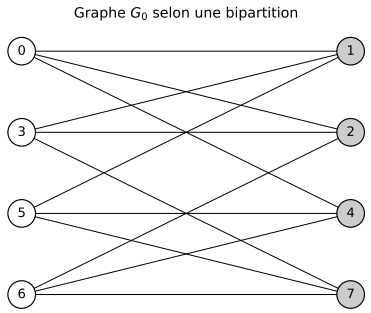
Graphes bipartis et couplages

Contexte

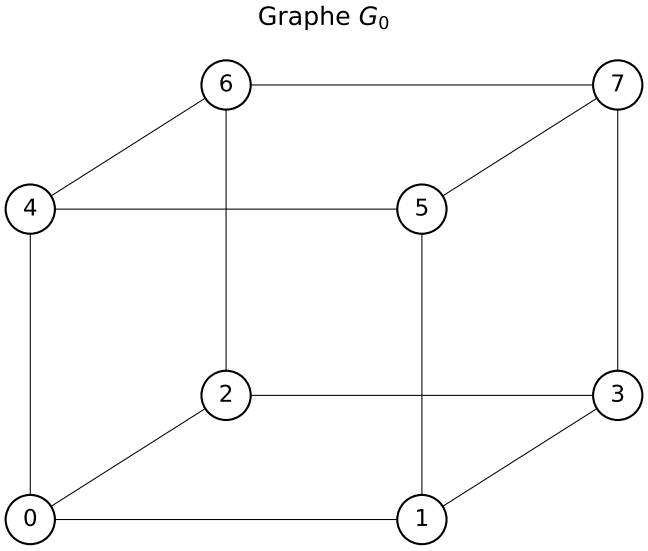


Ce problème est constitué de deux exercices.
Le premier étudie les graphes bipartis et propose deux méthodes pour déterminer une bipartition, si elle existe: une approche naïve par test de toutes les partitions possibles, et une approche par parcours en largeur.
Le second étudie les couplages dans les graphes bipartis, et propose deux méthodes pour construire un couplage maximum: une approche gloutonne non optimale, et une approche par chaînes augmentables (parcours en profondeur).

1 Exercice : Graphe biparti

Dans tout cet exercice on considère des graphes non orientés dont l'ensemble des sommets est de la forme $S = [\![0, n-1]\!]$.
Un tel graphe sera représenté par sa **liste d'adjacence**.
Ainsi le graphe G_0 ci-dessous est représenté par la liste d'adjacence suivante

$$L_0 = [[1, 2, 4], [0, 3, 5], [0, 3, 6], [1, 2, 7], [0, 5, 6], [1, 4, 7], [2, 4, 7], [3, 5, 6]].$$



1.1. Tracer sur votre copie les graphes G_1 et G_2 correspondant respectivement aux listes d'adjacence suivantes

$$L_1 = [[1, 3], [0, 2], [1, 3, 5], [0, 2, 4], [3, 5], [2, 4]]$$
$$L_2 = [[1, 2, 3], [0, 2], [0, 1, 3], [0, 2]].$$

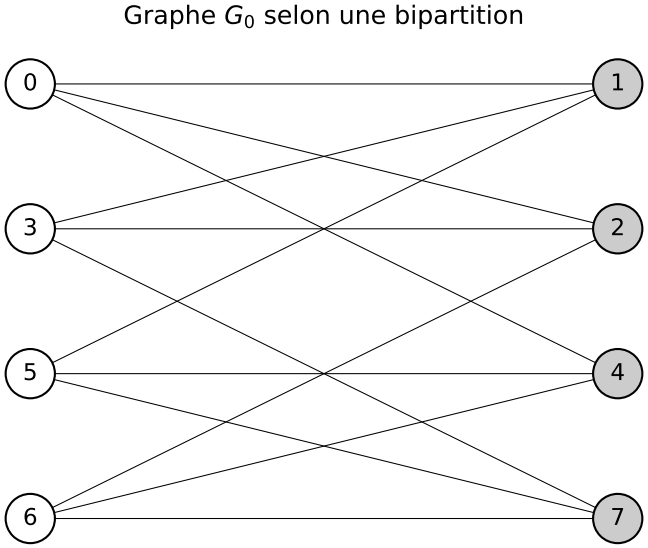
On dit qu'un graphe $G = (S, A)$ est **biparti** s'il existe une partition (ou plus précisément un recouvrement disjoint) de son ensemble de sommets en deux parties S_0 et S_1 telles que toute arête de G relie un sommet de S_0 à un sommet de S_1, c'est-à-dire telle que

$$S = S_0 \sqcup S_1 \quad \text{avec} \quad S_0 \cap S_1 = \emptyset$$
$$\forall \{u, v\} \in A, (u \in S_0 \text{ et } v \in S_1) \text{ ou } (u \in S_1 \text{ et } v \in S_0)$$

