

# Algorithmique avancée des graphes

<b>I</b>	<b>Arbre couvrant minimal</b>	<b>1</b>
I.1	Présentation du problème	1
I.2	Algorithme de Kruskal	1
I.3	Correction de l'algorithme de Kruskal	2
I.4	Complexité de l'algorithme de Kruskal	3
I.5	Prim ? TODO	3
<b>II</b>	<b>Kosaraju et composantes fortement connexes</b>	<b>3</b>
II.1	Rappel des définitions	3
II.2	Exemple	4
II.3	Rappels sur le parcours en profondeur	4
II.4	Graphe miroir	5
II.5	Algorithme de Kosaraju	6
<b>III</b>	<b>Couplage maximal dans un graphe biparti</b>	<b>6</b>
III.1	Problème	6
III.2	Chemin augmentant	6
III.3	Déterminer un chemin augmentant dans un graphe biparti	7
<b>IV</b>	<b>Exercices</b>	<b>8</b>
<b>V</b>	<b>Travaux pratiques</b>	<b>8</b>
V.1	Algorithme de Kruskal, union-find	9
V.2	Algorithme de Kosaraju et 2-SAT	13
V.3	Couplage maximal dans un graphe biparti	18
V.4	Algorithme de Ford-Fulkerson en C	24

On présente ici trois algorithmes qui complètent les notions vues en première année :

- le calcul d'un arbre couvrant de poids minimal dans un graphe pondéré
- le calcul des composantes fortement connexes dans un graphe orienté
- une notion de couplages maximale qui est un cas particulier d'un problème plus général de flot maximal

## I Arbre couvrant minimal

### I.1 Présentation du problème

Description informelle : on a un ensemble de maisons reliées par des routes, on cherche à poser des câbles le long des routes pour que toute paire de maison soit connectée. Quelle est la longueur minimale de câble à utiliser ?

On comprend assez vite qu'il faut que les câbles forment un graphe connexe et que pour des questions de réduction de coût, on peut supposer que les graphes sont acycliques. On cherche donc un arbre qui couvre chaque maison dont la somme des poids des arêtes, les longueurs des câbles, est minimale.

**Définition I.1** Soit  $G = (S, A)$  un graphe non orienté connexe on dit que  $T \subset A$  est un arbre couvrant de  $G$  lorsque :

- $\forall x \in S, \exists a \in T, x \in a$  : chaque sommet appartient à au moins une arête dans  $T$
- $(S, T)$  est un arbre, c'est-à-dire que c'est un sous-graphe acyclique et connexe

On notera ici  $\mathcal{T}(G)$  l'ensemble des arbres couvrants de  $G$ .

**Exemple** FIXME

Une manière naturelle d'obtenir un arbre couvrant est de faire un parcours quelconque ou de calculer les composantes connexes avec une structure union-find. Dans ce dernier cas, comme le graphe est connexe, on obtiendra directement un unique arbre dans la forêt qui est un arbre couvrant. C'est l'ordre de traitement des arêtes qui va aiguiller vers un arbre de  $\mathcal{T}(G)$  ou un autre.

**Définition 1.2** Soit  $G = (S, A, w)$  un graphe non orienté connexe et pondéré par  $w : A \rightarrow \mathbb{R}$ , on note  $w(T) = \sum_{a \in T} w(a)$  le poids d'un arbre couvrant de  $G$ .

Comme  $\mathcal{T}(G)$  est fini, il existe, au moins, un arbre  $T_0$  tel que

$$w(T_0) = \min_{T \in \mathcal{T}(G)} w(T)$$

On dit que  $T_0$  est un **arbre couvrant de poids minimal**.

**Remarque** En anglais, on parle de *minimum spanning tree*.

## I.2 Algorithme de Kruskal

Pour calculer un arbre couvrant de poids minimal, on va considérer le calcul des composantes connexes avec une structure union-find mais en traitant les arêtes dans l'ordre croissant de leurs poids : c'est l'algorithme de Kruskal.

### Algorithme - KRUSKAL

- Entrées :  
Un graphe non orienté pondéré connexe  $G = (S, A, w)$ .
- ★ Pour chaque  $x \in S$ 
  - `makeset(x)`
- ★ On trie  $A$  par ordre croissant de poids.
- ★ Pour chaque  $\{x, y\} \in A$  trié
  - Si `find(x) ≠ find(y)`
  - Alors `union(x, y)`
- ★ On renvoie l'unique arbre de la forêt.

## I.3 Correction de l'algorithme de Kruskal

On va montrer la correction d'une famille d'algorithmes à laquelle Kruskal appartient.

**Définition 1.3** On dit que  $T$  est une forêt minimale de  $G$  s'il existe  $T'$  arbre couvrant de poids minimal de  $G$  tel que  $T \subset T'$ .

**Remarque**  $\emptyset$  est ainsi une forêt minimale.

**Définition 1.4** Soit  $T$  une forêt minimale et  $a \in A$ . On dit que  $a$  est une arête **sûre** si  $T \cup \{a\}$  est encore une forêt minimale.

**Lemme 1.1** Soit  $T$  une forêt minimale qui n'est pas un arbre couvrant, il existe une arête sûre  $a$  telle que  $T \cup \{a\}$  soit encore une forêt minimale.

### ■ Preuve

Comme  $T$  est une forêt minimale, il existe  $T'$  arbre couvrant de poids minimal tel que  $T \subset T'$ . Comme  $T$  lui-même n'est pas un arbre couvrant, il existe  $a \in T' \setminus T$ . On a alors  $T \cup \{a\} \subset T'$  donc  $a$  est une arête sûre. ■

On en déduit un proto-algorithme de calcul d'un arbre couvrant de poids minimal :

**Algorithme** - ARBREMINQUELCONQUE

- Entrées :  
Un graphe non orienté pondéré connexe  $G = (S, A, w)$ .
- - ★ On pose  $T = \emptyset$
  - ★ Tant que  $T$  n'est pas un arbre-couvrant
    - Déterminer une arête sûre  $a \in E$
    - $T := T \cup \{a\}$
  - ★ Renvoyer  $T$

**Théorème 1.2** Cet algorithme renvoie un arbre couvrant de poids minimal.

**■ Preuve**

Cet algorithme vérifie directement l'invariant suivant :  $T$  est une forêt minimale. En effet, le choix d'une arête sûre permet de prolonger l'invariant.

Cet algorithme termine car il n'existe qu'un nombre fini d'arêtes à ajouter et comme une arête sûre ne peut pas l'être une fois qu'on l'a rajoutée, on ne peut pas faire plus d'itérations que le nombre d'arêtes.

L'algorithme renvoie donc un arbre couvrant de poids minimal. ■

**Remarque** Il s'agit d'un proto-algorithme car la partie critique est de déterminer une arête sûre et c'est la partie qui n'est pas explicitée pour le moment.

**Théorème 1.3** Si  $F$  est une forêt minimale non couvrante et  $e$  l'arête de plus petit poids reliant deux arbres de  $F$ , alors  $e$  est sûre pour  $F$ .

**■ Preuve**

Comme  $F$  est une forêt minimale, il existe  $T$  arbre couvrant de poids minimal tel que  $F \subset T$ .

Soit  $e \in T$ , auquel cas  $F \cup \{e\} \subset T$  est encore une forêt minimale. Soit  $e \notin T$  et comme  $T$  est un arbre,  $T \cup \{e\}$  possède un cycle. On sait que  $F \cup \{e\}$  est encore une forêt, donc ce cycle contient nécessairement une arête  $e' \in T \setminus F$ .

Par minimalité,  $w(e') \geq w(e)$ . Si on pose  $T' = (T \setminus \{e'\}) \cup \{e\}$  alors  $T'$  est un arbre couvrant car comme l'ajout de  $e$  à  $T$  induit un cycle contenant  $e'$ , les deux sommets de  $e'$  sont couverts par des arêtes dans ce cycle privé de  $e'$ . On en déduit de même que  $T'$  est connexe.

$T'$  est nécessairement acyclique car on a cassé le seul cycle contenu dans  $T \cup \{e\}$  en enlevant  $e'$ .

Reste que  $w(T') = w(T) - w(e') + w(e) \leq w(T)$  donc  $T'$  est un arbre couvrant de poids minimal.

Ainsi  $F \cup \{e\} \subset T'$  est une forêt minimale et  $e$  est sûre. ■

Comme  $\emptyset$  est une forêt minimale, on vient de valider l'invariant pour Kruskal qui est que la forêt disjointe est une forêt minimale.

