Chapitre 5

Trois exemples d'algorithmes de graphes

Hugo Salou MPI*

Dernière mise à jour le 31 mars 2023

Table des matières

Dans ce chapitre, on s'intéresse aux graphes. Nous rappellerons les notions et algorithmes de graphes vus l'année dernière. On considère 3 exemples d'algorithmes de graphes.

Par exemple, l'année dernière, nous avons vu comment décomposer un graphe non-orienté en composantes connexes : on choisit un sommet au hasard, puis on parcours les voisins de ce sommets, et on répète. Mais, dans un graphe orienté, la notion de « composante connexe » n'est plus la même dans un graphe orienté. C'est l'algorithme décrit dans la section 1.

1 Composantes fortement connexes (CFC)

Dans la suite de cette section, $G = (S, A)$ est un graphe orienté.

Définition : On dit que $v \in S$ est *accessible* depuis $u \in S$ s'il existe un chemin de u à v, que l'on note $u \xrightarrow{\star} v$. De même, on dit que $v \in S$ est *co-accessible* depuis $u \in S$ s'il existe un chemin de v à u, que l'on note $v \xrightarrow{\star} u$.

Définition ($u \sim_G v$) : On note $u \sim_G v$ si $u \xrightarrow{\star} v$ et $v \xrightarrow{\star} u$.

Remarque :
La relation $\sim_G$ est une relation d'équivalence.

Digression L'année dernière, dans un graphe non orienté, on peut noter $\sim$ la relation d'équivalence induite par, si $\{u, v\} \in A$, alors $u \sim v$. Dans ce cas, le même chemin permet d'aller de u à v, puis de v à u, et ce chemin est le même. Mais, dans un graphe orienté, si $u \sim_G v$, les chemins de u à v puis de v à u ne sont pas forcément les mêmes.

Définition (CFC) : On appelle *composantes fortement connexes* (CFC) d'un graphe G, les classes d'équivalences de la relation $\sim_G$.

Définition : On dit d'un graphe ayant une unique composante fortement connexe qu'il est *fortement connexe*.

Définition : On appelle *ensemble fortement connexe* un ensemble $V \subseteq S$ tel que G_V est fortement connexe où G_V est le *graphe induit* par V : $G_V = (V, A \cap V^2)$.

Lemme : Si W est une composante fortement connexe de G, alors W est un ensemble fortement connexe.

■

Propriété : Les composantes fortement connexes sont les ensembles fortement connexes qui sont maximaux pour l'inclusion.

■

Définition : On appelle *graphe réduit* de G le graphe orienté $\hat{G} = (\hat{S}, \hat{A})$ où

$$\hat{S} = \{\bar{x} \mid x \in S\} \quad \text{et} \quad \hat{A} = \big\{(\bar{x}, \bar{y}) \mid (x, y) \in A \text{ et } \bar{x} \neq \bar{y}\big\}.$$

Remarque : — $\hat{G}$ est acyclique.

— Pour tout couple $(\bar{x}, \bar{y}) \in \hat{S}^2$, si $\bar{x} \xrightarrow[\hat{G}]{\star} \bar{y}$, alors $\forall u \in \bar{x}, \forall v \in \bar{y}, x \xrightarrow[G]{\star} v$.

1.1 Rappels

Définition : La *bordure* d'un ensemble de sommets $V \subseteq S$, noté $\mathscr{B}(V)$, est l'ensemble des successeurs de V non dans V :

$$\mathscr{B}(V) = \{s \in S \setminus V \mid \exists u \in V,\ (u, s) \in A\}.$$

Définition (parcours) **:** Un *parcours* est une permutation des sommets $(L_1, L_2, \dots, L_n)$ telle que, pour $i \in [\![1, n-1]\!]$,

$$L_i \in \mathscr{B}(\{L_1, \dots, L_{i-1}\}) \quad \text{ou} \quad \mathscr{B}(\{L_1, \dots, L_{i-1}\}) = \varnothing.$$

On dit d'un L_i avec $i \in [\![1, n]\!]$ tel que $\mathscr{B}(\{L_1, \dots, L_{i-1}\}) = \varnothing$, que c'est un *point de régénération* du parcours.

