Algorithmique avancée pour les graphes

1 Composantes fortement connexes

1.1 Rappels sur le parcours en profondeur

Exercice 1.1. Sur la feuille de code :

1. Dans la fonction `explore`, rajouter une instruction pour éviter que les sommets soient parcourus plusieurs fois.
2. Quelle autre instruction est manquante ?
3. Dans la fonction parcours profondeur, pourquoi fait-on une boucle au lieu de juste lancer l'exploration depuis le sommet 0 ?

Proposition 1.1. *À la fin du parcours en profondeur, tous les sommets sont visités et ont des temps de fin de traitement différents.* `parent` *contient une forêt d'arborescences. On notera* $fin(x)$ *le temps de fin de traitement d'un sommet* x.

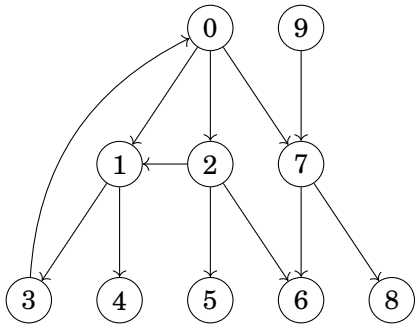
Proposition 1.2. *L'algorithme de parcours en profondeur s'exécute en* $\Theta(|S| + |A|)$.

Définition 1.1. À tout instant de l'algorithme, un sommet v est dans l'un des 3 états suivants :

- non visité s'il n'a pas été visité par l'algorithme,
- actif si `explore` a été appelé sur `v` mais n'a pas encore terminé,
- traité si `explore` a terminé son exécution sur `v`.

Le sommet en cours est le sommet `v` tel que `explore(v)` est la fonction dont le code est en train d'être exécuté. Il est actif.

Exercice 1.2.



Calculer les temps de fin de traitement pour le graphe ci-dessus. On supposera que les listes d'adjacences sont triées par ordre croissant.

Que dire des fin de traitements pour un graphe acyclique ?

Proposition 1.3. (*i*) *Il n'y a pas d'arc d'un sommet traité vers un sommet non visité.*

(*ii*) *Tout chemin d'un sommet traité vers un sommet non visité passe par un sommet actif.*

Proposition 1.4. *À tout instant, l'ensemble des sommets actifs est exactement l'ensemble des ancêtres du sommet en cours* (*sommet en cours inclus*).

1.2 Algorithme de Kosaraju

Définition 1.2. Deux sommets u et v sont fortement connectés si il y a un chemin de u à v et de v à u.

Proposition 1.5. *La forte connexité est une relation d'équivalence et ses classes d'équivalence sont appelées composantes fortement connexes.*

Proposition 1.6. *Soit* C_1 *et* C_2 *des composantes fortement connexes, on a* :

- *soit il n'y a aucun chemin d'un sommet de* C_1 *à un sommet de* C_2,
- *soit il y a un chemin de chaque sommet de* C_1 *à chaque sommet de* C_2.
  On dit alors que C_2 *est accessible depuis* C_1.

Proposition 1.7. *La relation « est accessible » est antisymétrique et transitive. C'est une relation d'ordre* (*partiel*).

Exercice 1.3. Prouver la proposition 1.7

Définition 1.3. Une composante source est une composante qui n'est accessible depuis aucune autre composante.

Une composante puits est une composante qui n'a accès à aucune autre composante.

Proposition 1.8. *Il existe toujours une composante source et une composante puits.*

Définition 1.4. Soit G un graphe. $G^\top$ est le graphe dont on a retourné toutes les arêtes.

Proposition 1.9. *Soit $G = (S, A)$ un graphe.*

Il y a un chemin de v à u dans $G^\top$ si et seulement si il y a un chemin de u à v dans G .