**Corollaire 1.4** Kruskal renvoie un arbre couvrant de poids minimal.

**I.4 Complexité de l'algorithme de Kruskal**

On peut décomposer l'algorithme :

- La création avec `makeset` est en  $O(|S|)$
- Le tri des arêtes est en  $O(|A| \log |A|)$ .
- La boucle est en  $O(|A| \alpha(|S|))$  avec  $\alpha(|S|) = o(\log |A|)$

Comme  $G$  est connexe, on a  $|A| \geq |S| - 1$  et ainsi  $|S| = O(|A|)$ . Ainsi la complexité globale est en  $O(|A| \log |A|)$ .

**Remarque** Kruskal fait partie de ces algorithmes qui sont linéaires après avoir fait un tri. C'est un cas que l'on a déjà vu avec les algorithmes gloutons. D'ailleurs, on peut dire que Kruskal est un choix glouton d'arête sûre.

## I.5 Prim? TODO

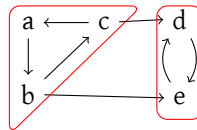
## II Kosaraju et composantes fortement connexes

### II.1 Rappel des définitions

Soit  $G = (S, A)$  un graphe **orienté** on note, pour  $x, y \in S$ ,  $x \rightsquigarrow y$  quand il existe un chemin dans  $G$  reliant  $x$  à  $y$ . On dit que  $y$  est *accessible* depuis  $x$ .

On note  $x \leftrightarrow y \iff x \rightsquigarrow y \wedge y \rightsquigarrow x$ . C'est la restriction symétrique de  $\rightsquigarrow$ . Comme  $\rightsquigarrow$  est réflexive et transitive, alors  $\leftrightarrow$  est une relation d'équivalence. Les classes d'équivalences pour  $\leftrightarrow$  sont appelées les **composantes fortement connexes** de  $G$ . On les note  $CFC(G) = S / \leftrightarrow$ .

Par exemple, sur le graphe :



On a les deux composantes fortement connexes  $\{a, b, c\}$  et  $\{d, e\}$ .

On remarque une différence fondamentale avec les composantes connexes, c'est qu'il peut y avoir des arrêtes entre deux composantes fortement connexes.

On cherche ici à déterminer un algorithme pour calculer les composantes fortement connexes d'un graphe.

### II.2 Exemple

TODO voir cours

### II.3 Rappels sur le parcours en profondeur

Comme on vient de le voir, le parcours en profondeur et ses temps d'entrée et de sorties sont très importants ici. On va donc faire des rappels sur ces notions.

On considère le programme suivant :

```
type statut = Inconnu | EnTraitement | Traite
```

```
type etat_dfs = {
  statut : statut array;
  mutable clock : int;
  entree : int array;
  sortie : int array
}
```

```
let tick etat =
  let t = etat.clock in
  etat.clock <- t+1;
  t
```

```
let rec dfs ladj etat x =
  if etat.statut.(x) <> Traite
  then begin
    etat.statut.(x) <- EnTraitement;
    etat.entree.(x) <- tick etat;
    List.iter (fun y ->
      if etat.statut.(y) = Inconnu
      then dfs ladj etat y) ladj.(x);
    etat.sortie.(x) <- tick etat;
    etat.statut.(x) <- Traite
  end
```

```

let initialise_dfs ladj =
  let n = Array.length ladj in
  {
    statut = Array.make n Inconnu;
    clock = 1;
    entree = Array.make n 0;
    sortie = Array.make n 0
  }

```

Le temps d'entrée est le moment où commence à traiter un sommet et son temps de sortie est le moment où on a fini de le traiter car on a vu tous ses descendants. On notera ici  $t_e(x)$  le temps d'entrée de  $x$  et  $t_s(x)$  son temps de sortie.

**Définition II.1** On considère un DFS d'un graphe  $G = (S, A)$  et  $x \in S$  découvert par ce parcours.

Soit  $y \in S$ , on dit que  $y$  est accessible par un chemin inconnu depuis  $x$  s'il existe un chemin de  $x$  à  $y$  ne passant que par des sommets de statut Inconnu au moment où on lance le DFS en  $x$ . On note  $x \rightsquigarrow_I y$ .

**Théorème II.1**  $x \rightsquigarrow_I y$  si et seulement si l'appel à DFS depuis  $x$  va appeler DFS sur  $y$  avant de se résoudre.

#### ■ Preuve

- $\Rightarrow$ ) : On va raisonner par récurrence sur la longueur du chemin inconnu.
  - ★ Initialisation : si le chemin inconnu est vide, c'est direct.
  - ★ Hérédité : si  $x \rightsquigarrow_I z \rightarrow y$  avec  $y$  inconnu et l'hypothèse de récurrence valide pour  $x \rightsquigarrow_I z$  alors au moment de l'appel au DFS sur  $z$ , on va forcément faire un appel au DFS sur  $y$ , voisin inconnu de  $z$ .
- $\Leftarrow$ ) : si on considère la chaîne des appels qui ont mené jusqu'à  $y$ , vu la condition sur le statut, ce sont nécessairement tous des sommets inconnus et ils forment un chemin, qui est donc un chemin inconnu.

■

**Théorème II.2** Soient  $x, y \in S$  tels que  $t_e(x) < t_e(y)$ .

Si  $x \rightsquigarrow_I y$ , alors  $t_f(y) < t_f(x)$ .

Sinon,  $t_f(x) < t_e(y)$ .

Autrement dit, soit  $[t_e(y); t_f(y)] \subset [t_e(x); t_f(x)]$ , soit  $[t_e(y); t_f(y)] \cap [t_e(x); t_f(x)] = \emptyset$ .

On dit que les temps sont bien parenthésés. En effet, si on considère le mot sur  $S$  avec les lettres  $(_x$  et  $)_x$  pour chaque sommet  $x$  et tel qu'on écrive la lettre  $(_x$  quand on note le temps d'entrée et  $)_x$  quand on note le temps de sortie, alors ce mot est bien parenthésé.

**Exemple** Sur l'exemple du graphe précédent (TODO ref précise) on pourrait avoir le mot  $(_a(b(c(d(e)e)d)c)b)_a$ .

#### ■ Preuve

Comme  $t_e(x) < t_e(y)$ , c'est qu'on a commencé le parcours en  $x$  avant de le commencer en  $y$ . Si  $x \rightsquigarrow_I y$  alors par le théorème précédent, on appelle le DFS sur  $y$  depuis l'appel du DFS sur  $x$ , donc le premier terminera avant le second et ainsi  $t_f(y) < t_f(x)$ .

Sinon, il n'est pas possible de rencontrer  $y$  en résolvant le DFS de  $x$ , donc on aura forcément fini de traiter  $x$  avant de commencer le parcours en  $y$ . Donc  $t_f(x) < t_e(y)$ .

■

**Définition II.2** Soit  $C \in CFC(G)$ , on note

$$t_e(C) = \min \{ t_e(x) \mid x \in C \}$$

$$t_f(C) = \max \{ t_f(x) \mid x \in C \}$$

On a alors une propriété de parenthésage des temps sur les composantes fortement connexes elles-mêmes.

**Théorème II.3** Soient  $C, C' \in CFC(G)$ .

Si il existe  $x \in C, y \in C'$  avec  $x \rightarrow y$ , alors  $t_f(C') < t_f(C)$ .

■ **Preuve**

On suppose qu'il existe  $x \in C, y \in C'$  avec  $x \rightarrow y$ .

Premier cas,  $t_e(C) < t_e(C')$ . On considère  $u \in C$  tel que  $t_e(C) = t_e(u)$ . On a alors forcément  $u \rightsquigarrow_I v$  pour tout  $v \in C \cup C'$ , en passant par  $x \rightarrow y$ . Ainsi  $x$  finit son DFS après tous les sommets dans  $C \cup C'$  donc  $t_f(C) = t_f(x) > t_f(C')$ .

Second cas,  $t_e(C) > t_e(C')$  si  $u \in C'$  tel que  $t_e(u) = t_e(C')$  alors on a visité tous les sommets de  $C'$  depuis  $u$ , donc  $t_f(u) = t_f(C')$  et on n'a rencontré aucun sommet de  $C$  car  $x \rightarrow y$  implique qu'il ne peut exister une arête de  $C'$  vers  $C$ . On a bien  $t_f(C) > t_f(C')$ . ■

## II.4 Graphe miroir

**Définition II.3** Soit  $G = (S, A)$  un graphe orienté, on appelle **graphe miroir** de  $G$  le graphe  $G^R = (S, A^R)$  où

$$\forall x, y \in S, (x, y) \in A \iff (y, x) \in A^R$$

Cela revient à renverser toutes les flèches du graphe  $G$ .