Lemme : Si $V \subseteq S$ est tel que $\mathscr{B}(V) = \varnothing$, il n'existe aucun chemin d'un sommet de V à un chemin de $S \setminus V$.

■

Définition : Soit $(L_1, L_2, \dots, L_n)$ un parcours de G. On note K le nombre de ses points de régénération. On note $(r_k)_{k \in [\![1,k]\!]}$ l'extractrice des points de régénération. En notant de plus $r_{K+1} = n + 1$. Le *partitionnement associé au parcours* est alors

$$\Big\{\{L_{r_i}, L_{r_i+1}, \dots, L_{r_{i+1}-1}\} \,\Big|\, i \in [\![1, K]\!]\Big\}..$$

Définition : Un partitionnement P_1 est un *raffinement* d'un partitionnement P_2 dès lors que

$$\forall C_1 \in P_1,\ \exists C_2 \in P_2,\ C_1 \subseteq C_2.$$

Propriété : Les composantes fortement connexes sont un raffinement des partitionnement des parcours.

■

1.2 Rangement particulier de graphe

Définition (Sommet ouvert) **:** Soit $(L_1, \dots, L_n)$ un parcours de G. Pour $k \in [\![1, n]\!]$ et $i \in [\![1, k[\![$, on dit que L_i est *ouvert* à l'étape k si

$$\mathrm{Succ}(L_i) \not\subseteq \{L_j \mid j \in [\![1, k[\![\}$$

où $\mathrm{Succ}(L_i)$ est l'ensemble des successeurs de L_i.

Cette définition nous permet de définir les parcours en largeur et en profondeur.

Définition (parcours en largeur) **:** Soit $(L_1, \dots, L_n)$ un parcours. Il est dit *en largeur* si chaque sommet du parcours qui n'est pas un point de régénération est un successeur du premier sommet ouvert à cette étape :

$$\forall k \in [\![2, n]\!], \qquad \mathscr{B}(\{L_j \mid j \in [\![1, k[\![\}) = \varnothing \quad \text{ou} \quad L_k \in \mathrm{Succ}(L_{i_0})$$

avec $i_0 = \min\{i \in [\![1, k[\![\mid L_i \text{ ouvert à l'étape } k\}$.

Définition (parcours en profondeur) **:** Soit $(L_1, \dots, L_n)$ un parcours. Il est dit *en largeur* si chaque sommet du parcours qui n'est pas un point de régénération est un successeur du dernier sommet ouvert à cette étape :

$$\forall k \in [\![2, n]\!], \qquad \mathscr{B}(\{L_j \mid j \in [\![1, k[\![\}) = \varnothing \quad \text{ou} \quad L_k \in \mathrm{Succ}(L_{i_0})$$

avec $i_0 = \max\{i \in [\![1, k[\![\mid L_i \text{ ouvert à l'étape } k\}$.

Définition (tri topologique) **:** Soit $(T_i)_{i \in [\![1,n]\!]}$ une permutation des sommets. On dit que T est un *tri topologique* si

$$\forall (i, j) \in [\![1, n]\!]^2, \qquad \text{si } (T_i, T_j) \in A \text{ alors } i \leqslant j.$$

Définition : Soit une permutation de sommets $(T_i)_{i \in [\![1,n]\!]}$. On appelle *rang* de $u \in S$ dans T le plus petit indice dans T d'un élément accessible ($u \xrightarrow{\star} v$) et co-accessible ($v \xrightarrow{\star} u$) depuis u. On définit

$$\mathrm{rang}_T(u) = \min\{i \in [\![1, n]\!] \mid T_i \sim_G u\}.$$

Définition : Étant donné une permutation $(T_i)_{i \in [\![1,n]\!]}$ des sommets de G, on définit

la relation

$$\preccurlyeq_T = \{(u, v) \in S^2 \mid \text{rang}(u) \leqslant \text{rang}(v)\}.$$

On dit alors que T est un *tri préfixe* dès lors que, pour tout $(u, v) \in S^2$, si $u \xrightarrow{\star} v$ alors $u \preccurlyeq_T v$.

