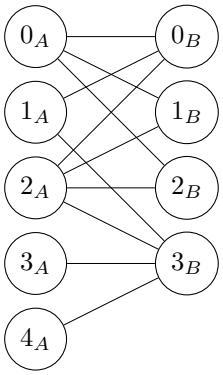
TP : Couplage de cardinal maximum dans un graphe biparti

1 Représentation des graphes bipartis

On rappelle qu'un graphe *biparti* est un graphe non orienté $G = (V, E)$ tel que l'ensemble V peut se partitionner en deux sous-ensembles A et B, tels que toute arête du graphe relie un sommet de A et un sommet de B. Par exemple, le graphe biparti G_0 ci-dessous possède 9 sommets, partitionnés en 5+4 sommets. On notera dans la suite $n_A = |A|$ et $n_B = |B|$. Dans la suite, bien que ce soit légèrement sous-optimal du point de vue de la complexité, on choisit de représenter un tel graphe via sa matrice d'adjacence. Comme le graphe est biparti, une matrice de taille $n_A \times n_B$ suffit, le coefficient en case (i_A, j_B) indiquant si le sommet $i_A \in A$ est relié au sommet $j_B \in B$, voir figure 1.



$$\begin{pmatrix} V & V & V & F \\ V & F & F & V \\ V & V & V & V \\ F & F & F & V \\ F & F & F & V \end{pmatrix}$$

FIGURE 1: Le graphe G_0 et sa représentation matricielle : on a indiqué V et F (pour vrai et faux) dans chaque case de la matrice.

Rappels. On rappelle les définitions vues en cours :

- un *couplage* du graphe est un sous-ensemble d'arêtes d'extrémités deux à deux disjointes. Par exemple $\emptyset$ ou $\{\{0_A, 1_B\}, \{1_A, 3_B\}\}$ sont des couplages de G_0 ;
- un *couplage maximal* est un couplage qui n'est inclus strictement dans aucun autre. Par exemple, le couplage $\{\{0_A, 0_B\}, \{1_A, 3_B\}, \{2_A, 2_B\}\}$ est maximal ;
- un *couplage de cardinal maximum* est un couplage qui a le plus grand cardinal. Évidemment un couplage de cardinal maximum est un couplage maximal, mais la réciproque est fausse. Le couplage précédent n'est pas de cardinal maximum, car il en existe un de cardinal 4 : $\{\{0_A, 1_B\}, \{1_A, 0_B\}, \{2_A, 2_B\}, \{3_A, 3_B\}\}$, qui est clairement de cardinal maximum puisque tous les sommets de B sont couplés. Il n'y a pas unicité du couplage de cardinal maximum (remplacer la dernière arête par $\{4_A, 3_B\}$ par exemple).

Représentation. En Caml, un graphe biparti sera représenté comme une matrice (tableau de tableaux) de booléens, de type `bool array array`. Un couplage sera représenté par un tableau de taille n_A, indiquant pour chaque sommet de A couplé le numéro du sommet de B correspondant, et -1 si le sommet n'est pas couplé. Voici par exemple la représentation du graphe G_0 et du couplage $\{\{0_A, 0_B\}, \{1_A, 3_B\}, \{2_A, 2_B\}\}$.

```
let g0 = [|[|true; true; true; false|]; [|true; false; false; true|];  [|true; true; true; true|];
[|false; false; false; true|]; [|false; false; false; true|]|] ;;
let c = [|0; 3; 2; -1; -1|] ;;
```

Fonctions utiles. On rappelle les manipulation suivantes sur les tableaux :

— `Array.length t` donne la longueur d'un tableau ;

— `t.(i)` accès au i-ème élément d'un tableau `t` ($0 \leq$ `i` $<$ `Array.length t`).

— `t.(i) <- x` modification du i-ème élément d'un tableau `t` ($0 \leq$ `i` $<$ `Array.length t`).

— `Array.make n x` crée un tableau de taille n rempli de x ;

— `Array.length m`, `Array.length m.(0)` nombre de lignes et colonnes d'une matrice (tableau de tableaux) ;

— `m.(i).(j)` : accès à l'élément situé sur la i-ème ligne et la j-ème colonne. De même pour la modification ;

— `Array.make_matrix n m x` crée une matrice $n \times m$ remplie de x.