**Théorème II.4**  $CFC(G) = CFC(G^R)$

■ **Preuve**

On remarque que la relation  $x \leftrightarrow_G y \iff x \leftrightarrow_{G^R} y$ . Les deux relations ont donc *a fortiori* les mêmes classes d'équivalence. ■

## II.5 Algorithme de Kosaraju

**Algorithme - KOSARAJU**

• Entrées :

Un graphe orienté  $G = (S, A)$

- ★ On initialise l'état d'un DFS pour  $G$ .
- ★ Tant qu'il y a des sommets inconnus, on lance un DFS depuis un sommet inconnu.
- ★ On trie  $S$  par **ordre décroissant** de temps de sortie.
- ★ On initialise l'état d'un DFS pour  $G^R$ .
- ★  $Comp \leftarrow \emptyset$
- ★ Tant qu'il y a un sommet inconnu  $x$ 
  - On lance un DFS dans  $G^R$  à partir de  $x$  en notant les nouveaux sommets traités dans la liste  $C$ .
  - $Comp \leftarrow Comp \cup \{C\}$
- ★ On renvoie  $Comp$ .

**Théorème II.5**  $Comp = CFC(G)$ .

■ **Preuve**

Il suffit de montrer l'invariant  $Comp \subset CFC(G)$  pour la dernière boucle. Au départ, comme  $Comp = \emptyset$  il est trivialement vérifié et à la fin, comme on aura traité tous les sommets, on aura nécessairement  $Comp = CFC(G)$ .

Supposons donc qu'on a  $Comp \subset CFC(G)$  et qu'on relance un parcours dans  $G^R$  à partir de  $x$ . On sait que la composante  $\bar{x}$  contenant  $x$  est forcément dans les sommets que l'on va traiter :  $\bar{x} \subset C$ . Si, par l'absurde, il existe un sommet  $y \in C \setminus \bar{x}$ , alors  $y$  est dans une autre composante  $\bar{y}$ . On a traité  $y$  depuis  $x$ , donc  $t_e(x) < t_e(y)$ . Comme  $x \rightsquigarrow y$ , on a par le théorème précédent  $t_f(y) < t_f(x)$ . On a donc traité la composante  $\bar{y}$  dans un parcours précédent et donc  $y$  est traité. Contradiction. ■

**Remarque** La complexité de cet algorithme est dominée par les deux itérations de DFS, on est donc en  $O(|S| + |A|)$ .

### III Couplage maximal dans un graphe biparti

#### III.1 Problème

**Définition III.1** Soit  $G = (S, A)$  un graphe non orienté, on appelle **couplage** de  $G$  une partie  $C \subset A$  telle que  $\forall e, e' \in C, e \cap e' = \emptyset$ .

On dit qu'un couplage est **maximal** pour  $G$  quand il est de cardinal maximal.

Rappel :

**Définition III.2** Soit  $G = (S, A)$  un graphe non orienté, on dit que  $G$  est **biparti** lorsqu'il existe  $S_1, S_2 \subset S$  avec  $S_1 \cup S_2 = S$  et  $S_1 \cap S_2 = \emptyset$  et toutes les arêtes relient un sommet de  $S_1$  et un sommet de  $S_2$ .

On se pose alors la question de déterminer un couplage maximal dans un graphe biparti. C'est un problème classique d'appariement. On peut ainsi citer le cas où on a des élèves et des écoles. On met une arête entre un élève et une école quand les deux veulent de l'autre. Un couplage maximal est alors une manière de placer le maximum d'élèves dans une école.

#### III.2 Chemin augmentant

**Définition III.3** Soit  $C$  un couplage d'un graphe et  $x$  un sommet. On dit que  $x$  est libre **vis-à-vis** de  $C$  si  $x$  n'appartient pas à une arête de  $C$ .

**Définition III.4** Soit  $C \subset A$  un couplage d'un graphe.

Un chemin de  $x$  à  $y$  composé des arêtes  $(e_1, \dots, e_{2n+1})$  est dit **augmentant** si les arêtes paires  $e_{2i} \in C$ , les arêtes impaires  $e_{2i+1} \notin C$  et  $x$  et  $y$  sont libres pour  $C$ .

**Lemme III.1** Soit  $C, C'$  des couplages de  $G = (S, A)$ . On considère  $G' = (S, C \Delta C')$ .

Les composantes connexes de  $G'$  sont :

- soit des sommets isolés
- soit des cycles **de longueur paire** alternant entre arêtes de  $C$  et  $C'$
- soit des chemins alternant entre  $C$  et  $C'$  ayant des extrémités distinctes.

##### ■ Preuve

Il suffit de remarquer que les sommets de  $G'$  sont de degré au plus 2 car ils sont de degré au plus 1 dans  $(S, C)$  et  $(S, C')$ .

De plus, comme les arêtes d'un couplage ne peuvent avoir des extrémités en commun un chemin devra forcément alterner entre arêtes de  $C$  et de  $C'$ . Les cycles sont donc nécessairement de longueur paire. ■

**Théorème III.2** (Bergé 1957)  $C$  n'a pas de chemin augmentant, ssi  $C$  est un couplage maximal.

##### ■ Preuve

On va montrer la contraposée :  $C$  non maximal ssi  $C$  a un chemin augmentant.

⇒) Supposons que  $G$  dispose d'un chemin augmentant  $\varphi$ , on considère  $C'$  différence symétrique de  $C$  et des arêtes dans  $\varphi$ . Ainsi,  $C'$  contient les arêtes de  $\varphi$  qui ne sont pas dans  $C$ , comme  $\varphi$  commence et finit avec des arêtes qui ne sont pas dans  $C$ ,  $C'$  a une arête de plus que  $C$ .

De plus, comme  $\varphi$  est élémentaire et qu'ils relient deux sommets libres, on a l'assurance que  $C'$  est un couplage. Ainsi  $C$  n'est pas maximal.

⇒) Supposons que  $C$  non maximal, il existe  $C'$  tel que  $|C'| > |C|$  et si on considère  $G' = (S, C \Delta C')$ , il a une composante qui contient au moins une arête de plus dans  $C'$  que dans  $C$ . Ça ne peut donc être un cycle et c'est un chemin qui est par construction augmentant pour  $C$ .

**Remarque** Il s'agit ici d'un cas particulier du théorème de Bergé.

### III.3 Déterminer un chemin augmentant dans un graphe biparti

On considère ici un graphe biparti avec  $S = S_1 \cup S_2$ .

Pour déterminer un chemin augmentant pour  $C$ , on considère une orientation des arêtes  $\{x, y\}$  ainsi :

- $x \rightarrow y$  si  $x \in S_2, y \in S_1$  et  $(x, y) \in C$ .
- $y \rightarrow x$  sinon

On rajoute également deux sommets :

- un sommet source noté  $s$  avec  $s \rightarrow x$  pour tout sommet **libre** dans  $S_1$
- un sommet but noté  $t$  avec  $x \rightarrow t$  pour tout sommet **libre** dans  $S_2$ .

On remarque qu'un sommet non libre  $y$  de  $S_2$  est nécessairement de degré 1 et avec une arête  $y \rightarrow x$  où  $x$  non libre et  $\{x, y\} \in C$ .

S'il existe un chemin de  $s \rightsquigarrow t$  dans ce graphe orienté, alors il est de la forme :

$$s \rightarrow x_1 \rightarrow y_1 \cdots \rightarrow y_n \rightarrow t$$

avec :

- $x_1$  libre dans  $S_1$
- $y_n$  libre dans  $S_2$
- tous les autres  $x_i$  et  $y_j$  sont non libres (ok) et deux à deux distincts (pas facile là!)
- $\{x_i, y_i\} \notin C$
- $\{y_i, x_{i+1}\} \in C$

Le chemin est donc augmentant

## IV Exercices

**Exercice 1** On considère un chemin entre deux sommets  $x$  et  $y$  dans un graphe non orienté pondéré. La largeur de ce chemin est le plus petit poids des arêtes présentes dans ce chemin. Le chemin vide de  $x$  à  $x$  est de largeur  $+\infty$ .

La distance de goulot d'étranglement entre  $x$  et  $y$  est la largeur maximale d'un chemin de  $x$  à  $y$ . S'il n'en existe pas, cette distance est  $-\infty$ .