Remarque :
Étant donné une permutation T, pour tout couple de sommets (u, v),

$$u \sim_G v \iff (u \preccurlyeq_T v \text{ et } v \preccurlyeq_T u).$$

1.3 Graphe transposé et CFC

Définition : Étant donné un graphe $G = (S, A)$, on appelle *graphe transposé* de G, que l'on note $G^\top$, le graphe

$$G^\top = \Big(S, \{(y, x) \in S^2 \mid (x, y) \in A\}\Big).$$

Propriété : Soit T un tri préfixe de G. Soit L un parcours de $G^\top$ utilisant l'ordre des points de régénération induit par T. Alors, la partition associée à L est la décomposition en composantes forment connexes.

■

1.4 Calcul de tri préfixe

On peut donc donner un algorithme calculant un tri préfixe. On donne cet algorithme en impératif, même si la version récursive est plus simple.

Algorithme 1 Calcul d'un tri préfixe

Entrée Un graphe $G = (S, A)$.
Sortie Un tri préfixe des sommets de G.

```
1 : Procédure ExploreDescendants(s, Visités, Res)
2 :     Entrée Un graphe G = (S, A), Res, Visités, s ∈ S.
3 :     Sortie Modifie Res et Visités de sorte que Res soit un parcours préfixe de Visités et des
        sommets accessibles depuis s.
4 :     todo ← pileVide
5 :     empiler((s, Succ(s)), todo)
6 :     Visités ← {s} ∪ Visités
7 :     tant que todo ≠ pileVide faire
8 :         (x, ℓ) ← depiler(todo)
9 :         si ℓ = ( ) alors
10 :            Res ← x · Res
11 :        sinon
12 :            t · ℓ' ← ℓ        ▷ on sépare la tête t du reste ℓ' de la pile ℓ.
13 :            empiler((x, ℓ'), todo)
14 :            si t ∉ Visités alors
15 :                Visités ← {t} ∪ Visités
16 :                empiler((t, Succ(t)), todo)

17 : Visités ← ∅
18 : Res ← ( )
19 : tant que S \ Visités ≠ ∅ faire
20 :     s ← un sommet de S \ Visités
21 :     ExploreDescendants(s, Visités, Res)
22 : retourner Res
```

On montre la correction de cet algorithme. On cherche des invariants *intéressants*, que l'on ne prouvera pas. Pour la boucle "tant que," dans la procédure ExploreDescendants, on choisit les invariants

1. pour tout couple de sommets $(u, v) \in S^2$, si $\mathscr{C}\ell(u) \subseteq \text{Res}$ et que $u \xrightarrow{\star} v$, alors $\mathscr{C}\ell(v) \subseteq \text{Res}$ et $\text{rand}_{\text{Res}}(u) \leqslant \text{rang}_{\text{Res}}(v)$,
2. $K(\text{todo}) \cup \text{Res} = \text{Visités}$, où où $K(\text{todo})$ est l'ensemble des premières composantes des couples de todo,
3. les clés de todo, du fond de la pile au sommet forment un chemin,
4. si u est un élément de Visités, et v est un descendant de u,
   - ou bien $v \in \text{Res}$,
   - ou bien $v \in K(\text{todo})$
   - ou bien v est un descendant d'un élément d'une liste adjointe à un élément $w \in K(\text{todo})$ tel que $u \xrightarrow{\star} w$.

On admet que ces 4 propriétés sont invariantes. À la fin, $\forall x \in \text{Res}$, $\mathscr{C}\ell(x) \subseteq \text{Res}$, et dnc l'invariant 1 assure alors que nous avons un tri préfixe.

1.5 Algorithme de Kosaraju

Algorithme 2 Algorithme de Kosaraju

Entrée Un graphe $G = (S, A)$
Sortie Les composantes fortement connexes de G

```
1 : On calcule un tri préfixe de G.
2 : On parcours G^⊤ en utilisant l'ordre T comme points de régénération.
3 : On retourne le plus petit partitionnement associé au parcours.
```

1.6 Applications

Théorème : 2-CNF-SAT $\in \mathbf{P}$.