2 Quelques vérifications

Question 1. Écrire une fonction `verifie g c` prenant en entrée un graphe biparti (représenté par une matrice de booléens $n_A \times n_B$) et un tableau d'entiers de taille n_A et renvoyant un booléen indiquant si `c` est un couplage de `g`. On supposera sans le vérifier que les entrées de `c` sont dans $[\![-1, n_B - 1]\!]$. On essaiera d'avoir une complexité $O(n_A + n_B)$.

```
# verifie g0 [|-1; -1; -1; -1; -1|] ;;
- : bool = true
# verifie g0 [|0; 3; 2; -1; -1|] ;;
- : bool = true
# verifie g0 [|0; 3; 2; -1; 1|] ;; (* l'arete {4A, 1B} n'existe pas *)
- : bool = false
# verifie g0 [|0; 3; 2; -1; 3|] ;; (* deux sommets couplés à 3B *)
- : bool = false
```

3 Algorithme glouton pour un couplage maximal

On développe dans cette partie un premier algorithme permettant de calculer un couplage maximal. L'idée est simple : parcourir toutes les arêtes du graphe, et rajouter une arête au couplage dès que c'est possible.

3.1 Première version

Question 2. Écrire la fonction `couplage_glouton_V1 g` renvoyant un couplage de `g` en suivant cette méthode. On parcourera la matrice de haut en bas puis de gauche à droite. Outre le couplage en construction (tableau de taille n_A), on utilisera un tableau de booléens de taille n_B indiquant pour chaque sommet de B s'il est déja couplé ou non.

```
# couplage_glouton_V1 g0 ;;
- : int array = [|0; 3; 1; -1; -1|]
```

Question 3. Quelle est la complexité de l'algorithme obtenu en fonction des cardinaux n_A et n_B de A et B ?

3.2 Deuxième version avec heuristique

Comme on le voit, le couplage renvoyé n'est pas de cardinal maximum. La version précédente peut être un peu améliorée dans la pratique, en suivant l'idée que pour optimiser le cardinal d'un couplage, on a intérêt à considérer en premier les arêtes du graphe dont les sommets aux extrémités ont des petits degrés, car sinon ils seraient plus dur à coupler ensuite. En résumé, on va appliquer la stratégie suivante :

— calculer les degrés des sommets du graphe ;

— extraire la liste des arêtes du graphe ;

— trier la liste précédente par valeurs croissantes de la fonction $\{v_A, w_B\} \mapsto \deg v_A + \deg w_B$;

— considérer les arêtes une à une comme dans la première version de l'algorithme.

Question 4. Écrire une fonction `couplage_glouton_V2 g` suivant cette idée. On pourra découper le travail comme suit :

— Écrire une fonction `degAB_aretes g0` renvoyant le tableau des degrés des sommets de A, de même pour B, et la liste des arêtes sous la forme de couples (v_A, w_B) (lire le rappel sur les listes situé après si besoin).

```
# degAB_aretes g0 ;;
- : int array * int array * (int * int) list =
([|3; 2; 4; 1; 1|], [|3; 2; 2; 4|], [(4, 3); (3, 3); (2, 3); (2, 2); (2, 1); (2, 0); (1, 3);
                (1, 0); (0, 2); (0, 1); (0, 0)])
```

— Utiliser la fonction `List.sort` pour trier la liste d'arêtes (il faudra déclarer la fonction de tri qui va bien, voir le rappel).

— Parcourir la liste triée en suivant la même idée que l'algorithme de la version 1.

```
# couplage_glouton_V2 g0 ;;
- : int array = [|2; 0; 1; -1; 3|]
```

Rappel sur les listes. On rappelle qu'une liste est *immuable* en Ocaml.

— pour parcourir une liste, on utilise en général une fonction récursive. Le motif `[]` filtre la liste vide, le motif `x::q` filtre une liste non vide, et permet de récupérer la tête de la liste (l'élément `x`), et la queue (le reste des éléments, sous forme de liste)

— pour construire une liste de façon itérative, il est nécessaire d'utiliser une référence : `let q=ref [] in ...` puis rajout des éléments via `q:= x :: !q`.

— la fonction `List.sort`, de type : `('a -> 'a -> int) -> 'a list -> 'a list` prend en entrée une fonction `f` de tri curryfiée à deux arguments, une liste, et renvoie une nouvelle liste contenant les mêmes éléments, de sorte que si `x` est placé avant `y`, alors `f x y` ≤ 0. Par exemple :

```
# let f x y = x - y in List.sort f [5; 7; 1; 9; 3; 4] ;;
- : int list = [1; 3; 4; 5; 7; 9]
```

Question 5. Quelle est la complexité de l'algorithme obtenu en fonction des cardinaux n_A et n_B de A et B ?