1. Prouver que l'arbre couvrant de poids **maximal** contient les chemins les plus larges entre toute paire de sommets.
2. Décrire un algorithme pour résoudre en temps  $O(|S| + |A|)$  le problème suivant : étant donné un graphe non orienté pondéré  $G = (S, A)$ ,  $x, y \in S$  et  $W \in \mathbb{R}$ , est-ce que la distance de goulot d'étranglement entre  $x$  et  $y$  est inférieure ou égale à  $W$ .
3. On suppose que la distance de goulot d'étranglement entre  $x$  et  $y$  est  $B$ .
  1. Prouver que la suppression d'une arête de poids inférieur à  $B$  ne change pas cette distance.
  2. Prouver que la contraction d'une arête de poids plus grand que  $B$  ne change pas cette distance. La contraction d'une arête  $(u, v)$  revient à identifier  $u$  et  $v$ , si cette contraction crée des arêtes parallèles, on ne conservera que l'arête de plus grand poids.

### ■ Preuve

1. On considère un arbre couvrant maximal  $T$  et deux sommets  $x, y$ . Supposons par l'absurde que  $T$  ne contienne pas le chemin le plus large de  $x$  à  $y$ . On considère alors le chemin dans  $T$  entre  $x$  et  $y$ , son arête de plus petit poids est  $e = \{a, b\}$ . On considère  $T' = T \setminus \{e\}$  qui n'est plus connexe. Le chemin le plus large entre  $x$  et  $y$  contient ainsi forcément une arête  $e' \notin T'$  et on peut considérer  $T' \cup \{e'\} = T''$  qui est un arbre couvrant avec  $w(T'') = w(T) + w(e') - w(e) > w(T)$  car  $w(e') \geq \text{largeur} > w(e)$ . Contradiction.



## **V Travaux pratiques**

## V.1 Algorithme de Kruskal, union-find

### V.1.i Union-Find

On va définir un type

```
struct union_find {
    int *parent;
    int *rank;
    int nelements;
};
typedef struct union_find union_find;
```

**Question V.1.1** Écrire des fonctions de prototypes :

```
union_find uf_makesets(int n);
int uf_find(union_find uf, int x);
void uf_union(union_find uf, int x, int y);
void uf_free(union_find uf);
```

Ces fonctions vont réaliser les opérations vues en classe. Pour `makesets` on va réaliser des `makeset` des éléments de 0 à  $n-1$  d'un coup en créant la valeur de type `union_find` (et notamment en allouant la mémoire). `union_free` va alors libérer toute la mémoire allouée dans une telle structure.

**Question V.1.2** Écrire des jeux de tests pour ces fonctions. Avec des asserts par exemple.

**Question V.1.3** On veut comparer les différentes optimisations de cette structure de données. Pour cela, on rajoute deux nouveaux paramètres booléens `compress` et `weighted` qui vont indiquer respectivement si on effectue de la compression de chemin ou si on utilise les rangs pour équilibrer lors des unions. Reprendre les fonctions précédentes pour qu'elles tiennent compte de ces paramètres. On pourra remplacer le prototype de `makesets` pour initialiser les valeurs : `union_find uf_makesets(int n, bool compress, bool weighted);`

**Question V.1.4** Écrire une fonction

```
int *random_tests(int n, int m);
```

qui génère un jeu de test aléatoire de  $m$  unions pour  $n$  valeurs. Ce jeu de test sera représenté par un tableau `t` de  $2m$  entiers indiquant indiquant qu'on souhaite effectuer des unions entre `t[2*i]` et `t[2*i+1]`. **On ne demande pas que les unions soient valides ici.**

*Rappel* : on peut utiliser `rand() % A` pour générer une valeur aléatoire entière entre 0 et  $A-1$ .

**Question V.1.5** Écrire une fonction

```
void apply_tests(union_find f, int *tests, int m);
```

qui va appliquer les tests qu'on vient de générer sur un `union_find` qui aura été initialisé par le bon appel à `uf_makesets`.

**Attention** ici il faudra faire attention à ne faire l'union que si elle est valide, c'est-à-dire uniquement lorsqu'elle fusionne deux arbres de la forêt.

Il est possible de mesurer des temps simplement en C à l'aide de la fonction `clock_t clock();` définie dans `time.h`.

Cette fonction renvoie un entier donnant le nombre de cycle d'horloges, une constante `CLOCKS_PER_SEC` permet alors d'en déduire un nombre de secondes ainsi :

```

clock_t start = clock();
// tache à mesurer en temps
clock_t end = clock();
// le nombre flottant de secondes écoulées
double elapsed = ((double)(end-start)/CLOCKS_PER_SEC);

```

**Question V.1.6** En déduire des tracé de comparaison du coût moyen en fonction de  $n$  pour  $2n$  unions. On comparera avec/sans l'union pondérée et la compression de chemins (quatre choix donc). Attention qu'il sera sûrement impossible de mesurer le temps pris par la version sans optimisation pour de grandes valeurs de  $n$ .

**Remarque** Pour tracer des courbes, on peut utiliser `gnuplot` ou faire un export vers Python avec `matplotlib`. On peut même faire un export en CSV dans la console et copier-coller dans Google Docs. La partie suivante présente rapidement `Gnuplot`.

### V.1.i.a Gnuplot in a nutshell

Si on a un fichier `data.dat` avec des valeurs séparées en colonnes par des tabulations ('`\t`'). Par exemple :

```

0  0  0
1  1  1
2  4  8
3  9  27
4  16 64

```

on peut procéder ainsi pour afficher un tracé avec `gnuplot` :

```

$ gnuplot
gnuplot> plot "data.dat"

```

Par défaut, il va tracer la deuxième colonne en fonction de la première.  
Pour tracer la colonne  $j$  en fonction de la colonne  $i$  :

```
gnuplot> plot "data.dat" using i:j
```

On peut demander de tracer des lignes avec :

```
gnuplot> plot "data.dat" using i:j with linespoint
```

Tout ou presque peut être abrégé donc la commande précédente peut s'écrire :

```
gnuplot> p "data.dat" u i:j w lp
```

Pour tracer plusieurs courbes, on peut mettre les descriptions à la suite :

```
gnuplot> p "data.dat" u 1:2 w lp, "data.dat" u 1:3 w lp, "data.dat" u 1:4 w lp
```

On peut aussi utiliser des boucles pour faire ça :

```
gnuplot> p for [col=2:4] "data.dat" using 1:col w lp
```

Pour écrire un fichier, on peut préciser un terminal de sortie, par exemple en png, avec :

```
gnuplot> set terminal pngcairo size 500,500
gnuplot> set output "fichier.png"

```

### V.1.ii Arbre couvrant de poids minimal

On considère un graphe de points du plan dont les arêtes sont données dans un tableau :

```

struct edge {
    int u;
    int v;
};
typedef struct edge edge;

struct vertex {
    double x;
    double y;
};
typedef struct vertex vertex;

struct graph {
    vertex *vertices;
    int nvertices;
    edge *edges;
    int nedges;
};
typedef struct graph graph;

```

Ainsi, l'arête  $e$  relie les sommets d'indice  $e.u$  et  $e.v$ . Un sommet  $v$  correspond au point du plan  $(v.x, v.y)$ .

#### Question V.1.7 Écrire une fonction

```
double w(graph *g, edge *e);
```

qui renvoie la distance euclidienne entre les deux sommets donnés par une arête.

**Attention** pour avoir `sqrt` il faudra `math.h` et lier avec `-lm`.

#### Question V.1.8 Écrire une fonction

```
graph randomize(int n, int m, double w, double h);
```

qui crée un graphe aléatoirement avec  $n$  sommets et  $m$  arêtes. Les sommets auront des coordonnées dans  $[0, w] \times [0, h]$ .

**Note** : pour  $N$  assez grand, on peut générer aléatoirement un double dans  $[0, w]$  ainsi : `w * ((double) (rand() % (N+1)) / N)`

**Question V.1.9** Pour trier les arêtes on va utiliser `qsort` de `stdlib.h`. Cette fonction pose plusieurs problèmes :  
 \* il faut faire des casts vers `void` \* explicitement \* comme elle ne prend pas de paramètre pour la comparaison, on doit faire des variables globales.

