voisins de 2 : 0, 1. Tous les drapaux valent 0, on n'ajoute aucun sommet.
    On affiche 2
4. On défile, courant = 3. File = []
    Tous les voisins de courant ont un drapeau valant 0.
    On affiche 3

La file est vide et la boucle est terminée.

L'affichage dans la console aura donné : 0, 1, 2, 3
```

C’est bien un parcours en largeur d’abord.

On explore tous les voisins avant d’avancer d’un niveau.

Algorithme : parcours en profondeur dans un graphe simple

La seule différence est qu’on utilise une *pile*.

```
source                              (un noeud du graphe)
pile     : [Source]                 (une pile)
drapeaux : [-1, -1, etc., -1]       (un tableau avec -1 pour chaque indice de
                                    sommet)
```

`drapeau = -1` : pas encore ajouté à la pile
`drapeau = 0` : déjà ajouté à la pile

Parcours en profondeur :

```
Changer le drapeau de la source à 0.

Tant que la pile n'est pas vide faire :
    courant = dépiler()
    Pour chaque voisin de courant qui n'a pas déjà été visité,
        l'ajouter à la pile.
        Changer leurs drapeaux à 0.
    visiter courant. # c'est ici qu'on fera généralement un travail.
```

Exemple

Sur le graphe précédent :

```
0. Pile = [0], drapeaux = [0, -1, -1, -1]
Début de la boucle.
1. On dépile : courant = 0, Pile = []
    Voisins de 0 : 1 et 2 qu'on ajoute à la pile. Pile = [1, 2]
      On change leurs drapeaux : drapeaux = [0, 0, 0, 1]
    On affiche 0
2. On dépile, courant = 1. Pile = [2]
    voisins de 1 : 0, 2, 3. On ajoute 3 à la pile (son drapeau vaut -1)
      Pile = [2, 3]
      On change le drapeau de 3 : drapeaux = [0, 0, 0, 0]
    On affiche 1

ATTENTION C'EST ICI QUE ÇA CHANGE : DÉPILER = SORTIR LE DERNIER

3. On dépile, courant = 3, Pile = [2]
    voisins de 3 : Tous les drapaux valent 0, on n'ajoute aucun sommet.
    On affiche 3
4. On dépile, courant = 2. Pile = []
    Tous les voisins de courant ont un drapeau valant 0.
    On affiche 2

La pile est vide et la boucle est terminée.

L'affichage dans la console aura donné : 0, 1, 3, 2
```

C'est bien un parcours en profondeur d'abord.

On explore un chemin le plus profondément possible avant d'avancer.

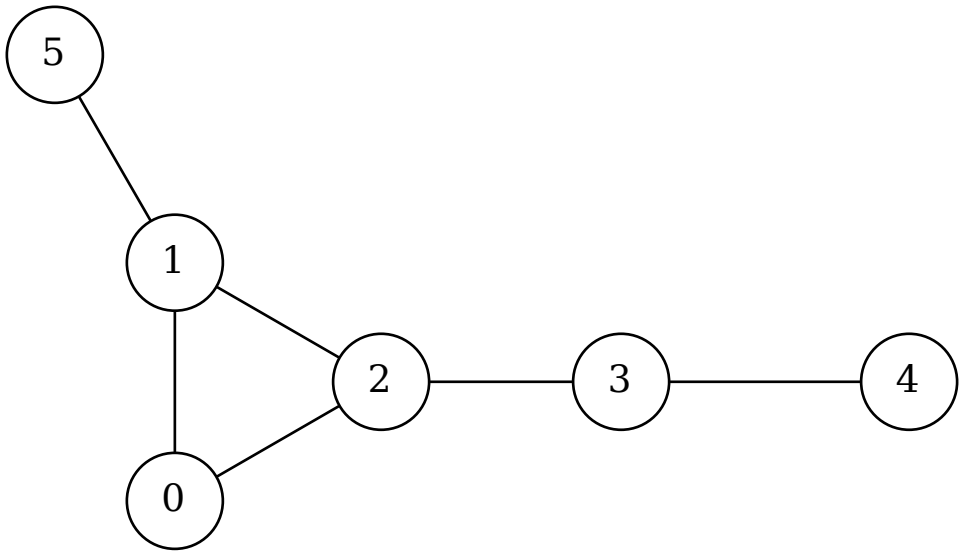
Cas des graphes orientés

Si le graphe est *orienté*, les algorithmes sont identiques.

On doit simplement ajouter les *successeurs* et non plus les voisins à la pile (ou à la file).

Déterminer un chemin dans un graphe simple

Cette fois on souhaite savoir quels sommets emprunter pour rejoindre une destination depuis une source.



Dans l'exemple ci-dessus on souhaite se rendre de la **source 0** à la **destination 4**.

Les deux chemins possibles sont :

0, 1, 2, 3, 4 et 0, 2, 3, 4.

On utilise généralement *un parcours en profondeur* lorsqu'on cherche à construire un chemin.

Cela peut sembler surprenant, les deux algorithmes étant très proches, mais la raison est qu'on a généralement des informations supplémentaires qui aident à s'orienter ou à choisir le prochain sommet à visiter.

Le principe est le même, on doit simplement conserver une information supplémentaire : d'où venons-nous ?

Algorithme : détermination d'un chemin dans un graphe simple.

Première étape : parcourir

On entretien un dictionnaire des visites, qui permettra de construire un chemin