Le couple (S_0, S_1) est appelé une **bipartition** du graphe G. Elle sera représentée en python par une liste B de n éléments à valeurs dans $\{0, 1\}$ telle que $S_0 = \{i \in [\![0, n-1]\!] \mid B[i] = 0\}$ et $S_1 = \{i \in [\![0, n-1]\!] \mid B[i] = 1\}$.
Par exemple le graphe associé à la liste d'adjacence L_0 ci-dessus est biparti, avec

$$S_0 = \{0, 3, 5, 6\} \quad \text{et } S_1 = \{1, 2, 4, 7\},$$

toute arête relie bien un sommet de S_0 à un sommet de S_1 .



Cet exemple de bipartition est représentée en python par la liste

$$B = [0, 1, 1, 0, 1, 0, 0, 1].$$

1.2. Les graphes G_1 et G_2 sont-ils bipartis ? Justifier la réponse.

On cherche dans la suite de cet exercice à déterminer si un graphe donné est biparti ou non, et le cas échéant à trouver une partition de son ensemble de sommets en deux parties S_0 et S_1 vérifiant la condition de bipartition.

Une première approche est de tester toutes les partitions possibles de l'ensemble des sommets en deux parties S_0 et S_1 et de vérifier pour chacune d'elles si elle vérifie la condition de bipartition.

1.3. Écrire une fonction python `est_bipartition(L,B)` qui prend en arguments une liste d'adjacence L représentant un graphe G et une liste B de n éléments à valeurs dans $\{0, 1\}$ représentant une partition de l'ensemble des sommets de G en deux parties S_0 et S_1, et qui retourne `True` si la partition représentée par B est une bipartition du graphe représenté par L, et `False` sinon.

L'ensemble des partitions étant en bijection avec l'ensemble $\{0, 1\}^{[\![0,n-1]\!]}$ des listes de n éléments à valeurs dans $\{0, 1\}$, on peut, pour générer toutes les partitions possibles, générer toutes les écritures binaires des entiers de 0 à $2^n - 1$.

1.4. Écrire une fonction python `ecriture_binaire(k,n)` qui prend en argument un entier k entre 0 et $2^n - 1$ et un entier n, et qui retourne la liste de n éléments à valeurs dans $\{0, 1\}$ correspondant à l'écriture binaire de k sur n bits, en commençant par le bit de poids faible (correspondant à 2^0) et en terminant par le bit de poids fort (correspondant à 2^{n-1}).

Ainsi à l'entier $k \in [\![0, 2^n - 1]\!]$ on associe la liste $[b_0, b_1, \ldots, b_{n-1}]$ telle que

$$k = \sum_{i=0}^{n-1} b_i \times 2^i.$$

Par exemple comme

$$47 = 1 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 1 \times 2^5 + 0 \times 2^6 + 0 \times 2^7$$

et que

$$47 \leq 256 - 1 = 2^8 - 1$$

on a que `ecriture_binaire(47,8)` doit retourner la liste $[1, 1, 1, 1, 0, 1, 0, 0]$.

1.5. Démontrer la correction et la terminaison de la fonction `ecriture_binaire(k,n)`.

1.6. En déduire une fonction python `est_biparti_naif(L)` qui prend en argument une liste d'adjacence L représentant un graphe G et qui retourne le tuple `(True, B)` si le graphe G est biparti et que B est une liste de n éléments à valeurs dans $\{0, 1\}$ représentant une bipartition du graphe G, et qui retourne le tuple `(False, None)` sinon.

1.7. Quelle est la complexité de la fonction `est_biparti_naif(L)` en fonction du nombre de sommets $n = |S| = \texttt{len}(L)$ et de $p = |A|$ le nombre d'arêtes du graphe représenté par L ?

Un autre algorithme pour construire une bipartition d'un graphe G est d'utiliser un parcours en largeur du graphe. Plus précisément:

- On initialise une liste B de n éléments à `None`.
- Tant qu'il reste des sommets non visités (c'est à dire de valeur `None` dans B):
  ▷ On choisit un tel sommet s et on lui attribue la valeur 0.
  ▷ On effectue un parcours en largeur complet à partir de s : lorsqu'on visite un sommet u, pour chaque voisin v de u :
  ∗ si v n'a pas encore reçu de valeur, on lui attribue la valeur $1 - B[u]$;
  ∗ sinon, on vérifie que $B[v] \neq B[u]$ et l'on échoue dans le cas contraire.
- Si aucun échec ne s'est produit, on retourne la bipartition B. Sinon le graphe n'est pas biparti.

