chap 4 : graphes

Léo SAMUEL

23 décembre 2020

1 Vocabulaire

Un graphe est formé de :
— un ensemble, appelé ensemble des sommets (Noté S ou V)
— un sous-ensemble de S^2, appelé ensemble des arrêtes (Noté E)

Exemple 1 *$S = 1, 2, 3, 4$ et $A = (1, 2), (1, 3), (2, 1)$ Alors (S, A) est un graphe. Représentation graphique :*

Soit $k = (S, A)$ un graphe.
— Un chemin est une suite $(s_0, ..., s_n) \in S^{n+1}$ tq $\forall i \in [\![0, n[\![, (s_i, s_{i+1} \in A$
— Deux sommet s,$t \in S^n$ sont dits reliés lorsqu'il existe un chemin de s à t.
— La longueur d'un chemin est le nombre d'arretes empruntées
— $\forall (s, t)) \in S^2)$ deux sommets reliés. On note $d(s, t)$ la longueur du plus court chemin de s à t.
— On dit que G est non orienté lorsque $\forall (s, t) \in S^2, (s, t) \in A \equiv (t, s) \in A$.
— Supposons G non-orienté
  — on dit que G est connexe lorsque tous ses sommets sont reliés
  — $\forall s \in S$, on appelle composant connexe de s l'ensemble des sommets accessible depuis s

Propriété 1 *Supp G connexe et non orienté. La distance vérifie :*
— *$\forall (s, t) \in S^2, d(s, t) = d(t, s)$*
— *$\forall (s, t, u) \in S^3, d(s, t) \leq d(s, u) + d(u, t)$ (Inégalité triangulaire)*
— *$\forall (s, t) \in S^2, d(s, t) = 0 \equiv s = t$*
— *si G est orienté, les deux derniers points restent valides*
Si G n'est pas connexe, on pose $\forall (s, t) \in S^2$ non connectés : $d(s, t) = +\infty$

Démonstration 1 *Soit $(s, t) \in S^2$, notons $a = d(s, t)$ et $b = d(t, s)$. Montrons que $a = b$.
Soit γ un plus court chemin de s à t. Il est de longueur a. Soit $s_o, ..., s_a$ les sommets traversé par γ.*

*$(s_a, s_{a-1}, ..., s_o)$ est aussi un chemin. Notons le γ^T. Sa longueur est a.
Donc il existe un chemin de longueur a de t vers s. Donc $d(t, s) \leq l$, cad $b \leq a$.
Puis avec le même raisonement, on a $a \leq b$*

Soit $(s, t, u) \in S^3$. Soit γ_1 un chemin de s à t et γ_2 un chemin de t à s.

*Notons @ la concatenation des chemin et $|.|$ la longueur.
$\gamma_1 @ \gamma_2$ est un chemin de s à u de longueur $|\gamma_1| + |\gamma_2|$
Donc il existe un chemin de s à u de longueur $|\gamma_1| + |\gamma_2|$.
Donc $d(s, u) \leq |\gamma_1| + |\gamma_2| = d(s, t) + d(t, u)$*

*Soit $(s, t) \in S^2$:
Si $s = t$, soit $\gamma = (s)$. γ relie s à t et sa longueur est 0.*

Si $d(s, t) = 0$. Soit γ un plus court chemin de s à t. $|\gamma| = 0$. Donc le sommet de départ est le sommet d'arrivé donc $s = t$

2 Implémentation

Soit G un graphe et (S, E) ses composants
Soit $n = |S|$.
On supp dans la suite que $S = [\![0, n[\![$
On utilise principalement deux méthodes pour enregistrer G.

- *Matrice d'adjacence : C'est la matrice $M \in M_n(N)$ tq $\forall (i,j) \in [\![0, n[\![^2, M_{i,j} = (1 si (i,j) \in A)(0 sinon)$*
- *Tableau de liste d'adjacences : C'est le tableau g de longueur n tq $\forall i \in [\![0, n[\![$. $g.(i)$ contient la liste des voisins de i*

```
type graphe1 = int array array;;
type graphe2 = int list array;;
```

3 Parcours d'un graphe

Souvent, on a besoin de parcourir les sommets de proche en proche à partir d'un sommet de départ.