```
Fonction Parcours en profondeur(source, destination)

  prochains     = [source]              (une pile)
  prédécesseurs = {source: Vide}        (un dictionnaire)

  tant que la pile n'est pas vide :
    On dépile courant.
    pour chaque voisin de courant
      Si voisin n'est pas déjà dans le dictionnaire des prédécesseurs
          On l'ajoute au dictionnaire avec comme valeur "courant"
          prédécesseurs[voisin] = courant
          On empile "voisin" dans la pile des prochains

          Si courant == destination, on arrête la boucle.

  On retourne à la fin le dictionnaire des visites
```

Deuxième étape, créer le chemin depuis le dictionnaire des visites.

```
Fonction créer un chemin (prédécesseurs, source, destination)
  Si destination n'est pas dans le dictionnaire prédécesseurs :
      il n'est pas possible d'atteindre la destination et on retourne None

  Sinon:
    on initialise le chemin DEPUIS LA FIN
    chemin = [destination]

    pred = destination
    Tant que pred != source :
      On attribue à predecesseur sa valeur dans le dictionnaire :
        pred = prédécesseurs[pred]

      on l'ajoute au DEBUT du chemin
      chemin = [pred] + chemin

  On retourne chemin
```

Remarque La source est forcement dans le dictionnaire des prédécesseurs, donc la boucle s'arrête toujours.

Exemple

Sans détailler toutes les étapes voici ce qu'on obtient sur l'exemple :

Le parcours en profondeur visite les sommets dans l'ordre suivant :

0, 1, 5, 2, 3, 4

Les prédécesseurs sont alors :

```
{0:None, 1:0, 2:0, 5:1, 3:2, 4:3}
```

La fonction créer un chemin remplit alors le chemin comme ceci :

```
chemin = [4], prédécesseur de 4 : 3
chemin = [3, 4], prédécesseur de 3 : 2
chemin = [2, 3, 4], prédécesseur de 2 : 0
chemin = [0, 2, 3, 4], 0 est la source, on a terminé.
```

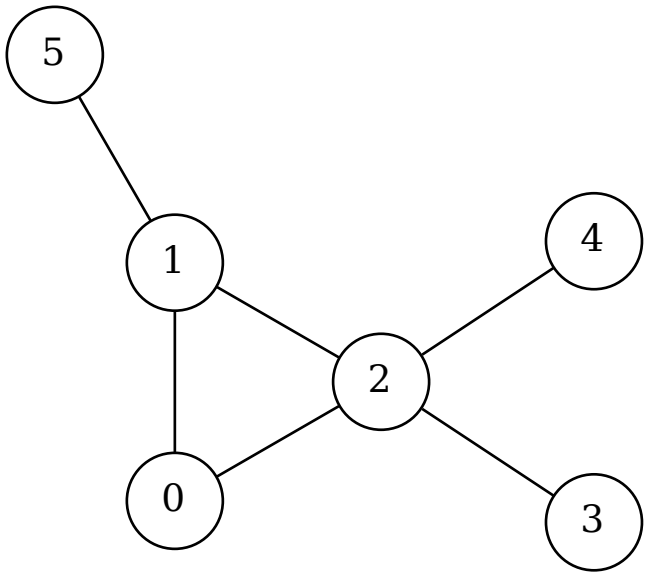
Elle retourne alors : `[0, 2, 3, 4]`

Recherche de la présence de cycle dans un graphe simple

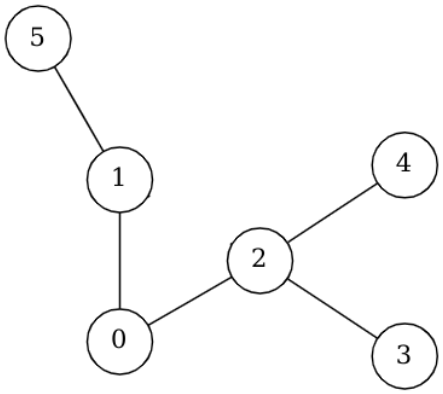
Un cycle est un chemin dont la source et la destination sont égales.

C'est un chemin qui revient sur lui même.

Par exemple le graphe ci-dessous contient un cycle `[0, 1, 2]` :



Si on enlève simplement l'arête entre (1, 2) on obtient un graphe qui n'a plus de cycle :



On souhaite créer une fonction qui réponde "Vrai" pour le premier graphe (il a un cycle) et "Faux" pour le second: il n'en a pas.

Algorithme : présence d'un cycle dans un graphe

Il existe beaucoup de variantes. Nous n'en présentons qu'une.

On utilise un parcours en largeur (en profondeur c'est pareil).

La différence est le **tableau des drapeaux**.

Cette fois, lorsqu'on dépile, on ajoute la règle suivante :

- **lorsqu'on dépile un élément, on passe le drapeau à 1.**

Et lorsqu'on cherche à empiler les voisins, on ajoute la règle suivante :

- **si un voisin rencontré a un drapeau à 0, c'est qu'il y a un cycle.**

Algorithme complet