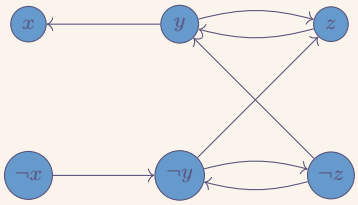

FIGURE 1 – Représentation d'une formule 2-CNF-SAT par un graphe

Preuve :
Soit $H \in$ 2CNF. On pose

$$H = (\ell_{1,1} \vee \ell_{1,2}) \wedge \ldots \wedge (\ell_{n,1} \vee \ell_{n,2}).$$

Dans la suite, on note $\ell^c_{i,j}$ le littéral opposé à $\ell_{i,j}$. À la formule H, nous associons le graphe G_H défini comme suit :

$$S_H = \{(\ell_{i,j}) \mid i \in [\![1,n]\!], j \in \{1,2\}\} \cup \{(\ell^c_{i,j}) \mid i \in [\![1,n]\!], j \in \{1,2\}\},$$
$$A_H = \{(\ell^c_{i,1}, \ell_{i,2} \mid i \in [\![1,n]\!]\} \cup \{(\ell^c_{i,2}, \ell_{i,1}) \mid i \in [\![1,n]\!]\}.$$

Lemme : Si ρ est un modèle de H et $u \xrightarrow{\star} v$ tel que $[\![u]\!]^\rho = \boldsymbol{V}$, alors $[\![v]\!]^\rho = \boldsymbol{V}$. ■

Propriété : H est satisfiable si, et seulement si aucune variable et sa négation ne se trouvent dans la même CFC de G_H. ■

Algorithme 3 Solution au problème 2CNFSAT

Entrée H une 2-CNF
Sortie ρ un modèle de H ou None si H n'est pas satisfiable

On construit G_H
On construit les CFC $C_1, \ldots, C_p$ de G_H (dans un ordre topologique)
si il existe x et $i \in [\![1,p]\!]$ tel que $x \in C_i$ et $\neg x \in C_i$ **alors**
    retourner None
sinon
    retourner ρ défini comme

$$\rho : \mathbb{Q} \longrightarrow \mathbb{B}$$
$$x \longmapsto \begin{cases} \boldsymbol{F} & \text{si } i < j \\ \boldsymbol{V} & \text{sinon} \end{cases}$$

    où $x \in C_i$ et $\neg x \in C_j$.

□

2 Arbres couvrants de poids minimum

Définition (Arbre) : Soit $G = (S, A)$ un graphe non-orienté. On dit que G est un *arbre* si G est connexe et acyclique.

Définition (Arbre couvrant) : Étant donné un graphe non orienté pondéré par poids positifs $G = (S, A, c)$,[1] on dit de $G' = (S', A')$ que c'est un *arbre couvrant* de G si $S' = S$ et $A' \subseteq A$, et G' est un arbre.

Définition (Arbre couvrant de poids minimum) : Étant donné un graphe non orienté pondéré $G = (S, A, c)$ et un arbre couvrant $T = (S', A')$, on appelle *poids* de l'arbre T la valeur $\sum_{a \in A'} c(a)$.

Si G est connexe, il admet au moins un arbre couvrant, on peut définir l'*arbre couvrant de poids minimum* (ACPM).

On définir alors le problème

$$\text{ACPM}^{2} \begin{cases} \textbf{Entrée} & : G = (S, A, c) \text{ connexe} \\ \textbf{Sortie} & : \text{le poids de l'arbre couvrant de poids minimum.} \end{cases}$$

Algorithme 4 Algorithme de KRUSKAL

Entrée $G = (S, A, c)$ un graphe connexe
Sortie Un arbre couvrant de poids minimum

```
1 : B ← ∅
2 : U ← ∅
3 : tant que il existe u et v tels que u ≁_B v faire
4 :     Soit {x, y} ∈ A \ U de poids minimal
5 :     si x ∼_B y alors
6 :         U ← {{x, y}} ∪ U
7 :     sinon
8 :         U ← {{x, y}} ∪ U
9 :         B ← {{x, y}} ∪ B
10 : retourner T = (S, B)
```

Propriété : L'algorithme de KRUSKAL est correct. ∎

À la fin, B induit un graphe connexe et B est contenu dans un ACPM, c'en est donc un.