3.1 Vocabulaire et invariants de boucle

- *Un sommet est dit *blanc* s'il n'est pas découvert*
- *Un sommet est dit *noir* s'il a été traité*
- *Un sommet est dit *gris* s'il est découvert mais pas traité (Sommet alors à traité)*
- *(V N) : Les voisins des sommet noir sont noirs ou gris*
- *(V G) : Tout sommet gris a au moins un voisin noir*

*Les seuls changement de couleurs autorisés sont de *blanc vers gris* et de *gris vers noir**

NB : Pour que le programme termine, il faudra éviter de revenir à un noir.

Consequences des invariants de boucle :

Lemme 1 *On suppose (V N) et (V G) vérifiés. On suppose que l'algo a aussi satisfait à (C C). On suppose qu'il n'y a plus de sommet gris.*
Alors les sommets blancs sont déconnectés des noirs cad : $\nexists$ des chemins d'un noir vers un blanc

Démonstration 2 *Supposons qu'il existe un chemin γ tq en notant $n = |\gamma|$ et $s_0, ...s_n$ ses sommets, s_0 est noir et s_n est blanc. Soit $i = \max\{k \in [\![0, n]\!], s_k noir\}$. On a $i \leq n$ car s_n est blanc. Donc s_{i+1} existe, et est différent de noir. Donc s_{i+1} est blanc car pas de gris par hypothese. Donc (V N) n'est pas vérifiée en s_i et s_{i+1}*

Lemme 2 *On suppose (V N), (V G), (C C). Notons D l'ensemble des gris ou noir au debut de l'algorithme. Alors tout sommet noir ou gris est relié à D $(*)$*

Démonstration 3 *Notons $(*)$ la propriété et verifions que c'est un invariant de boucle.*
Initialisation : Soit s un sommet noir ou gris au debut de l'algo alors $s \in D$. Prendre $\gamma = (s)$ de longueur 0 il relie s à l.*
Heredité : Supposons $(*)$ vraie à un instant de l'algo. Effectuant une étape qui respecte (V N), (V G), et (C C).*
S'il y a un changement de couleurs de gris vers noir, l'ensemble des sommets noirs ou gris n'est pas changé donc $(*)$ reste vrai.
S'il y a un changement blanc vers gris, Soit s le sommet concerné. Par (V G), s a un voisin noir,disons t. Par hypothese de récurrence, t est relié à l. Alors s est relié à l

On suppose qu'il n'y a plus de (V N), (V G), (C C). On suppose qu'au début de l'algo, $N = \emptyset$ et un seul sommet est gris, notons le s_d. On suppose que à la fin, $G = \emptyset$

Alors à la fin de la boucle, $N =$ (la composante connexe de s_d)

Rappel : La composante connexe de s_d est l'ensemble des sommets reliés a s_d

C'est aussi la plus petite composante connexe de G contenant s_d

Démonstration 4 *Montrons que $N \subset CC$ de s_d. Soit $s \in N$, par le lemme 2, s est rellié à s_d*
Montrons que $N \not\subset CC$ de s_d. Soit s relié a s_d, par le lemme 1, $s \notin B$ et $G = \emptyset$ alors $s \in N$

Ainsi un algo vérifiant nos 3 props permet de trouver la composante connexe de s_d. Ce sera notre premier exemple. Une simple modification permettra de trouver un chemin de s_d vers un autre sommet s_a. Puis un plus court chemin.

3.2 Squelette de programme impératif

3.2.1 Algorithme

Entrée :

- *Un Grapge (S, A)*
- *Un sommet $s_d \in S$*

```
Créer 3 ensembles $N$,$G$ et $B$
$N$ <- $\emptyset$
$B$ <- S
$G$ <- $\emptyset$

Peindre $s_d$ en gris

Tant que $G \not= \emptyset$:

    extraire un sommet $s$ de $G$
    Faire quelque chose
    Peindre $s$ en noir

    $\forall t$ voisin de $s$:
        Si $t \in B$, le peindre en gris
Renvoyer le resultat
```

3.2.2 Les invariants de boucle

- *(V N) est verifié*
- *(C C) est vérifié*
- *(V G) est vérifié*

3.2.3 En pratique

Comment enregistrer N,G,B ?