```
source                              (un noeud du graphe)
file     : [Source]                 (une file)
drapeaux : [-1, -1, etc., -1]       (un tableau avec -1 pour chaque indice de
                                    sommet)
```

Dans le tableau `drapeaux`, si un sommet est d'indice 2,

`drapeaux[2] = -1` signifie qu'on ne l'a **pas encore ajouté** à la file.
`drapeaux[2] = 0` signifie qu'on l'a **déjà ajouté** à la file mais pas encore **visité**.
`drapeaux[2] = 1` signifie qu'on l'a **déjà visité** le sommet.

Parcours en largeur :

```
Fonction Contient un cycle (graphe)

Choisir un sommet (n'importe lequel) et l'ajouter à la file.

Tant que la file n'est pas vide faire :
    courant = défiler()
    passer le drapeau de courant à 1.

    Pour chaque voisin de courant :
      Si son drapeau vaut 0:
        On a déjà rencontré ce sommet ! Il y a un cycle.
        Cycle_present = Vrai
      Si son drapeau vaut -1 :
        l'ajouter à la file.
        Changer son drapeaux en 0.

Retourner Cycle_present
```

Remarque : si le drapeau du voisin vaut 1, inutile de repasser par là.

Exemples détaillés

Graphe n°1

```
file = [0], drapeaux = [-1, -1, -1, -1, -1, -1], Cycle_present = faux

1. courant = 0. Voisins = 1, 2. drapeaux = [1, 0, 0, -1, -1, -1]. File = [1, 2]
2. courant = 1. Voisins = 2, 5.
    Le drapeau de 2 vaut 0 !!! Il y a un cycle.
    Cycle_present = Vrai
    ... le parcours se continue ...

On retourne Vrai
```

Graphe n° 2

```
file = [0], drapeaux = [-1, -1, -1, -1, -1, -1], Cycle_present = Faux

1. courant = 0. Voisins = 1, 2.    drapeaux = [1, 0, 0, -1, -1, -1]. File = [1, 2]
2. courant = 1. Voisins = 0, 5.    drapeaux = [1, 1, 0, -1, -1, 0].  File = [2, 5]
3. courant = 2. Voisins = 0, 3, 4. drapeaux = [1, 1, 1, 0, 0, 0].    File = [5, 3, 4]
4. courant = 5. Voisins = 1.       drapeaux = [1, 1, 1, 0, 0, 1].    File = [3, 4]
5. courant = 3. Voisins = 2.       drapeaux = [1, 1, 1, 1, 0, 1].    File = [4]
```

`6. courant = 4. Voisins = 2.  drapeaux = [1, 1, 1, 1, 1, 1].  File = []`

`On retourne Faux`

À aucun moment la variable `Cycle_present` n'a changé d'état.

Compléments

Preuve des algorithmes

Parcours

La pile / file qu'on remplit au fur et à mesure reçoit bien chaque sommet du graphe.

On lui retire un élément à chaque étape de la boucle. Elle ne peut recevoir deux fois le même élément. Donc l'algorithme se termine.

Ensuite, comme elle reçoit chaque élément une fois et qu'on visite chaque élément, on visite tous les sommets.

Présence d'un cycle

Si on ne rencontre jamais de sommet avec 0 comme drapeau, c'est qu'on a déjà visité chaque sommet avant d'y revenir. On ne repasse donc jamais par un sommet visité, comme s'il était un nouveau voisin. Il n'y a donc pas de cycle.

Si on rencontre un sommet avec le drapeau 0, il figure donc deux fois dans la pile / file (sinon son drapeau serait passé à 1).

Il existe donc un chemin qui part de ce point et revient à ce point sans reculer. C'est donc un cycle.

Complexité des algorithmes

La complexité de ces algorithmes est la même pour tous.

Ils reposent tous sur un parcours en largeur ou en profondeur.

Dans le pire des cas on visite chaque sommet en suivant chaque arête. La complexité est donc $O(|E| + |V|) (par exemple un arbre).

où, rappelons, V est l'ensemble des sommets, E est l'ensemble des arêtes. et $|K|$ désigne le nombre d'éléments d'un ensemble K.