On vous demande ainsi d'ajouter dans votre fichier :

```

graph *global_graph = NULL;
int compare_edges(const void *e1, const void *e2)
{
    double w1 = w(global_graph, (edge *)e1);
    double w2 = w(global_graph, (edge *)e2);
    if (w1 < w2)
        return -1;
    if (w1 > w2)
        return 1;
    return 0;
}

```

et pour trier les arêtes du graphe  $g$ , on pourra alors écrire :

```

global_graph = g;
qsort(g->edges, g->nedges, sizeof(edge), compare_edges);

```

**Question V.1.10** On sait qu'un arbre couvrant contient `nvertices-1` arêtes, on va donc stocker un arbre couvrant comme un tableau d'entiers `mst` de cette longueur. `mst[i]` sera l'indice de la *i*ème arête dans l'arbre.

Écrire ainsi une fonction

```
int *kruskal(graph *g);
```

qui renvoie le tableau `mst` en l'ayant alloué préalablement.

On s'assurera que le graphe est connexe en vérifiant que la structure `union_find` utilisée ne contient qu'un seul arbre.

**Question V.1.11** Faire des essais de graphes aléatoires en affichant le graphe et par-dessus dans une autre couleur l'arbre couvrant de poids minimal obtenu.

## V.2 Algorithme de Kosaraju et 2-SAT

### V.2.i Kosaraju

On va considérer un type graphe où les sommets ne sont pas forcément des entiers et où on peut stocker une table de hachage pour faire la correspondance entre la valeur d'un sommet et son indice.

```
OCaml
type 'a graphe = {
  sommets : 'a array;
  sommets_indices : ('a, int) Hashtbl.t;
  ladj : int list array
}
```

**Question V.2.1** Écrire une fonction qui étant donné un tableau `t` va construire la table des associations `t.(i)` → `i` qui permettra ainsi de remonter du tableau à l'indice.

**Note** C'est le moment de retourner apprendre par coeur la doc de OCaml <https://v2.ocaml.org/api/Hashtbl.html>

#### ■ Preuve

```
OCaml
let table_indices (t:'a array) : ('a, int) Hashtbl.t =
  let h = Hashtbl.create (Array.length t) in
  for i = 0 to Array.length t - 1 do
    Hashtbl.add h t.(i) i
  done;
  h
```

**Question V.2.2** Écrire une fonction `indice` qui prend un graphe et un sommet et renvoie son indice en utilisant la table `sommets_indices`.

#### ■ Preuve

```
OCaml
let indice g a = Hashtbl.find g.sommets_indices a
```

**Question V.2.3** En déduire une fonction `cree_graphe` qui va prendre un tableau `sommets` et créer un graphe sans arêtes dont ce sont les sommets.

#### ■ Preuve

```
OCaml
let cree_graphe sommets =
  {
    sommets = sommets;
    sommets_indices = table_indices sommets;
    ladj = Array.make (Array.length sommets) []
  }
```

**Question V.2.4** Écrire une fonction `ajoute_arete` telle que `ajoute_arete g a b` ajoute l'arête `a → b` où les sommets sont donnés par leur valeur et non pas par leur indice.

#### ■ Preuve

```

OCaml | let ajoute_arete g a b =
        let ia = indice g a in
        let ib = indice g b in
        if not (List.mem ib g.ladj.(ia))
        then g.ladj.(ia) <- ib :: g.ladj.(ia)

```

Les deux cellules suivantes permettent de définir le graphe vu dans le cours en exemple.

```

OCaml | let ex_cours = cree_graphe [['a';'b';'c';'d';'e';'f';'g';'h';'i';'j';'k';'l';'m';'n';'o';'p']]

OCaml | List.iter (fun (a,b) -> ajoute_arete ex_cours a b)
        [(('a','b'); ('b','f')); (('g','a'); ('f','g'));
         (('c','h'); ('g','c')); ('d','c'); ('h','d');
         (('e','f'); ('f','l')); ('e','i'); ('g','k');
         (('h','l'); ('i','n')); ('j','m'); ('j','k');
         (('k','l'); ('k','h')); ('l','o'); ('l','p');
         ('m','i'); ('n','j'); ('n','o'); ('o','k')]]

```

**Question V.2.5** Écrire une fonction `graphe_miroir` qui renvoie le graphe miroir du graphe donné, c'est-à-dire qui renverse toutes les arêtes.

#### ■ Preuve

```

OCaml | let graphe_miroir g =
        let gr = {
            sommets = g.sommets;
            sommets_indices = g.sommets_indices;
            ladj = Array.make (Array.length g.sommets) []
        } in
        for i = 0 to Array.length g.sommets - 1 do
            List.iter (fun j -> gr.ladj.(j) <- i :: gr.ladj.(j)) g.ladj.(i)
        done;
        gr

```

**Question V.2.6** Écrire une fonction `dfs` qui va parcourir en profondeur un graphe et appliquer une fonction de **post-traitement**. Pas besoin de noter les temps de sortie puisqu'ils correspondent au moment où on effectue ce traitement.

#### ■ Preuve

```

OCaml | let rec dfs g trait visites x =
        if not visites.(x)
        then begin
            visites.(x) <- true;
            List.iter (fun y ->
                if not visites.(y)
                then dfs g trait visites y
            ) g.ladj.(x);
            trait x
        end

```

**Question V.2.7** En déduire une fonction `kosaraju` qui renvoie un tableau indiquant le numéro de la composante fortement connexe de chaque sommet.

Attention, cette fonction va être un peu compliquée, on indique les grandes lignes ici :

- on initialise un tableau de visites
- on va construire une liste `ordre` des indices de sommets rencontrés lors d'un premier DFS de tout le graphe grâce à un post-traitement (avec une référence)
- on calcule  $G^R$  et on initialise un tableau de visites
- on fait des DFS dans  $G^R$  mais en prenant les sommets depuis la liste `ordre`. En post-traitement, on va remplir un tableau `comp` où `comp.(x)` sera le numéro de la composante connexe de `x`.

On commence par numéroter les composantes à 1.

#### ■ Preuve

OCaml

```

let kosaraju g =
  let n = Array.length g.sommets in
  let ordre = ref [] in
  let visites = Array.make n false in
  for i = 0 to n-1 do
    if not visites.(i)
    then dfs g (fun x -> ordre := x :: !ordre) visites i
  done;
  let gr = graphe_miroir g in
  let visites = Array.make n false in
  let ncomp = ref 0 in
  let comp = Array.make n 0 in
  List.iter (fun ia ->
    if not visites.(ia)
    then begin
      incr ncomp;
      dfs gr (fun x -> comp.(x) <- !ncomp) visites ia
    end
  ) !ordre;
  comp

```

**Question V.2.8** Cette représentation des composantes va nous être utile mais elle n'est pas très maniable. Écrire une fonction `listes_comp` telle que `listes_comp sommets comp` renvoie le tableau des composantes données sous forme de listes de sommets.

#### ■ Preuve

OCaml

```

let listes_comp sommets comp =
  let n = Array.fold_left max 0 comp in
  let comp_l = Array.make n [] in
  for i = 0 to Array.length comp - 1 do
    let n = comp.(i) in
    comp_l.(n-1) <- sommets.(i) :: comp_l.(n-1)
  done;
  comp_l

```

**Question V.2.9** On va résoudre 2-SAT en utilisant l'algorithme de Kosaraju. Tout d'abord on rappelle que les formules de 2-SAT ne comportent que deux littéraux dans chaque clause.

Exemple :  $f = (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b) \wedge (a \vee \neg c)$

On reprend un type proche de ce qui a été fait l'an dernier, sauf que les clauses sont des couples de littéraux.



```
OCaml | type lit = Pos of char | Neg of char
      | type clause = lit * lit
      | type formule = clause list
```

La formule donnée au dessus s'écrit alors :

```
OCaml | let f = [ (Pos 'a', Neg 'b'); (Neg 'a', Pos 'b'); (Neg 'a', Neg 'b'); (Pos 'a', Neg 'c') ]
```

**Question V.2.10** Écrire une fonction `label : lit -> char` qui renvoie l'étiquette d'un littéral et une fonction `neg : lit -> lit` qui renvoie le littéral opposé d'un littéral donné.

#### ■ Preuve

```
OCaml | let label = function Pos a | Neg a -> a
```

```
OCaml | let neg = function Pos a -> Neg a | Neg a -> Pos a
```

■

**Question V.2.11** Écrire une fonction `variables` qui renvoie les étiquettes de tous les littéraux **sans répétitions**. On pourra utiliser `List.sort_uniq Stdlib.compare`.