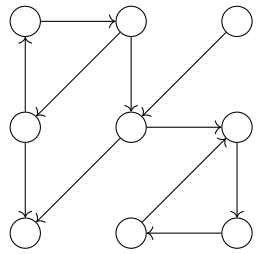
Lemme 1.10. *On suppose qu'on a appliqué un parcours en profondeur à un graphe G . Soient C_1 et C_2 des composantes connexes, et soient u_1 et u_2 leurs sommets ayant la plus grande fin de traitement. Si C_2 est accessible depuis C_1, alors $fin(u_2) < fin(u_1)$.*

Algorithme 1.1 (Algorithme de Kosaraju)**.**

> Données : G
>
> Faire un parcours en profondeur de G
>
> Pour chaque sommet de G pris dans l'ordre décroissant de fin de traitement :
>
> Si v n'est pas marqué :
>
> Marquer tous les sommets non marqués et accessibles depuis v dans $G^\top$ par une nouvelle couleur.

Théorème 1.11. *À la fin de l'algorithme de Kosaraju, deux sommets ont la même couleur si et seulement si ils sont dans la même composante fortement connexe.*

Proposition 1.12. *L'algorithme de Kosaraju s'exécute en $\Theta(|S| + |A|)$.*



Exercice 1.4.

Éxécuter l'algorithme de Kosaraju sur le graphe ci-dessus.

Exercice 1.5. On considère le graphe dont les sommets sont les composantes fortement connexes et tel qu'il y a une arrête de C_1 à C_2 si et seulement si C_2 est accessible depuis C_1. Montrer que ce graphe est acyclique. Que peut-on dire de l'ordre dans lequel l'algorithme de Kosaraju découvre les composantes fortement connexes.

Exercice 1.6. Sur la feuille de code :

1. Compléter la fonction `explore2` pour qu'elle remplisse la pile des fins de traitement.
2. Compléter la fonction `sommets_accessibles` pour qu'elle marque les sommets accessibles.
3. Compléter la ligne 76.

1.3 Application au problème 2-SAT

Définition 1.5. Dans le problème 2-SAT, on donne un ensemble de n variables et un ensemble de 2-clauses sur ces variables (disjonction de deux littéraux). On doit dire s'il existe une valuation satisfaisant toutes les clauses.

Proposition 1.13. *On a pour deux littéraux a et b :*

$$a \wedge b \quad \Leftrightarrow \quad \neg a \to b \quad \Leftrightarrow \quad \neg b \to a \qquad \text{et} \qquad \neg\neg a \quad \Leftrightarrow \quad a.$$

Définition 1.6. On définit le graphe G dont les sommets sont les $2n$ littéraux. Pour chaque clause $a \vee b$, on crée les arcs $\neg a \to b$ et $\neg b \to a$, en utilisant l'équivalence $\neg\neg a \Leftrightarrow a$ si nécessaire.

Proposition 1.14. *Si x et $\neg x$ sont dans la même composante fortement connexe de G alors $x \Leftrightarrow \neg x$ et les clauses ne sont pas satisfiables.*

Proposition 1.15. *Soit a et b deux littéraux. Si a et b sont dans la même composante fortement connexe C, alors $\neg a$ et $\neg b$ sont aussi dans une même composante fortement connexe notée $\neg C$.*

Proposition 1.16. *S'il n'y a pas de composante fortement connexe contenant un littéral et sa négation, alors les clauses sont satisfiables.*

Proposition 1.17. *Il existe un algorithme linéaire (en $O(|S| + |A|)$) pour résoudre 2-SAT.*

Exercice 1.7. Prouver la proposition 1.17.

2 Arbre couvrant de poids minimum

2.1 Énoncé du problème

Dans cette partie, $G = (S, A)$ est un graphe connexe non orienté qu'on muni d'une fonction de poids $w : A \longrightarrow \mathbb{Q}$.

Définition 2.1. Un ensemble d'arêtes couvre un sommet s'il contient une arête adjacente à ce sommet.