Une structure pour la gestion des partitions : UnionFind.

Définition (Type de données abstrait UnionFind) : On définit le type de données abstrait UnionFind comme contenant

- un type `t` de partitions ;
- un type `elem` des éléments manipulés par les partitions ;
- `initialise_partition` : `elem list` $\to$ `t` retournant le partitionnement dans lequel chaque élément est seul dans sa classe ;

1. on dit que c est la fonction de pondération de ce graphe
2. Arbre Couvrant de Poids Minimum

— `find` : `(t * elem)` $\to$ `elem` retournant un représentant de la classe de l'élément. Si deux éléments x et y sont dans la même classe, dans le partitionnement p, alors `find`$(p, x) =$ `find`(p, y) ;

— `union` : `(t*elem*elem)` $\to$ `t` retourne le partitionnement dans lequel on a fusionné les classes des arguments.

On implémente ce type abstrait en OCaml.

Remarque (Niveau zéro – listes de liste) :

```
type 'a t = 'a list list

let initialise_partition (l: 'a list): 'a t =
  List.map (fun x -> [ x ] ) l

let rec find (p: 'a t) (x: 'a): 'a =
  match p with
  | classe :: classes ->
      if List.mem x classe then List.hd classe
      else find classes x
  | [] -> raise Not_Found

let est_equiv (p: 'a t) (x: 'a) (y: 'a): bool =
  (find p x) = (find p y)

let rec extrait_liste (x: 'a) (p: 'a t): 'a list * 'a p =
  match p with
  | classe :: classes ->
      if List.mem x classe then (classe, classes)
      else
        let cl, cls' = extrait_liste x classes in
        (cl, classe :: cls')
  | [] -> raise Not_Found

let union (p: 'a t) (x: 'a) (y: 'a): 'a t =
  if est_equiv p x y then p
  else
    let cx, p' = extrait_liste x p in
    let cy, p'' = extrait_liste y p' in
    (cx @ cy) :: p''
```

Code 1 – Implémentation du type UnionFind en OCaml

Remarque (Niveau un – tableau de classes) :
Dans la case du tableau, on inscrit le numéro de sa classe. Pour `find`, on prend le premier ayant la même classe. Pour `union`, on re-numérote vers un numéro commun. Par exemple,

0	1	0	0	1	2
0	1	2	3	4	5

$\longleftrightarrow \quad \{\{0, 2, 3\}, \{1, 4\}, \{5\}\}.$

Remarque (Niveau deux – tableau de représentants) :
Dans les cases du tableau, on écrit le représentant de la classe de i. Pour `find`, on lit la case. Pour `union`, on re-numérote vers un numéro commun. Par exemple,

2	4	2	2	4	5
0	1	2	3	4	5

$\longleftrightarrow \quad \{\{0, 2, 3\}, \{1, 4\}, \{5\}\}.$

Remarque (Niveau trois – arbres) :
Pour `union`$(0,1)$, on cherche le représentant de 0 (2) puis celui de 1 (4). On fait pointer 4 vers 2. Pour la suite de l'implémentation, *c.f.* DM$_3$.

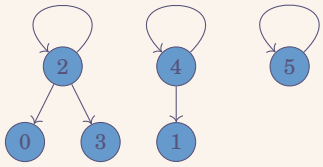

Figure 2 – Représentation par des arbres

Avec cette nouvelle structure, on peut maintenant revenir sur l'algorithme de Kruskal.

Algorithme 5 Algorithme de Kruskal – version 2

Entrée Un graphe $G = (S, A, c)$ un graphe non orienté, pondéré
Sortie Un acpm

```
1 : Soit (e_i)_{i∈⟦1,m⟧} un tri des arrêtes par coût croissant
2 : f ← 0        ▷ Nombre d'union effectuées
3 : p ← initialise_partition(S)
4 : I ← 0
5 : B ← ∅
6 : tant que f < n − 1 faire
7 :     {x, y} ← e_I
8 :     si find(p, x) ≠ find(p, y) alors
9 :         p ← union(p, x, y)
10 :        B ← B ∪ {{x, y}}
11 :        f ← f + 1
12 :    I ← I + 1
13 : retourner (S, B)
```

