

On trouvera sur le site <https://github.com/nathaniel-carre/MP-LLG> un fichier `utilitaire.ml` à télécharger (dans le dossier TP/TP1-Couplages). Pour ouvrir ce fichier depuis un autre fichier `.ml`, on peut écrire, en entête, la commande suivante :

```
#use "Chemin/du/fichier/utilitaire.ml";;
```

en remplaçant bien sûr le chemin du fichier de manière adéquate (attention, il faut penser à transformer les \ des chemins Windows en /, le cas échéant).

1 Préliminaires

1. Lire les descriptions du type de données et des fonctions utilitaires dans le fichier `utilitaire.ml`.

On représente un graphe biparti non orienté non pondéré par un objet de type `graphe` de telle sorte que si $G = (S = X \sqcup Y, A)$ est un graphe non orienté non pondéré représenté par un objet `g` de type `graphe`, alors :

- $n_X = |X|$ est égal à `g.nx`, $n_Y = |Y|$ est égal à `g.ny`, $X = \llbracket 0, n_X - 1 \rrbracket$ et $Y = \llbracket n_X, n_X + n_Y - 1 \rrbracket$;
- pour $s \in S$, `g.adj.(s)` est une liste chaînée contenant les voisins de s .

Un couplage C dans un graphe $G = (S, A)$ est représenté par un tableau d'entiers `c` de taille $|S|$ tel que pour $s \in S$, `c.(s)` est égal à $t \in S$ si $\{s, t\} \in C$, et `c.(s)` est égal à -1 si s est un sommet libre pour C .

Le graphe `g1` créé dans le fichier `utilitaire.ml` est représenté figure 1. On y représente également un couplage C_1 . On pourra utiliser le graphe et le couplage pour tester les fonctions demandées.

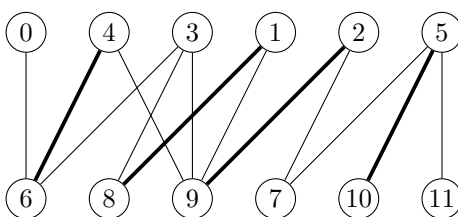


FIGURE 1 – Un graphe biparti G_1 et un couplage C_1 (en gras).

2. Créer un tableau correspondant au couplage C_1 .
3. Écrire une fonction `cardinal_couplage (g: graphe) (c: int array) : int` qui calcule le cardinal d'un couplage C dans un graphe G .

2 Couplage de cardinal maximum

On représente un chemin sous la forme d'une liste chaînée de ses sommets.

4. Écrire une fonction récursive :

```
chemin_alternant (g: graphe) (c: int array) (vus: bool array) (x: int) : int list option
```

qui prend en argument un graphe biparti $G = (X \sqcup Y, A)$, un couplage C de G , un tableau de booléens `vus` et un sommet $x \in X$ et renvoie une option de chemin alternant pour C commençant par x et terminant par un sommet libre de Y . La fonction ne devra pas explorer les sommets déjà vus (dans le tableau `vus`), et marquer comme `vus` les sommets explorés. La fonction renverra `None` s'il n'existe pas de tel chemin et `Some sigma` s'il existe un tel chemin `sigma`.

Indication : les arêtes de Y vers X d'un tel chemin sont directement déterminées par C .

5. Écrire une fonction `augmenter (c: int array) (sigma: int list) : unit` qui prend en argument un couplage C et un chemin σ supposé augmentant pour C et modifie C en $C\Delta\sigma$.

Indication : la liste σ sera supposée de longueur paire.

6. En déduire une fonction `couplage_maximum (g: graphe) : int array` qui calcule et renvoie un couplage de cardinal maximum de G .
7. Vérifier qu'un couplage maximum pour le graphe `g2` est de cardinal 73 et qu'un couplage maximum pour le graphe `g3` est de cardinal 9060.

3 Algorithme de Hopcroft-Karp

L'algorithme usuel de recherche de couplage maximum dans un graphe biparti effectue autant de parcours de graphe que le cardinal du couplage, ce qui résulte en une complexité en $\mathcal{O}(|S||E|)$ dans le cas général. L'algorithme de Hopcroft-Karp améliore cette complexité en trouvant plusieurs chemins augmentants d'un coup pour augmenter le couplage en cours de construction. Il se résume de cette manière.