4 Algorithme hongrois pour un couplage de cardinal maximum

L'algorithme glouton précédent, même la deuxième version, ne donne pas nécessairement un couplage de cardinal maximum. Par exemple, dans le graphe suivant

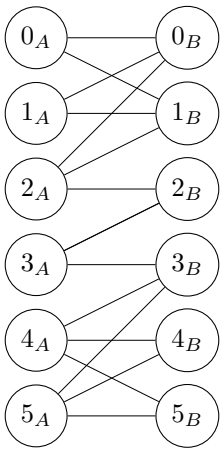


FIGURE 2: Le graphe G_1

l'algorithme `couplage_glouton_V2` renvoie un couplage de cardinal 5 :

```
# couplage_glouton_V2 g1 ;;
- : int array = [|0; 1; -1; 2; 4; 5|]
```

alors que le couplage $\{\{i_A, i_B\} \mid 0 \leq i \leq 5\}$ est de cardinal 6. Dans cette section, on implémente l'algorithme hongrois, permettant de calculer un couplage de cardinal maximum en détectant des *chemins augmentants*.

Rappel. Un chemin augmentant dans un graphe biparti $G = (A \cup B, E)$ pour un couplage C est un chemin dans G :

— démarrant en un sommet **non couplé** de A ;

— alternant ensuite sommet de B puis un sommet de A ;

— et terminant en un sommet **non couplé** de B.

— **Contrainte : les arêtes suivies de A vers B sont toutes des arêtes qui n'appartiennent pas au couplage, les arêtes suivies de B vers A sont toutes des arêtes du couplage.**

On a vu en cours qu'un couplage qui n'est pas de cardinal maximum admet nécessairement un chemin augmentant, et qu'on peut utiliser un tel chemin pour augmenter de 1 le cardinal d'un couplage. La figure suivante montre un couplage de G_0 qui n'est pas de cardinal maximum, un chemin augmentant, et le couplage obtenu après augmentation.

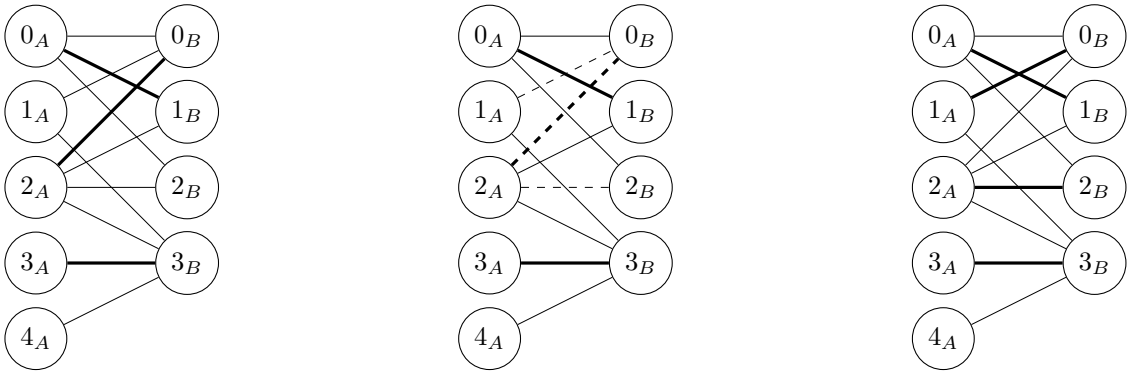


FIGURE 3: Le graphe G_0 et le couplage $\{\{0_A, 1_B\}, \{2_A, 0_B\}, \{3_A, 3_B\}\}$. Le chemin $1_A, 0_B, 2_A, 2_B$ (au centre en pointillés) est augmentant. L'utiliser donne le couplage de droite.

Pour calculer un chemin augmentant s'il en existe un, il suffit de parcourir en profondeur le graphe, avec quelques modifications. On rappelle l'algorithme de parcours en profondeur en pseudo-code :

Algorithme 1 : Parcours en profondeur du graphe

```
Entrée : Un graphe G donné par liste d'adjacence
deja_vu[v] ← Faux  pour tout sommet v;
Fonction pp(u) :
    deja_vu[u] ← Vrai;
    pour tout voisin v de u faire
        si deja_vu[v] = Faux alors
            pp(v)
pour tout sommet u du graphe faire
    si deja_vu[u] = Faux alors
        pp(u)
```

Les modifications à apporter à ce parcours sont :

— il faut dissocier la fonction *pp* suivant qu'on l'appelle sur un sommet de A ou de B, on écrira donc *deux* fonctions distinctes, mutuellement récursives (`let rec pa x = ... and pb x = ...`). Dans un appel sur un sommet i_A de A, il faudra exclure le sommet de B éventuellement couplé avec i_A. Inversement, dans un appel sur un sommet i_B de B, on ne considérera comme voisin que le sommet éventuellement couplé avec i_B.