#### ■ Preuve

```
OCaml | let rec variables f =
      |> let rec var_aux f =
      |> match f with
      |> [] -> []
      |> c::f' -> var_clause c @ var_aux f'
      |> and var_clause (l1,l2) = [label l1; label l2]
      |> in List.sort_uniq Stdlib.compare (var_aux f)
```

■

**Question V.2.12** On peut écrire  $a \vee b$  sous la forme implicative  $(\neg a \rightarrow b) \wedge (\neg b \rightarrow a)$ .

En faisant cela, on peut alors construire un graphe sur les littéraux où  $l \rightarrow l'$  quand cette implication apparaît dans la formule.

La formule  $f$  donne alors le graphe d'implication suivant :

Écrire une fonction `implication_graphe` qui étant donnée  $f$  renvoie ce graphe.

#### ■ Preuve

```
OCaml | let implication_graphe f =
      |> let v = variables f in
      |> let s = List.concat (List.map (fun x -> [Pos x; Neg x]) v) in
      |> let sommets = Array.of_list s in
      |> let g = cree_graphe sommets in
      |> List.iter (fun (a,b) ->
      |> ajoute_arete g (neg a) b;
      |> ajoute_arete g (neg b) a
      |> ) f;
      |> g
```

■

On cherche alors une valuation des variables qui soit telle qu'il n'y ait pas une arête  $\top \rightarrow \perp$  entre un littéral vrai et un littéral faux. En effet, toutes les autres affectations vérifieront la clause correspondante. Ainsi, tous les littéraux d'une composante fortement connexe doivent nécessairement avoir la même valeur de vérité. Si  $C \rightarrow C'$  pour deux composantes, on ne peut pas affecter la valeur vraie à  $C$  et fausse à  $C'$ . On remarque que le graphe est symétrique par contraposition, donc il existe une composante niée  $\neg C$  pour chaque composante  $C$ .

L'idée pour résoudre 2-SAT est donc de calculer les composantes fortement connexes et de remonter leur DAG en affectant vraie à la première composante rencontrée entre  $C$  et  $\neg C$ .

Pour résoudre 2-SAT, on va appliquer Kosaraju sur le graphe d'implication de la formule.

Exemple pour  $f$  :

**O**Caml `kosaraju (implication_graphe f)`

Sur ce graphe on remarque que si deux littéraux opposés sont dans la même composante fortement connexe, alors la formule est insoluble. Si ce n'est pas le cas, on peut démontrer que la valuation suivante fonctionne :

$$\forall a \in V, v(a) = \begin{cases} \top & \text{si } comp(a) > comp(\neg a) \\ \perp & \text{sinon} \end{cases}$$

Cette affirmation repose sur deux propriétés :

- les composantes sont numérotées dans l'ordre croissant du tri topologique de leur DAG (graphe acyclique orienté)
- le graphe d'implication possède une symétrie par contraposition : si  $a_1 \rightarrow a_2 \cdots \rightarrow a_n$  alors  $\neg a_n \rightarrow \neg a_{n-1} \cdots \rightarrow \neg a_1$ .

**Question V.2.13** En déduire une fonction `resout_2sat` qui étant donné une formule renvoie `None` si elle est insoluble, et `Some l` où  $l$  est une liste de couples (variable, valeur) représentant une valuation solution.

#### ■ Preuve

**O**Caml

```

let resout_2sat f =
  let g = implication_graphe f in
  let comp = kosaraju g in
  let n = Array.length g.sommets / 2 in
  let valuation = ref [] in
  let insoluble = ref false in
  for i = 0 to n-1 do
    let a = label g.sommets.(2*i) in
    if comp.(2*i) = comp.(2*i+1)
    then insoluble := true;
  done;
  if !insoluble
  then None
  else Some !valuation

```

■

## V.3 Couplage maximal dans un graphe biparti

### V.3.i Graphe biparti

On va considérer ici un graphe biparti donné par une liste d'arêtes sous la forme (source, but). Voici un exemple :

**OCaml** `[ (0,3); (1,3); (1,4); (2,3); (2,4) ]`

En commençant à numéroté à 0, on peut donc supposer que le nombre de sommets est égal au plus grand entier apparaissant dans un couple plus un.

**Question V.3.1** Écrire une fonction `nombre_sommets : (int * int) list -> int` qui calcule ce nombre.

#### ■ Preuve

**OCaml** `let rec nombre_sommets l =  
 match l with  
 | [] -> 0  
 | (a,b)::q -> let n = nombre_sommets q in  
 max (1 + max a b) n`

On va maintenant récupérer la liste des premières composantes et la liste des secondes composantes **sans répétition**.

Pour cela, on commence par écrire une fonction permettant de réaliser `t :: q` en omettant `t` en cas de répétition.

**Question V.3.2** Écrire une fonction `cons_uniq : 'a -> 'a list -> 'a list` telle que `cons_uniq t q` renvoie `t :: q` si `t` n'est pas dans `q` et `q` sinon.

#### ■ Preuve

**OCaml** `let cons_uniq t q =  
 if List.mem t q  
 then q  
 else t::q`

**Question V.3.3** Écrire une fonction `separe_liste : (int * int) list -> int list * int list` qui prend en argument une liste de couple d'entiers et renvoie le couple des listes **sans répétitions** des premières et secondes composantes.

#### ■ Preuve

**OCaml** `let rec separe_liste l =  
 match l with  
 | [] -> [], []  
 | (a,b)::q -> let l1, l2 = separe_liste q in  
 (cons_uniq a l1, cons_uniq b l2)`

**Question V.3.4** Écrire une fonction `intersection : 'a list -> 'a list -> 'a list` qui prend en arguments deux listes sans répétitions et renvoie une liste contenant les éléments présents dans les deux listes.

## ■ Preuve

```
OCaml | let rec intersection l1 l2 =
        List.filter (fun x -> List.mem x l2) l1
```

**Question V.3.5** En déduire une fonction `biparti : (int * int) list -> bool` qui vérifie qu'une liste d'arêtes permet effectivement de représenter un graphe biparti où les sommets sont d'un côté les premières composantes et de l'autre les secondes.

## ■ Preuve

```
OCaml | let biparti l =
        let l1, l2 = separe_liste l in intersection l1 l2 = []
```

De cette liste d'arêtes représentant un graphe biparti à  $n$  sommets, on va en déduire le graphe lui-même en le représentant sous la forme d'un couple `(ladj, sources)` où `ladj` est une représentation en liste d'adjacences et `sources` est un tableau de booléen indiquant si un sommet appartient à la première composante du graphe biparti, i.e. à la première composante d'un des couples d'arêtes.

Ainsi, les sommets qui n'apparaissent pas dans les arêtes sont associés implicitement à la seconde composante, ce qui ne sera pas gênant dans la suite.

**Question V.3.6** Écrire une fonction `graphe_aretes : (int * int) list -> int list array` qui renvoie le tableau des listes d'adjacence d'un graphe **non orienté** donné sous la forme d'une liste d'arêtes.

## ■ Preuve

```
OCaml | let graphe_aretes l =
        let n = nombre_sommets l in
        let ladj = Array.make n [] in
        List.iter (fun (a,b) ->
            ladj.(a) <- b :: ladj.(a); ladj.(b) <- a :: ladj.(b)) l;
        ladj
```

**Question V.3.7** En déduire une fonction `graphe_biparti : (int * int) list -> int list array * bool array` qui renvoie le couple `(ladj, sources)` représentant le graphe biparti.

## ■ Preuve

```
OCaml | let graphe_biparti l =
        let g = graphe_aretes l in
        let s1, _ = separe_liste l in
        let sources = Array.make (Array.length g) false in
        List.iter (fun i -> sources.(i) <- true) s1;
        g, sources
```

## V.3.ii Couplage maximal

On va programmer l'algorithme pour déterminer un couplage maximal par des bascules successives de chemins augmentant (voir preuve du théorème de Berge).

On va adopter deux représentations d'un couplage. La représentation élémentaire comme sous-liste de la liste d'arêtes du graphe.

Par exemple  $[(0,3); (1,4)]$  est un couplage pour la liste d'arêtes données plus haut.

**Question V.3.8** Écrire une fonction `est_couplage : int list -> bool` qui vérifie si une liste d'arêtes est un couplage en vérifiant qu'il n'y a pas d'arêtes coincidentes.  
 Votre fonction devra être de complexité **linéaire**.