Entrée : Graphe $G = (X \sqcup Y, A)$ biparti non orienté

Début algorithme

$C \leftarrow \emptyset$

Tant que il existe un chemin augmentant pour C **Faire**

 Trouver $E = \{\sigma_1, \sigma_2, \dots, \sigma_k\}$ un ensemble maximal de plus courts chemins augmentants pour C , disjoints deux à deux.

 Augmenter C avec les chemins de E .

Renvoyer C

L'ensemble E est maximal au sens où tout autre chemin augmentant aurait au moins un sommet en commun avec l'un des σ_i . La difficulté de l'algorithme consiste à déterminer un tel ensemble E en complexité linéaire en la taille du graphe. L'idée pour ce faire est la suivante :

- on effectue un parcours en **largeur** alternant depuis les sommets libres de X , pour déterminer les distances des sommets de G à un sommet libre de X dans des chemins alternants ;
- dans l'ordre croissant des distances précédentes, on effectue des parcours en **profondeur** depuis les sommets libres de Y , pour trouver des plus courts chemins alternants jusqu'aux sommets libres de X .

8. Écrire une fonction `bfs_alternant (g: graphe) (c: int array) : int array * int array` qui prend en argument un graphe biparti $G = (S = X \sqcup Y, A)$ et un couplage C et renvoie un couple (`ordre_bfs`, `dist`) qui sont deux tableaux d'entiers de taille $|S|$ tels que :

- pour $s \in S$, `dist[s]` contient la longueur minimale d'un chemin alternant d'un sommet libre de X à s . En particulier, si $x \in X$ est un sommet libre, alors `dist[x]` doit valoir 0. Par convention, s'il n'existe pas de tel chemin, on posera `dist[s] = -1` ;
- `ordre_bfs` contient les sommets de S accessible par un chemin alternant depuis un sommet libre de X , par ordre croissant de `dist`. S'il existe des sommets de S non accessibles par un tel chemin, on complètera le tableau `ordre_bfs` par des valeurs -1 .

Indication : on pourra utiliser le tableau `ordre_bfs` en guise de file, en gardant en mémoire l'indice du prochain élément à sortir de la file et l'indice de la prochaine case libre du tableau.

9. Écrire une fonction

`dfs_alternant (g: graphe) (c: int array) (dist: int array)`

`(vus: bool array) (y: int) : int list option`

qui prend en argument un graphe biparti $G = (S = X \sqcup Y, A)$, un couplage C , un tableau `dist` tel que renvoyé par la fonction précédente, un tableau de booléens `vus` et un sommet $y \in Y$ et renvoie une option de **plus court** chemin alternant pour C commençant par y et terminant par un sommet libre de X . La fonction ne devra pas explorer les sommets déjà vus (dans le tableau `vus`), et marquer comme `vus` les sommets explorés. La fonction renverra `None` s'il n'existe pas de tel chemin et `Some sigma` s'il existe un tel chemin `sigma`.

Indication : pour garantir qu'il s'agit d'un plus court chemin, on n'explorera que les voisins dont la distance vaut 1 de moins que y .

10. En déduire une fonction `hopcroft_karp` (`g`: `graphe`) qui calcule un couplage de cardinal maximum dans un graphe biparti selon l'algorithme de Hopcroft-Karp.
11. Vérifier la correction de l'algorithme sur les graphes `g1`, `g2` et `g3`.
12. Comparer les performances temporelles entre les fonctions `couplage_maximum` et `hopcroft_karp` sur les graphes `g2` et `g3`.
13. [À faire après le TP] Montrer qu'après chaque passage dans la boucle **Tant que** de l'algorithme de Hopcroft-Karp, la longueur minimale d'un chemin augmentant pour C augmente d'au moins 1.
14. [À faire après le TP] Soit C^* un couplage de cardinal maximum. Montrer qu'après $\sqrt{|S|}$ passages dans la boucle **Tant que**, $|C^*| - |C|$ vaut au plus $\sqrt{|S|}$.
15. [À faire après le TP] En déduire la complexité temporelle de l'algorithme de Hopcroft-Karp.

4 Couplage de cardinal maximum et de poids minimum

Adapté d'un sujet de J.-B. Bianquis.

Si on considère $G = (S = X \sqcup Y, A, f)$ un graphe non orienté pondéré biparti, on peut étendre la recherche de couplage maximum en cherchant un couplage de poids minimum parmi ceux de cardinal maximum, le poids $f(C)$ d'un couplage C étant la somme des poids de ses arêtes.