Étude de complexité. Notons $C^n_{\texttt{find}}$ un majorant du coût de `find` sur une structure contenant n éléments, notons $C^n_{\texttt{union}}$ un majorant du coût de `union` sur une structure contenant n éléments, et notons $C^n_{\texttt{init}}$ un majorant du coût de `init` sur une structure contenant n éléments. La complexité de cet algorithme est de

$$\mathcal{O}\big(C^n_{\texttt{init}} + 2m\, C^n_{\texttt{find}} + n\, C^n_{\texttt{union}} + m \log_2 m\big).$$

3 Couplage dans un graphe biparti

Définition (Couplage) : On appelle *couplage* d'un graphe non orienté $G = (S, A)$, la donnée d'un sous-ensemble $C \subseteq A$ tel que

$$\forall \{x, y\}, \{x', y'\} \in C, \qquad \{x, y\} \cap \{x', y'\} \neq \varnothing \implies \{x, y\} = \{x', y'\}.$$

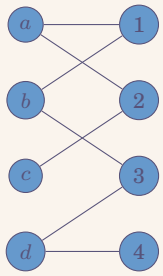

FIGURE 3 – Exemple de couplage

Définition : Un couplage est dit *maximal* s'il est maximal pour l'inclusion ($\subseteq$). Un couplage est dit *maximum* si son cardinal est maximal.

REMARQUE :
Dans toute la suite, on ne considère que des graphes bipartis.

Définition : Étant donné un graphe biparti $G = (S, A)$ et un couplage C, un sommet x est dit *libre* dès lors que

$$\forall \{y, z\} \in C,\ x \notin \{y, z\}.$$

Une chaîne élémentaire[3] $(c_0, c_1, \ldots, c_{2p+1})$ est dit *augmentante* si

- c_0 et c_{2n+1} sont libres ;
- $\forall i \in [\![0, p]\!]$, $\{c_{2i}, c_{2i+1}\} \in A \setminus C$;
- $\forall i \in [\![0, p-1]\!]$, $\{c_{2i+1}, c_{2i+2}\} \in C$.

Propriété : Étant donné un graphe biparti $G = (S, A)$ avec $S = S_1 \uplus S_2$ (partitionnement du graphe biparti), un couplage C est maximum si, et seulement s'il n'admet pas de chaînes augmentantes.

■

3. *i.e.* une chaîne sans boucles.
3. On représente $\Rightarrow$ pour les arrêtes dans le couplage C.

Algorithme 6 ChaîneAugmentante : Trouver une chaîne augmentante dans un graphe biparti $G = (S, A)$ muni d'un couplage C partant d'un sommet $s \in S$

```
1 : Procédure Augmente(x, chaîne)
2 :     pour y ∈ Succ(x) \ chaîne faire
3 :         si y est libre dans C alors
4 :             retourner Some(chaîne ⊎ (y))
5 :         sinon
6 :             Soit z tel que {y, z} ∈ C.
7 :             r ← Augmente(z, chaîne ⊎ (y, z))
8 :             si r ≠ None alors
9 :                 retourner r
10 :            retourner None
11 : si s est libre dans C alors
12 :     retourner Augmente(s, (s))
13 : sinon
14 :     retourner None
```

> Remarque :
> Si un sommet n'est pas libre dans le couplage C, il n'est pas libre dans les couplage obtenus par inversion de chaîne depuis C.

Algorithme 7 Calcul d'un couplage maximum

```
Entrée G = (S, A) un graphe biparti, avec S = S1 ⊎ S2
1 : C ← ∅
2 : Done ← ∅
3 : tant que ∃x ∈ S1 \ Done faire
4 :     Soit un tel x.
5 :     r ← ChaîneAugmentante(G, C, x)
6 :     si r ≠ None alors
7 :         Some(a) ← r
8 :         On inverse la chaîne a dans C.
9 :     Done ← {x} ∪ Done
10 : retourner C
```

Annexe A. Remarques supplémentaires

Le parcours d'un graphe $G = (S, A)$ a une complexité en $\mathcal{O}(|S| + |A|)$.