#### ■ Preuve

OCaml

```
let est_couplage el =
  let n = nombre_sommets el in
  let libres = Array.make n true in
  let couplage = ref true in
  List.iter (fun (x,y) ->
    couplage := !couplage && libres.(x) && libres.(y);
    libres.(x) <- false; libres.(y) <- false) el;
  !couplage
```

L'autre représentation est un tableau indiquant pour un sommet  $i$ , soit `Some j` quand on a une arête  $\{i, j\}$  dans le couplage et `None` sinon. En effet, un couplage réalise une fonction partielle involutive de  $A \rightarrow A$ .

Ainsi, le tableau `[| Some 3; Some 4; None; Some 0; Some 1 |]` permet de représenter le couplage  $[(0,3); (1,4)]$  en considérant que les sommets sont dans `[|0; 4|]`.

**Question V.3.9** Écrire des fonctions permettant de passer d'une représentation à une autre :

- `couplage_liste_vers_tab : (int * int) list -> int option array`
- `couplage_tab_vers_liste : bool array -> int option array -> (int * int) list` ici, on a besoin de connaître les sources pour choisir comment placer les arêtes.

#### ■ Preuve

OCaml

```
let couplage_liste_vers_tab l =
  let n = nombre_sommets l in
  let couplage = Array.make n None in
  List.iter (fun (a,b) ->
    couplage.(a) <- Some b;
    couplage.(b) <- Some a) l;
  couplage

let couplage_tab_vers_liste sources couplage =
  let cpl_aretes = ref [] in
  let n = Array.length sources in
  for i = 0 to n-1 do
    if sources.(i)
    then match couplage.(i) with
      | None -> ()
      | Some j -> cpl_aretes := (i,j) :: !cpl_aretes
  done;
  !cpl_aretes
```

On va définir ici le graphe résiduel associé à un couplage de manière implicite avec une fonction permettant de décider si une arête  $x \rightarrow y$  est dans le graphe résiduel.

**Attention** il peut y avoir des arêtes  $x \rightarrow z$  quand  $(x, y)$  est dans le couplage.

**Question V.3.10** Écrire une fonction  
`arete_residuelle : bool array -> int option array -> int -> int -> bool`  
 qui permet, en appelant `arete_residuelle sources couplage x y` permet de décider si  $x \rightarrow y$  est dans le graphe résiduel.

## ■ Preuve

```

OCaml
let arete_residuelle sources couplage x y =
  if sources.(x)
  then couplage.(x) <> Some y
  else couplage.(x) = Some y

```

Pour chercher un chemin augmentant, on va commencer par effectuer un parcours en profondeur récursif dans un graphe avec une notion d'arête implicite, une fonction comme `arete_residuelle`, et on va remplir un tableau de prédécesseur où  $x$  est le prédécesseur de  $y$  si c'est le DFS depuis  $x$  qui a appelé le DFS depuis  $y$ .

**Dans toute la suite, on suppose que le prédécesseur de  $x$  est  $x$  lui-même quand on a lancé le DFS initialement depuis le sommet  $x$ .**

**Question V.3.11** Écrire une fonction

`dfs : int list array -> (int -> int -> bool) -> int option array -> int -> unit`  
 telle que `dfs g est_arete pred x` effectue un DFS depuis le sommet  $x$  dans le graphe donné sous forme de listes d'adjacence par  $g$ , avec une fonction `est_arete` permettant de tester si une arête est à considérer dans le parcours et en remplissant le tableau des prédécesseurs `pred`, qui est un `int option array` car la valeur vaut `None` tant que le sommet n'est pas découvert.

*Rappel* : on a spécifié que `pred.(x)` valait `Some x` si  $x$  était un des sommets initiaux sur lesquels on a commencé le DFS.

## ■ Preuve

```

OCaml
let rec dfs_impl g est_arete pred x =
  List.iter (fun y ->
    if pred.(y) = None && est_arete x y
    then begin
      pred.(y) <- Some x;
      dfs_impl g est_arete pred y
    end)
  g.(x)

```

**Question V.3.12** Écrire une fonction

`remonte : int option array -> int -> int list`  
 telle que `remonte pred x` renvoie la liste des sommets allant du point de départ d'un DFS ayant rempli `pred` jusqu'à  $x$  en remontant la relation de prédécesseurs.

## ■ Preuve

```

OCaml
let remonte pred x =
  let rec aux pred y acc =
    match pred.(y) with
    | None -> failwith "Impossible"
    | Some x when x = y -> y::acc
    | Some x -> aux pred x (y::acc)
  in
  aux pred x []

```

On considère qu'on a effectué un DFS dans le graphe résiduel associé à un couplage depuis un sommet et on cherche maintenant à écrire une fonction qui permet déterminer s'il existe un chemin depuis ce sommet vers un sommet **libre** dans les cibles, i.e. les sommets **qui ne sont pas des sources**.

**Question V.3.13** Écrire une fonction

`cherche_chemin : int option array -> bool array -> int option array -> int -> int list option`

telle que `cherche_chemin pred sources couplage depart` renvoie `Some phi` si `phi` est la liste des sommets visités dans un chemin du graphe résiduel de `depart` un sommet cible **libre** pour couplage et `None` sinon.

## ■ Preuve

```

OCaml
let cherche_chemin pred sources couplage depart =
  let rec aux i =
    if i = Array.length sources
    then None
    else if not sources.(i) && couplage.(i) = None
           && pred.(i) <> None
    then let phi = remonte pred i in
         if List.hd phi = depart
         then Some phi
         else aux (i+1)
    else aux (i+1)
  in
  aux 0

```

**Question V.3.14** Écrire une fonction

`chemin_augmentant : int list array -> bool array -> int option array -> int -> int list option`

telle que `chemin_augmentant g sources couplage depart` renvoie `Some phi` où `phi` est la liste des sommets d'un chemin augmentant issu de la source `depart` (et donc arrivant nécessairement dans un sommet cible **libre**).

## ■ Preuve

```

OCaml
let chemin_augmentant g sources couplage depart =
  let n = Array.length g in
  let pred = Array.make n None in
  pred.(depart) <- Some depart;
  dfs_impl g (arete_residuelle sources couplage) pred depart;
  cherche_chemin pred sources couplage depart

```

**Question V.3.15** Écrire une fonction

`bascule_chemin : int option array -> int list -> unit`

telle que `bascule_chemin couplage chemin` où `chemin` est un chemin augmentant pour le couplage, va faire la bascule de toutes les arêtes de `chemin` : celles dans le couplage sont enlevées et celles qui n'y sont pas sont rajoutées.

**Indice** : seule la parité permet de déterminer celles qui y sont dans la mesure où on a un chemin augmentant.

## ■ Preuve

```

OCaml
let rec bascule_chemin couplage chemin =
  match chemin with
  | s::t::q -> couplage.(s) <- Some t; couplage.(t) <- Some s; bascule_chemin couplage q
  | [] -> ()
  | [_] -> failwith "Le chemin est forcément de longueur impaire"

```

**Question V.3.16** En déduire une fonction,

`couplage_maximal : (int * int) list -> (int * int) list`

telle que `couplage_maximal aretes`, où `aretes` est un graphe biparti donné sous forme d'une liste d'arêtes, renvoie un couplage maximal sous la forme d'une liste d'arêtes.

Pour cela, on va itérer sur chaque sommet source en cherchant un chemin augmentant depuis celui-ci et en le basculant. Quand on aura traité toutes les sources, on est certain qu'il n'y a plus de chemin augmentant.

### ■ Preuve

OCaml

```
let couplage_maximal l =
  let g, sources = graphe_biparti l in
  let n = Array.length g in
  let couplage = Array.make n None in
  for i = 0 to n-1 do
    if sources.(i) && couplage.(i) = None
    then match chemin_augmentant g sources couplage i with
      | None -> ()
      | Some ch -> bascule_chemin couplage ch
  done;
  couplage_tab_vers_liste sources couplage
```

■



## V.4 Algorithme de Ford-Fulkerson en C

### V.4.i Manipulations de base sur les graphes

On pourra sauter cette partie si on est à l'aise sur ces notions et partir du fichier `tp_couplage_1.c`.  
On va considérer le type suivant pour des graphes avec listes chaînées d'adjacence :

```
struct edge_list {
    int dst;
    struct edge_list *next;
};
typedef struct edge_list edge_list;

struct graph {
    int nvertices;
    bool directed;
    edge_list **edge_lists;
};
typedef struct graph graph;
```

**Question V.4.1** Écrire une fonction de prototype

```
graph *graph_create(int nvertices, bool directed);
```

qui crée un graphe vide à `nvertices` sommets et dont le caractère orienté est donné par le booléen `directed`.

