Chapitre 14 : Algorithme de Kosaraju

10 janvier

Les figures utilisées dans ce document proviennent de l'ouvrage suivant :

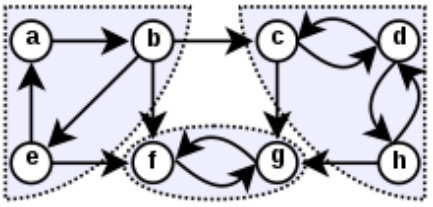
https ://jeffe.cs.illinois.edu/teaching/algorithms/
On rappelle la notion de sous graphe induit :

Définition 1. *Soit $G = (S, A)$ un graphe et $S' \subset S$ un sous ensemble de sommets de G. On appelle $G' = (S', A')$ le sous graphe induit par S' où $A' = \{(x, y) \in A : x \in S' \text{ et } y \in S'\}$.*

On rappelle la définition des composantes fortement connexes dans un graphe orienté :

Définition 2. *On définit la relation R sur S par $\forall x, y \in S, xRy$ ssi il existe un chemin de x à y et il existe un chemin de y à x. Cette relation est une relation d'équivalence et ses classes d'équivalences forment une partition de S. Chaque classe d'équivalence S_i induit un graphe $G_i = (S_i, E_i)$ et l'ensemble des graphes G_i est l'ensemble des composantes fortement connexes de G.*

Voici un exemple illustrant cette notion :



On a vu en première année qu'on peut identifier les composantes connexes d'un graphe non orienté à l'aide d'un parcours (qu'il soit en profondeur ou en largeur).
Calculer les composantes fortement connexes est une tâche a priori plus ardue mais on va voir que cela peut être fait en exploitant le parcours en profondeur et ses propriétés.

1 Rappels sur le parcours en profondeur

Commençons par rappeler le principe du parcours en profondeur. On part d'un sommet, on le traite et on appelle récursivement la procédure sur tous ses voisins s'ils n'ont pas encore été découverts.

```
pp_rec(g,s) :
   if (not(decouvert[s]) :
      decouvert[s]<-true;
      pour tous les x voisins de s :
         pp_rec(g,x);
      traite[s]<-true;
```

Un appel à `pp_rec(g,s)` va parcourir les sommets accessibles depuis le sommet `s`. Afin de s'assurer de visiter tous les sommets du graphe et non pas seulement ceux accessibles depuis `s`, on appellera successivement cette procédure sur tous les sommets qui n'ont pas encore été découverts lors des appels précédents.

```
pp(g) :
   pour tous les sommets x de g :
      traite[x]<-false;
      decouvert[x]<-false;
   pour tous les sommets x de g :
      si non decouvert[x] :
          pp_rec(g,x);
```

La distinction entre les sommets découverts et non découverts est essentielle pour écrire un parcours en profondeur. Elle garantit que chaque sommet n'est traité qu'une seule fois et que l'algorithme s'arrête.

Nous avons ici ajouté la notion de sommet traité qui signifie qu'un sommet ainsi que tous les sommets accessibles depuis ce dernier ont été découverts. Cette notion s'est déjà révélée utile pour les applications comme la détection de cycle dans un graphe orienté ou le tri topologique et nous sera aussi utile ici.

Un parcours en profondeur induit une forêt du parcours en profondeur. Cette dernière est définie par une relation de parenté : on dit que v est le fils de u si l'appel à `pp_rec(g,v)` est directement effectué par la fonction `pp_rec(g,u)`. Ainsi, v est descendant de u si l'appel à `pp_rec(g,v)` est fait au sein de la fonction `pp_rec(g,u)` (par elle même ou par un de ses appels récursifs imbriqués). En d'autre terme :

v sera un descendant de u dans la forêt ssi lorsque v est découvert alors u est découvert mais pas encore traité

(la fonction `pp_rec(g,u)` n'est pas encore terminée).

C'est pour la caractérisation de cette relation de descendance que l'on a introduit la notion de sommet traité.

En plus de la caractérisation de la relation de descendance entre sommets au sein de la forêt de parcours, soulignons une propriété fondamentale :
Si au moment de la découverte du sommet `u` il existe un chemin reliant `u` à `v` composé uniquement de sommets non découverts alors `v` sera un descendant de `u`.