— dans la boucle principale, on ne parcourera que les sommets de A, et on ne fera appel à la fonction *pa* que sur un sommet non couplé.

Question 6. Écrire une fonction `parcours_prof g c r` prenant en entrée :

— le graphe G ;

— le couplage C ;

— le *couplage réciproque* R : c'est un tableau de taille n_B qui indique pour chaque sommet de B le numéro du sommet de A avec lequel il est couplé, ou -1 s'il n'est pas couplé ;

et explorant le graphe G en suivant ce principe. Votre fonction renverra deux tableaux de prédecesseurs (un pour A, un pour B).

Voici le résultat sur le graphe G_0 et le couplage $C = \{\{\{0_A, 1_B\}, \{2_A, 0_B\}, \{3_A, 3_B\}\}$, sur la figure les prédecesseurs des sommets explorés sont inscrits à côté du nœud.

```
# marquage g0 [|1; -1; 0; 3; -1|] [|2; 0; -1; 3|] ;;
- : int array * int array = ([|1; -1; 0; 3; -1|], [|1; 2; 0; 2|])
```

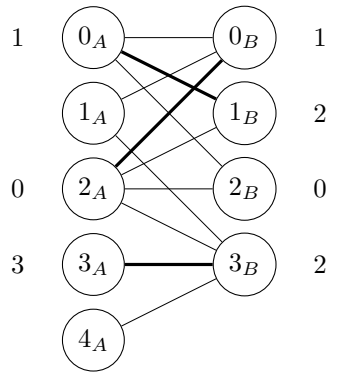


FIGURE 4: Parcours en profondeur sur le graphe G via le couplage C : les prédécesseurs des sommets de A sont indiqués à gauche, ce sont les numéros des sommets de B, et réciproquement de l'autre côté.

Pour voir s'il existe un chemin augmentant, il suffit de vérifier s'il existe dans B un sommet non couplé qui a été atteint par le parcours. Dans l'exemple précédent, le sommet 2_B (le seul non couplé) a été atteint (on le voit car il possède un prédecesseur dans le parcours).

Question 7. Écrire une fonction `non_couple_B r precB`, prenant en entrée un couplage réciproque et le tableau des prédecesseurs de B dans le parcours précédent, et renvoyant l'indice d'un sommet non couplé atteint par le parcours s'il en existe un, ou -1 sinon.

```
# non_couple_B [|2; 0; -1; 3|] [|1; 2; 0; 2|] ;;
- : int = 2
```

Pour obtenir effectivement un chemin augmentant, il suffit de remonter depuis ce sommet jusqu'à un sommet non couplé de A (prédecesseur non défini), en suivant les tableaux de prédecesseurs : on obtient le chemin augmentant à l'envers. Dans l'exemple de la figure 4, le chemin augmentant obtenu est (toujours à l'envers) : $2_B \to 0_A \to 1_B \to 2_A \to 0_B \to 1_A$. Il suffit alors de modifier le couplage pour obtenir un couplage où un sommet de plus est couplé : coupler 2_B et 0_A, 1_B et 2_A et 0_B et 1_A.

Question 8. Écrire enfin la fonction `couplage_cardinal_max g` permettant le calcul d'un couplage de cardinal maximum. On pourra partir du couplage $\emptyset$ (représenté par un tableau rempli de -1, de même que le couplage réciproque R), et appliquer des parcours en profondeur tant que l'on peut faire augmenter la taille du couplage. Après un tel parcours, mettre à jour le couplage et le couplage réciproque.

```
# couplage_cardinal_max g0 ;;
- : int array = [|2; 0; 1; 3; -1|]
# couplage_cardinal_max g1 ;;
- : int array = [|1; 0; 2; 3; 5; 4|]
```

Question 9. Donner dans ce cas la complexité en fonction des cardinaux n_A et n_B de A et B.

Remarque : on peut améliorer un petit peu l'algorithme précédent. Dans le parcours en profondeur, on peut s'arrêter dès qu'un sommet de B non couplé est atteint. Vous pouvez réécrire l'algorithme en suivant cette idée.