Définition 2.2. Un arbre couvrant d'un graphe connexe $G = (S, A)$ est un ensemble de $|S| - 1$ arêtes sans cycle. Il couvre tous les sommets.

Proposition 2.1. *Tout graphe connexe admet au moins un arbre couvrant.*

Exercice 2.1. Prouver la proposition 2.1.

Définition 2.3. Le problème de l'arbre couvrant de poids minimum consiste, étant donné un graphe connexe $G = (S, A)$, à trouver un arbre couvrant dont la somme des poids des arêtes est minimale.

2.2 Unir et trouver

On définit une structure pour manipuler efficacement les classes d'équivalences d'une relation d'équivalence.

Définition 2.4. Une structure unir & trouver (Union-Find) est une structure de données qui dispose de deux opérations :

- `trouver(x)` : renvoie un représentant de la classe d'équivalence de `x`. Il est unique pour chaque classe.
- `unir(x,y)` : unit dans la structure les classes de `x` et `y`.

Implémentation naïve par un dictionnaire

Le représentant de chaque élément est donné par un dictionnaire.

Exercice 2.2. Sur la feuille de code, compléter les fonctions `trouver` et `unir`.

Proposition 2.2. *L'opération Trouver a une complexité $O(1)$ et Unir une complexité $O(n)$ avec n le nombre d'éléments.*

Implémentation par des arbres

Le représentant de la classe d'un d'élément est la racine de l'arbre dans lequel se situe l'élément.

Exercice 2.3. Sur la feuille de code, compléter les fonctions `trouver_naif` et `unir_naif`.

1ère heuristique : l'union par rang

On ajoute à chaque élément la profondeur de son sous-arbre (ou un majorant). On l'initialise à 0. Lors de la fusion, on fait pointer l'arbre de plus petit rang sur celui de plus grand rang.

Proposition 2.3. *Si x a rang k alors le sous-arbre qu'il engendre a une taille d'au moins 2^k.*

Proposition 2.4. *Le rang d'un sommet est la profondeur du sous arbre qu'il engendre.*

Proposition 2.5. *Les opérations Unir et Trouver ont toutes deux une complexité $O(\log(n))$*

2ème heuristique : la compression de chemin

Lors des appels à trouver, on fait pointer chaque élément parcouru directement vers la racine de l'arbre.

Proposition 2.6. *La complexité de m opérations Unir ou Trouver est $O(m\alpha(n))$ où α est une fonction presque constante (par exemple $\alpha(n) = o(\log\log(n))$).*

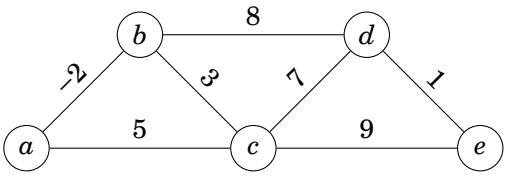
2.3 Algorithme de Kruskal

Algorithme 2.1 (Algorithme de Kruskal)**.**

Données : G, *poids*

$E \leftarrow \emptyset$
Initialiser une structure unir & trouver avec les sommets de G
Trier $A(G)$ par ordre croissant de poids
Pour chaque arête $(u, v) \in A(G)$ pris dans l'ordre des poids croissants :
    Si trouver $(u) \neq$ trouver (v) :
        $E \leftarrow E \cup \{(u, v)\}$
        union (u, v)
Renvoyer E

Exercice 2.4. Appliquer l'algorithme de Kruskal au graphe suivant :



Notation 2.5. On notera $G[E]$ le graphe ayant pour sommets $S(G)$ et arêtes E.

Proposition 2.7. *À tout instant la structure Unir & Trouver représente exactement les composantes connexes de* $G[E]$.