2 Graphe miroir

Définition 3. *Le graphe miroir de G est le graphe où on a interverti l'orientation des arêtes. Soit $G = (S, A)$ un graphe orienté, le graphe miroir de G, noté $\bar{G}$ est définit par (S, A') où $A' = \{(x, y) : (y, x) \in A\}$.*

Propriété 1. *Soit $G = (S, A)$ un graphe orienté. Le miroir du miroir de G est G lui-même.*

Définition 4. *Soit $G = (S, A)$ un graphe orienté. On dit qu'un sommet $s \in S$ est un puit ssi G ne contient aucune arête de la forme $(s, \ldots)$.*

Soit $G = (S, A)$ un graphe orienté. On dit qu'un sommet $s \in S$ est une source ssi ssi G ne contient aucune arête de la forme $(\ldots, s)$.

Théorème 1. *Soit $G = (S, E)$ un graphe orienté. Un sommet $s \in S$ est un puit dans G ssi c'est une source dans $\bar{G}$.*

Définition 5. *Soit $G = (S, E)$ un graphe orienté.*
Pour tout $s \in S$, on note $reach_G(s) = \{x \in S : s \rightsquigarrow_G x\}$.

Définition 6. *Soit $G = (S, E)$ un graphe orienté.*
Pour tout $s \in S$, on note $reach_G^{-1}(s) = \{x \in S : x \rightsquigarrow_G s\} = \{x \in S : s \in reach(x)\}$.

Propriété 2. *Soit $G = (S, E)$ un graphe orienté. Pour tout $s \in S$, $reach_G^{-1}(s) = reach_{\bar{G}}(s)$.*

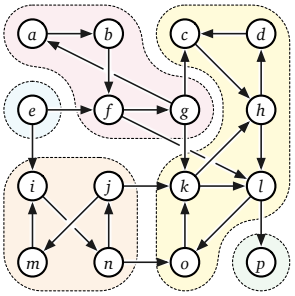
3 Calcul d'une composante fortement connexe

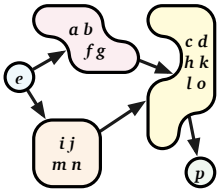
Soit $G = (S, E)$ un graphe orienté.

Pour tout $s \in S$, $cfc(s) = \{x \in s : x \rightsquigarrow_G s \text{ et } s \rightsquigarrow_G x\} = \text{reach}_G(s) \cap \text{reach}_{\bar{G}}(S)$.

Ainsi, à l'aide de n'importe quel type de parcours d'origine `s` on peut déterminer ces ensembles en parcourant une fois G et une fois $\bar{G}$. On peut donc calculer une CFC en $O(|S| + |A|)$ et toutes les CFC en $O(|S|.|A|)$.

3.1 Algorithme de Kosaraju : calcul de l'ensemble des composantes fortement connexes en temps linéaire





Faisons un pp en suivant l'ordre alphabétique puis en suivant l'inverse de l'ordre alphabétique.

Que remarque-t-on ? Nous venons de mettre en évidence l'idée principale de l'algorithme de Kosaraju : on fait un parcours en profondeur du graphe en prenant un ordre bien choisi sur les sommets pour garantir que chaque appel par la fonction `pp` à la fonction `pp_rec` détermine une composante connexe puisque les sommets accessibles depuis le sommet qui fait le premier appel sont exactement toute sa composante fortement connexe (**et rien de plus**).

Il suffit donc de choisir à chaque fois un sommet dont la composante fortement connexe est un puit c'est-à-dire dont aucune arête ne sort.

On obtient alors l'algorithme suivant :

```
CFC(g):
   num=0;
   tant que g n'est pas vide :
        C initialisée à vide
        choisir un sommet s dans une composante puit
        appeler pp_rec(g,s) en l'adaptant pour que les sommets visités
        soient considérés dans la CFC num et insérés dans C
        num++
        supprimer dans G tous les sommets de C et leurs arêtes
```

Une chose reste très floue ici : comment déterminer un sommet dans une composante fortement connexe puit ? Ce n'est pas si simple.

Lemme 1. *Soit $G = (S, A)$ un graphe orienté. Le sommet qui est traité en dernier par un parcours en profondeur est dans une composante fortement connexe source (cad dans laquelle aucune arête ne rentre).*

Lemme 2. *Soit $G = (S, A)$ un graphe orienté. Un sommet est dans une CFC source de $\bar{G}$ ssi il est dans une CFC puit de G.*

On en déduit que **trouver un sommet dans une CFC puit est équivalent à considérer le dernier sommet traité dans un parcours en profondeur de $\bar{G}$.**

Ainsi, l'ordre sur les sommets que l'on va utiliser pour parcourir G à la recherche des CFC est l'ordre inverse de la fin de traitement des sommets associée à un parcours en profondeur de $\bar{G}$.