Si C est un couplage de G et σ est un chemin augmentant pour C , alors on définit le **coût** de σ par :

$$f_C(\sigma) = \sum_{a \in \sigma \setminus C} f(a) - \sum_{a \in \sigma \cap C} f(a)$$

On propose d'implémenter l'algorithme suivant :

Entrée : Graphe $G = (X \sqcup Y, A)$ biparti non orienté

Début algorithme

$C \leftarrow \emptyset$

Tant que il existe un chemin augmentant pour C **Faire**

 Trouver σ un chemin augmentant pour C de coût minimal.

$C \leftarrow C \Delta \sigma$.

Renvoyer C

16. [À faire après le TP] Montrer que si σ est un chemin augmentant pour C , alors $f(C \Delta \sigma) = f(C) + f_C(\sigma)$.
17. [À faire après le TP] Montrer que l'algorithme précédent renvoie un couplage de poids minimum parmi les couplages de cardinal maximum.

La recherche d'un chemin augmentant de coût minimal se ramène à une recherche de plus court chemin dans un graphe orienté particulier. Si C est un couplage de G , on pose $G_C = (S', A', g)$ le graphe orienté pondéré défini par :

– $S' = S \cup \{s, t\}$;

– l'ensemble A' contient les arêtes suivantes :

- * les (s, x) pour $x \in X$ sommet libre pour C , avec un poids $g(s, x) = 0$;
- * les (y, t) pour $y \in Y$ sommet libre pour C , avec un poids $g(y, t) = 0$;
- * les (x, y) avec $x \in X$, $y \in Y$ et $\{x, y\} \in A \setminus C$, avec un poids $g(x, y) = f(x, y)$;
- * les (y, x) avec $x \in X$, $y \in Y$ et $\{x, y\} \in C$, avec un poids $g(y, x) = -f(x, y)$.

18. [À faire après le TP] Montrer que (s, s_1, \dots, s_k, t) est un chemin de poids minimal dans G_C si et seulement si (s_1, \dots, s_k) est un chemin augmentant pour C dans G de coût minimal.

On représente un graphe non orienté biparti pondéré comme la donnée d'un objet de type `graphe` et d'une matrice d'adjacence pondéré de type `float array array` (les poids sont des flottants). Un graphe orienté pondéré sera représenté par une matrice de type `float array array`. Il est à noter que `infinity` est un flottant en OCaml qui se comporte comme $+\infty$ (plus grand que tout flottant, absorbant pour $+$). On supposera que s'il n'existe pas d'arête entre s et t dans le graphe, alors la case d'indices (s, t) vaut `infinity` dans la matrice.

19. Écrire une fonction

```
poids_couplage (g: graphe) (m: float array array) (c: int array) : float
```

qui détermine le poids d'un couplage C dans un graphe G .

20. Écrire une fonction `floyd_warshall (m: float array array) : int array array` qui prend en argument une matrice et un entier correspondant à la description d'un graphe orienté pondéré $G = (S, A, f)$ et renvoie une matrice `pred` de taille $n \times n$ telle que pour $s, t \in S^2$, `pred.(s).(t)` contient le prédécesseur de t dans un plus court chemin de s à t s'il en existe un, et -1 sinon. Par convention, on posera `pred.(s).(t)` égal à -1 . La fonction `floyd_warshall` pourra modifier la matrice `m` directement si nécessaire.
21. Écrire une fonction

```
plus_court_chemin (m: float array array) (s: int) (t: int) : int list option
```

qui prend en argument une matrice et un entier correspondant à la description d'un graphe orienté pondéré $G = (S, A, f)$ et deux sommets $s, t \in S^2$ et renvoie une option de liste contenant les sommets intermédiaires d'un plus court chemin de s à t (c'est-à-dire le chemin de s à t sans ses extrémités s et t). La fonction renverra `None` s'il n'existe pas de tel chemin.

22. Écrire une fonction

```
construire_GC (g: graphe) (m: float array array) (c: int array) : float array array
```

qui construit une matrice d'adjacence pondéré correspondant au graphe G_C . Le choix de la numérotation des sommets s et t ajoutés est laissé libre.

23. En déduire une fonction

```
couplage_maximum_poids_minimum (g: graphe) (m: float array array) : int array
```

qui renvoie un couplage de poids minimum parmi ceux de cardinal maximum dans le graphe G .

24. Vérifier que le poids du couplage renvoyé pour G_1 et M_1 est 77,101 et que celui pour G_2 et M_2 est 2747,422.
25. Déterminer la complexité temporelle de la fonction précédente.