Proposition 2.8. *À tout instant,* $G[E]$ *est acyclique.*

Proposition 2.9. *À la fin de l'algorithme, deux sommets* u, v *adjacents de G sont dans la même composante connexe dans* $G[E]$. *En particulier, si G est connexe,* $G[E]$ *est connexe et E est un arbre couvrant.*

Proposition 2.10. *L'algorithme de Kruskal renvoie un arbre couvrant de poids minimal.*

Exercice 2.5. Proposer un algorithme pour trouver un arbre couvrant de poids maximal.

Proposition 2.11. *Le temps d'exécution de l'algorithme de Kruskal est en* $O(|A| \ln(S))$.

Exercice 2.6. Sur la feuille de code, compléter les lignes 44, 45, 46 et 48.

3 Couplage maximum dans un graphe biparti

3.1 Couplage maximum

Définition 3.1. Un couplage dans un graphe $G = (S, A)$ est un ensemble d'arêtes C tel que chaque sommet soit incident à au plus une arête de C. Soit $(u, v) \in C$, on dit que u et v sont couplés.

Définition 3.2. Un graphe $G = (S, A)$ est biparti si on peut partitionner S en S_1 et S_2 tels que $A \subseteq S_1 \times S_2$.

Définition 3.3. Un couplage maximum est un couplage qui contient le plus grand nombre d'arêtes possible. De façon équivalente, il couvre le plus grand nombre de sommets possible.

S'il couvre tous les sommets, il est dit parfait.

3.2 Chemin augmentant

Définition 3.4. Soit $G = (S, A)$ un graphe et C un couplage de G.

Un chemin augmentant est un chemin simple de G dont les extrémités ne sont pas couvertes et dont les arêtes appartiennent alternativement à $A \setminus C$ et à C.

Lemme 3.1. *Soit C un couplage dans un graphe* $G = (S, A)$ *et P un chemin augmentant* (*où, par abus, P désigne la suite de sommets et la suite d'arêtes*). *Alors* $C' = C \Delta P = (C \cup P) \setminus (C \cap P)$ *est un couplage et* $|C'| = |C| + 1$.

Δ *est l'opérateur de différence symétrique.*

Lemme 3.2. *Soit* C, C' *deux couplages dans un graphe* $G = (S, A)$ *avec* $|C'| > |C|$. *Alors, C possède un chemin augmentant.*

Théorème 3.3. *Un couplage est maximum si et seulement si il ne contient pas de chemin augmentant.*

3.3 Algorithme

Proposition 3.4. *Soit $G = (S, A)$ un graphe biparti avec $S = S_1 \cup S_2$ et $A = S_1 \times S_2$. Soit C un couplage.*

(1) *Un chemin augmentant de G a une extrémité dans S_1 et l'autre dans S_2.*

On oriente les arêtes de C de S_2 vers S_1 et les arêtes de $A \setminus C$ de S_1 vers S_2 pour obtenir G'.

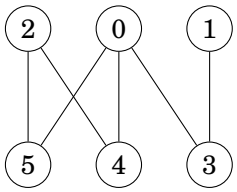
(2) *Un chemin augmentant de G devient un chemin de G' d'un sommet de S_1 non couplé à un sommet non couplé de S_2.*

(3) *Un chemin de G' d'un sommet non couplé de S_1 à un sommet non couplé de S_2 est un chemin augmentant de G.*

On peut donc déterminer si un graphe biparti admet un chemin augmentant à l'aide d'un parcours de graphe.

Proposition 3.5. *On a donc un algorithme pour trouver un couplage maximum* :
On part d'un couplage vide. Ensuite on cherche un chemin augmentant et on augmente le couplage. Puis on recommence tant que c'est possible.

Exercice 3.1. Appliquer cet algorithme au graphe suivant :



Exercice 3.2. Sur la feuille de code :

1. Compléter les lignes 56, 57 et 58.
2. Qu'est ce qui change si on remplace la ligne 77 par : `b = b || chemin_augmentant (g, i);` ?
3. Quelle est la complexité ?