L'algorithme final a donc la forme suivante :

1. on calcule $\bar{G}$ et on fait un parcours en profondeur afin de calculer les dates de fin de traitement.
2. on fait un parcours en profondeur de G en suivant l'ordre inverse des dates de fin de traitement obtenues à l'étape précédente pour sélectionner un nouveau sommet pour lancer `pp_rec`.

On remarque que plutôt que de calculer explicitement les dates de fin de traitement, il suffit de les empiler dans une pile quand on a fini de les traiter afin d'obtenir une pile qui contient les sommets dans l'ordre souhaité.

```
let mirror g =
   let n = Array.length g in
   let res = Array.make n [] in
   let rec parcours_voisins i l = match l with
        |[]->()
        |a::suite-> res.(a)<-i::res.(a);
   in for i = 0 to n-1 do
           parcours_voisins i g.(i);
      done;
   res;;

let rec pp_rec_rev g dec liste s =
   if (not dec.(s)) then
      begin
      dec.(s)<-true;
      List.iter (pp_rec_rev g dec liste) (g.(s));
      liste:=s::(!liste);
      end;;

let pp_rev g =
    let n = Array.length g in
    let dec = Array.make n false in
    let listepuit = ref [] in
    for i = 0 to n-1 do
       if (not dec.(i)) then
         pp_rec_rev g dec listepuit i ;
    done;
    !listepuit;;

let rec pp_rec g dec num cc s =
   if (not dec.(s)) then
      begin
      dec.(s)<-true;
      cc.(s)<-num;
      List.iter (pp_rec g dec num cc) (g.(s));
      end;;

let kosaraju g =
    let n = Array.length g in
    let dec = Array.make n false in
    let cc = Array.make n (-1) in
    let listepuit = pp_rev (mirror g) in
    let rec parcourir = function
        |[]->()
        |a::suite when not dec.(a)->pp_rec g dec a cc a; parcourir suite;
        |a::suite-> parcourir suite;
    in parcourir listepuit;
    cc;;
```

3.2 Graphe des CFC

Soit $G = (S, A)$ un graphe orienté. On peut définir son graphe des composantes connexes comme le graphe dont les sommets correspondent à chaque composante connexe de G et on met une arête de C vers C' ssi il existe une arête $x \to y$ avec $x \in C$ et $y \in C'$. On remarque que ce graphe est acyclique (DAG) et que trouver l'ordre dans lequel déterminer les composantes est équivalent à faire un tri topologique sur ce DAG.

On peut d'ailleurs tester qu'un graphe orienté est acyclique ssi on peut toujours trouver des sommets puits (de degré entrant nul), les éliminer avec leurs arêtes jusqu'à obtenir le graphe vide.

Cette dernière remarque permet de garantir que l'algorithme de Kosaraju est bien correct car il existe à chaque étape une composante puit dans le graphe (un sommet puit dans le graphe des composantes fortement connexes) et celle-ci peut alors être reconstruite à partir de n'importe lequel de ses sommets, par exemple celui dont la date de fin de traitement est la plus grande.

4 Application au problème 2SAT

Considérons le problème 2SAT défini de la manière suivante : on prend en entrée φ une formule sous forme normale conjonctive avec toutes ses clauses de taille 2 et on veut décider si elle est satisfiable ou non.

On va montrer qu'il existe un algorithme linéaire qui permet de répondre à cette question.

Définition 7. *Soit φ une formule en 2-FNC sur l'ensemble de variables $\mathcal{V} = \{v_1, \dots, v_n\}$. On définit le graphe d'implication de φ, noté G_φ, par $G_\varphi = (S, A)$ où : $S = \mathcal{V} \cup \{\neg v : v \in \mathcal{V}\}$: les sommets sont les littéraux.*

Si $\ell_1 \vee \ell_2$ est une clause de la formule alors on met une arête de $\neg\ell_1$ vers ℓ_2 et une autre arête de $\neg\ell_2$ vers ℓ_1.

Construire les graphes d'implication des formules :

— $\Phi = (\neg y \vee \neg z) \wedge (\neg x \vee z) \wedge z \wedge (y \vee x) \wedge (\neg y \vee z)$

— $\Psi = (\neg x \vee t) \wedge (x \vee \neg z) \wedge (y \vee \neg z) \wedge (x \vee z) \wedge (\neg y \vee \neg x) \wedge (\neg t \vee z)$.

Théorème 2. *Si G_φ admet un chemin de ℓ_1 vers ℓ_2 alors $\ell_1 \Rightarrow \ell 2$ est conséquence syntaxique de φ.*

Théorème 3. *φ est satisfiable si et seulement si aucune composante fortement connexe de G_φ ne contient à la fois une variable et sa négation.*