**Question V.4.2** Écrire des fonctions de prototype

```
void edge_list_free(edge_list *l);
void graph_free(graph *g);
```

permettant de libérer la mémoire occupée par un graphe.

**Question V.4.3** Écrire une fonction de prototype

```
bool is_connected(graph *g, int x, int y);
```

qui teste s'il existe une arête  $x \rightarrow y$ .

**Question V.4.4** Écrire une fonction de prototype

```
void add_edge(graph *g, int src, int dst);
```

qui ajoute une arête de `src` à `dst` dans le graphe. On ne fera aucune vérification quant au fait que l'arête soit déjà présente.

**Question V.4.5** Écrire une fonction de prototype

```
void add_edges(graph *g, int *vtx_pairs, int npairs);
```

qui prend en entrée un tableau `t` de `2*npairs` entiers et qui rajoute des arêtes de `t[2i]` à `t[2i+1]`.

**Question V.4.6** Écrire une fonction de prototype

```
void graph_print(graph *g);
```

qui affiche le graphe sous un format lisible. On s'en servira pour faire des vérifications.

**Question V.4.7** Écrire une fonction de prototype

```
⌋ int *dfs_init(graph *g, int src);
```

qui renvoie un tableau `pred` valant `-1` pour tous les sommets autre que `src` et valant `src` pour `src`.

**Question V.4.8** Écrire une fonction de prototype

```
⌋ void dfs(graph *g, int *pred, int x);
```

qui effectue une étape récursive d'un DFS en partant du sommet `x` et en remplissant le tableau de prédécesseur `pred` : `pred[y] = x` si `y` est découvert par le sommet `x`.

**Question V.4.9** Écrire une fonction de prototype

```
⌋ int *dfs_launch(graph *g, int src);
```

qui renvoie le tableau `pred` à l'issue d'un DFS depuis `src`.

**V.4.ii Retour sur les couplages de graphe biparti**

On pourra sauter cette partie si on est à l'aise sur ces notions et partir du fichier `tp_couplage_2.c`.

On va reprendre les notations du TP précédent. Pour indiquer les deux parties des sommets d'un graphe biparti, on va appeler les premiers sommets des sources et d'autres des buts. L'appartenance à un ensemble ou à l'autre étant déterminée par un tableau de booléens `sources`.

```
⌋ graph *g1 = graph_create(5, false);
   int vtx_pairs1[] = { 0, 3, 1, 3, 1, 4, 2, 3, 2, 4 };
   bool sources1[] = { true, true, true, false, false };
   add_edges(g1, vtx_pairs1, 5);

   graph *g2 = graph_create(12, false);
   int vtx_pairs2[] = { 0, 8, 0, 7, 2, 6, 2, 9, 3, 8, 4, 8, 4, 9, 5, 11 };
   bool sources2[] = { true, false, true, true, true, true,
                       false, false, false, false, false, false };
   add_edges(g2, vtx_pairs2, 8);
```

Pour les couplages, on va considérer un tableau d'entiers `matching` où `matching[x] = y` si  $\{x, y\}$  est dans le couplage.

**Question V.4.10** Écrire une fonction de prototype

```
⌋ graph *residual_graph(graph *g, bool *sources, int *matching);
```

qui renvoie le graphe résiduel **orienté** associé au couplage du graphe biparti.

**Question V.4.11** Écrire une fonction de prototype

```
⌋ void swap_along_path(int *matching, int *pred, int x);
```

qui considère un sommet but libre `x` atteint par un DFS depuis un sommet source libre dans le graphe résiduel et bascule les arêtes du chemin correspondant en utilisant le tableau `pred` des prédécesseurs.

**Question V.4.12** Écrire une fonction de prototype

```
⌋ bool augment(graph *g, bool *sources, int *matching, int *pred);
```

qui considère un graphe biparti, un couplage et un tableau `pred` issu d'un DFS depuis une source libre et renvoie `true` après avoir basculé un chemin augmentant s'il en existe un, et `false` sinon.

**Question V.4.13** Écrire une fonction de prototype

```
int *maximal_matching(graph *g, bool *sources);
```

qui renvoie un tableau `matching` décrivant un couplage maximal.

### V.4.iii Ford-Fulkerson

**Note** si vous vous sentez à l'aise, il peut être intéressant de directement essayer d'implémenter Ford-Fulkerson. On fera attention à préserver la complexité, ce qui implique de bien réfléchir à l'organisation de votre code. C'est un exercice délicat mais très instructif. Cette partie vous propose un choix d'implémentation parmi d'autres.

Pour pouvoir travailler sur l'algorithme de Ford-Fulkerson, il va être nécessaire de reporter des changements de flot le long d'un chemin dans le graphe résiduel vers le graphe initial. On va donc avoir besoin de considérer les arêtes dans un chemins et pas seulement les sommets donnés par la relation prédécesseurs **et** on va avoir besoin d'accéder directement à une arête du graphe initial depuis une des deux arêtes créées dans le graphe résiduel.

C'est pour cela qu'on enrichit le type précédent :

```
struct edge_list {
    int src;
    int dst;
    int flow; // le flot sur l'arete
    int capacity; // la capacite de l'arete
    struct edge_list *ref; // NULL ou l'arete associee
    struct edge_list *next;
};
typedef struct edge_list edge_list;

struct graph {
    int s; /* graphe s-->*-->t */
    int t;
    int nvertices;
    edge_list **edge_lists;
};
typedef struct graph graph;
```

**Question V.4.14** Reprendre la fonction `graph_create` pour tenir compte de `s` et `t`.

```
graph *graph_create(int nvertices, int s, int t);
```

Ici, le graphe est forcément orienté.

**Question V.4.15** Reprendre la fonction `add_edge` pour tenir compte de cela, cette fonction affectera un flot nul et une référence `NULL` :

```
edge_list *add_edge(graph *g, int src, int dst, int capacity);
```

**Question V.4.16** Reprendre la fonction `graph_print` pour afficher le flot

```
void graph_print(graph *g);
```

**Question V.4.17** Reprendre les fonctions de parcours pour qu'elles correspondent aux prototypes suivants :

```

    edge_list **dfs_init(graph *g);
    void dfs(graph *g, edge_list **pred, int x);
    edge_list **dfs_launch(graph *g);

```

Tous les DFS démarreront de  $g \rightarrow s$ .

Ici, le tableau des prédécesseurs indique l'arête empruntée pour remonter. Comme la source  $g \rightarrow s$  du DFS aura une valeur NULL dans ce tableau, il est nécessaire de la rajouter en argument pour ne pas faussement la considérer comme étant non visitée.

**Question V.4.18** Écrire une fonction de prototype

```

    graph *residual_graph(graph *g);

```

qui renvoie le graphe résiduel associé au flot présent sur les arêtes de  $g$ .

On fera en sorte que les arêtes du graphe résiduel fassent référence aux arêtes de  $g$  dont elles proviennent. On va utiliser le champ `capacity` pour l'étiquette et le champ `flow` aura la valeur 1 si on est dans l'orientation de  $g$  et -1 si on est dans l'orientation inverse (retour du flot).

**Question V.4.19** Écrire une fonction de prototype

```

    void adjust_flow_along_path(edge_list **pred, int x);

```

qui étant donné un tableau `pred` issu d'un DFS depuis la source du graphe de flot et le sommet `t` destination **visité** va remonter le chemin augmentant pour ajuster le flot dans le graphe initial.

Il faudra deux parcours ici pour obtenir le minimum puis pour ajuster le flot.

**Question V.4.20** Écrire une fonction de prototype

```

    void maximal_flow(graph *g);

```

qui calcule un flot maximal depuis un flot nul.

**Question V.4.21** Testez votre fonction sur l'exemple du cours :

```

int main(void)
{
    graph *g = graph_create(6);
    add_edge(g, 0, 1, 20);
    add_edge(g, 0, 2, 10);
    add_edge(g, 1, 2, 10);
    add_edge(g, 1, 3, 5);
    add_edge(g, 3, 2, 15);
    add_edge(g, 2, 4, 10);
    add_edge(g, 4, 3, 10);
    add_edge(g, 3, 5, 15);
    add_edge(g, 4, 5, 20);
    maximal_flow(g);
    graph_print(g);

    graph_free(g);
}

```

**Question V.4.22** Assurez-vous de n'avoir aucune fuite mémoire.

**Question V.4.23** Appliquez cela au cas d'un graphe biparti pour retrouver la méthode vue précédemment.