1.8. Compléter la fonction `est_biparti_bfs(L)` qui prend en argument une liste d'adjacence L représentant un graphe G et qui retourne le tuple `(True, B)` si le graphe G est biparti et que B est une liste de n éléments à valeurs dans $\{0, 1\}$ représentant une bipartition du graphe G, et qui retourne le tuple `(False, None)` sinon.

```python
from collections import deque

def est_biparti_bfs(L):
    n = len(L)
    B = [None for _ in range(n)]
    # On parcourt chaque composante connexe
    for s in range(n):
        if ...:                          # à compléter
            B[s] = 0
            file = deque([s])
            # On effectue un parcours en largeur à partir de s
            while file:
                u = ...                  # à compléter
                for v in L[u]:
                    if ...:              # à compléter
                        ...              # à compléter
                        ...              # à compléter
                    else:
                        if ...:          # à compléter
                            return (False, None)
    return (True, B)
```

1.9. Quelle est la complexité de la fonction `est_biparti_bfs(L)` en fonction du nombre de sommets $n = |S| = \texttt{len}(L)$ et de $p = |A|$ le nombre d'arêtes du graphe représenté par L ?

1.10. Commenter.

2 Exercice: Couplages dans un graphe biparti

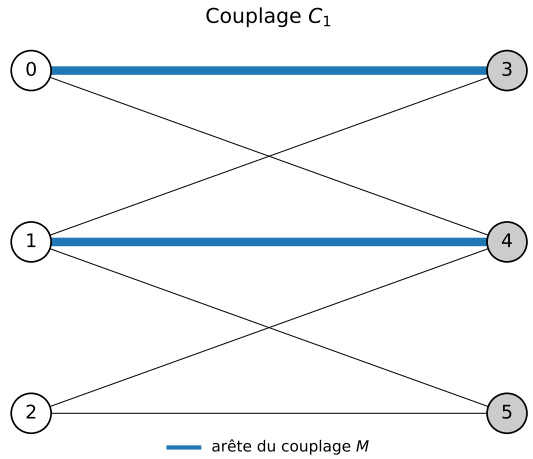
Dans toute cette section, $G = (S, A)$ désigne un graphe non orienté biparti, dont on note (S_0, S_1) une bipartition. Le graphe est représenté en python par sa liste d'adjacence L et sa bipartition par une liste B.
On appelle **couplage** de G toute partie $M \subseteq A$ telle que deux arêtes distinctes de M n'ont aucun sommet en commun.
Un sommet $s \in S$ est dit **saturé** par M s'il est extrémité d'une arête de M, et **libre** sinon.
Un couplage sera représenté en python par un dictionnaire C associant à chaque sommet saturé u l'unique sommet v tel que l'arête $\{u, v\}$ appartient au couplage. Ainsi $C[u] = v$ si et seulement si $C[v] = u$, et un sommet u est libre si et seulement si u n'est pas une clé de C.
De plus un couplage M est dit **maximum** si pour tout couplage $\widetilde{M}$ de G on a $|\widetilde{M}| \leq |M|$.
Par exemple pour le graphe G_3 de liste d'adjacence

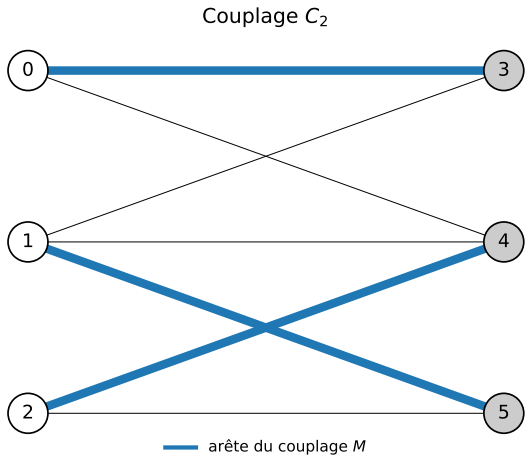
$$L_3 = [[3, 4], [3, 4, 5], [4, 5], [0, 1], [0, 1, 2], [1, 2]]$$

et de bipartition

$$B_3 = [0, 0, 0, 1, 1, 1]$$

on a plusieurs couplages possibles:





Couplage correspondant au dictionnaire

$$C_1 = \{0:3, 3:0, 1:4, 4:1\}.$$

Couplage correspondant au dictionnaire

$$C_2 = \{0:3, 3:0, 1:5, 5:1, 2:4, 4:2\}.$$

2.11. Montrer que sur l'exemple précédent le couplage C_1 n'est pas maximum et que le couplage C_2 est maximum.

2.12. Écrire une fonction python `est_couplage(L, C)` qui prend en arguments une liste d'adjacence L et un dictionnaire C, et qui retourne `True` si C représente un couplage du graphe représenté par L, et `False` sinon.