En général,

- *B n'est pas enregistré. Ce sont les sommets ni noirs, ni gris*
- *N : Un tableau "$deja_v u$"$tq \forall i \in S$, $deja_v u.(i) <=> i \in N$*
- *G : ça dépend de l'ordre dans lequel on veut traiter les sommets*

3.2.4 En autorisant les doublons dans G

Il est souvent plus pratique et parfois obligatoire d'utiliser à la place de G une structure qui autorise les doublons.
Exemple : $(1,4,0,2,4)$ *Après avoir traité* 4*, celui-ci devient noir, et la file contient donc des noirs.*

Ainsi :

- *La structure utilisée ne s'appellera plus G mais aVisiter*
- *Quand on sort un sommet de aVisiter, il faut vérifier qu'il est gris*
  Alors l'algo devient

```
Créer 3 ensembles $N$,$G$ et $B$
$N$ <- un tab de |S| bool initialement faux
aVisiter <- Une structure contenant initialement $s_d$
Peindre $s_d$ en gris
Tant que $G \not= \emptyset$:
    extraire un sommet $s$ de Avisiter
    Si Non N.(s):
        Faire quelque chose
        Peindre $s$ en noir
        $\forall t$ voisin de $s$:
            Si NON N.(t):
                Mettre t dans aVisiter
Renvoyer le resultat
```

Exemple 2 *Calcul de composante connexe. Avec pour aVisiter une file d'attente*

```
let composante_connexe_largeur g sd=
let n= Array.length g in
let deja_vu= Array.make n false in
let a_Visiter= Queue.create () in
Queue.add sd a_Visiter;
let rec visite_voisins = function
    | [] -> ()
    | t::q -> Queue.add t a_Visiter;
            visite_voisins q
in
while  not (Queue.is_empty a_Visiter) do
    let s= Queue.take a_Visiter in
    if not (deja_vu.(s)) then
    (
    visite_voisins g.(s);
    deja_vu.(s) <- true;
    )
done;
(* Maintenant, la composante connexe de sd correspond aux sommetsde deja_vu *)
let res = ref [] in
for i=0 to n-1 do
    if deja_vu.(i) then
    res:= i::(!res)
done;
!res
;;
```

3.2.5 Terminaison

Variant de boucle :

- *Pour la version 1 (sans doublons dans G), le nombre de sommets non noirs est un variant de boucle.*
- *Pour la version 2 (avec* $A_visiterquipeutcontenirdesdoublons), onpeutprendreelecouple(nombredesommetsnonnoir, |aVisiter|)po$

3.2.6 Complexité

Méthode de l'exercice 1 :
Prendre chaque ligne, voir ce qu'elle coute et combien de fois max elle est executée.

3.3 Parcours en largeur

Un parcours en largeur traite d'abord les sommets les plus proches du sommet de départ.

Pour réaliser un parcours en largeur, on prend aVisiter de type file d'attente.
Les deux exemples précedents étaient des parcours en largeur.

3.3.1 Invariant de boucle du parcours en largeur

Propriété 2 *A un certain instant d'un parcours en largeur. Soit* n *le nombre de sommets dans la file et* $s_0, ..., s_{n-1}$ *les sommets dans l'ordre. Alors* $\exists d \in N$ *et* $k \in [\![0, n[\![$ *tq* $(s_{n-1}, ..., \overrightarrow{s_k}, ..., s_0)$
- $s_0, ..., s_{k-1}$ *sont à distance inferieur a* d *de* s_d
- $s_k, ..., s_n$ *sont a distance inferieur a* $d+1$ *de* s_d

En outre, les noirs sont les sommets à distance inferieur a $d-1$ *ainsi que les sommets à distance* d *qui ne sont pas dans la file.*

3.4 Parcours en profondeur

Dans un parcours en profondeur, on poursuit un chemin autant que possible avant de partir sur un autre. Plus précisement, on visite en priorité les voisins du dernier sommet visité. Il suffit de remplacer la file par une pile.

Exemple 3 *Calcul d'un composante connexe*

```
let composante_connexe_profondeur g sd=
let n= Array.length g in
let deja_vu= Array.make n false in
let a_Visiter= Stack.create () in
Stack.push sd a_Visiter;
let rec visite_voisins = function
    | [] -> ()
    | t::q -> Stack.push t a_Visiter;
            visite_voisins q
in
while  not (Stack.is_empty a_Visiter) do
    let s= Stack.pop a_Visiter in
    if not (deja_vu.(s)) then
    (
    visite_voisins g.(s);
    deja_vu.(s) <- true;
    )
done;
(* Maintenant, la composante connexe de sd correspond aux sommetsde deja_vu *)
let res = ref [] in
```

```
for i=0 to n-1 do
    if deja_vu.(i) then
    res:= i::(!res)
done;
!res
;;
```