On cherche dans la suite de cet exercice à construire un couplage maximum de G.

Un premier algorithme glouton pour construire un couplage de G est le suivant:

- Tant qu'il reste des sommets libres dans S_0 :
  - ▷ On choisit le premier (pour l'ordre usuel de $[\![0, n-1]\!]$) sommet libre $u \in S_0$ et on le marque comme saturé.
  - ▷ Si u possède au moins un voisin libre dans S_1 on prend v ce sommet, on marque v comme saturé et on ajoute l'arête $\{u, v\}$ au couplage.

2.13. Écrire une fonction python `couplage_glouton(L, B)` qui prend en arguments une liste d'adjacence L d'un graphe biparti et sa bipartition B, et qui retourne un couplage de G obtenu par ce procédé glouton.

2.14. Quelle est la complexité de cette fonction en fonction de n et de p ?

2.15. Quel est le résultat de cette fonction appliquée au graphe G_4 représenté par la liste d'adjacence

$$L_4 = [[3,4],[3],[5],[0,1],[0],[2]]$$

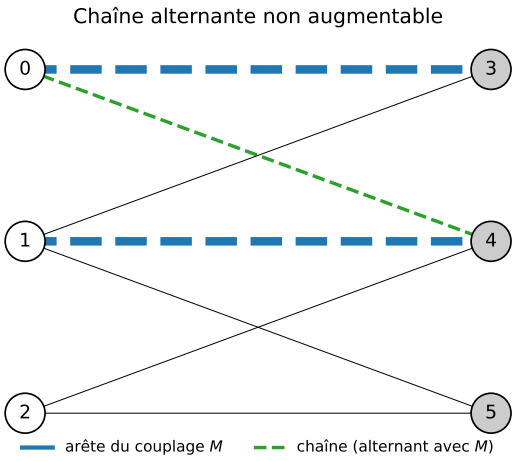
et de bipartition

$$B_4 = [0,0,0,1,1,1].$$

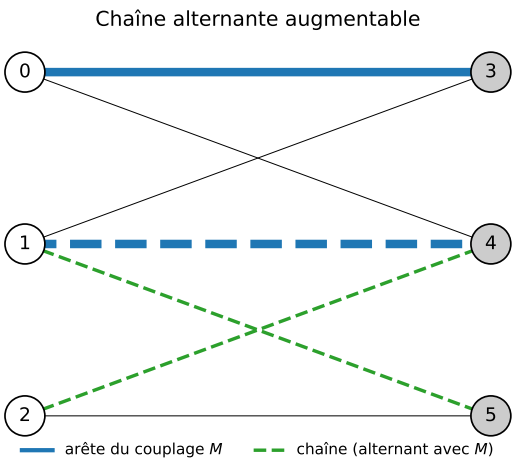
Remarque: un schéma propre ainsi que le résultat suffisent pour justification à cette question.

2.16. Le couplage retourné par la fonction `couplage_glouton` est-il toujours maximum ?

On s'intéresse désormais à un algorithme dit «d'augmentation» pour créer un couplage maximum.
Soit M un couplage de G.
Une **chaîne** de G est une suite de sommets $s_0, s_1, \ldots, s_k$ deux à deux distincts tels que pour tout $i \in [\![0, k-1]\!]$, les sommets s_i et s_{i+1} sont reliés par une arête de G, c'est à dire $\{s_i, s_{i+1}\} \in A$.
Une telle chaîne sera notée $s_0 - s_1 - \cdots - s_k$.
Une chaîne $s_0 - s_1 - \cdots - s_k$ est dite **alternante** relativement à un couplage M si les arêtes appartiennent alternativement à $A \setminus M$ et à M.
Une chaîne alternante $s_0 - s_1 - \cdots - s_k$ est dite **augmentable** si ses deux extrémités s_0 et s_k sont libres.
Par exemple, avec les mêmes notations que dans l'introduction de la section 2, C_1 est un couplage de G_3 et on a les chaînes suivantes:



La chaîne $1-4-0-3$ est une chaîne alternante relativement à C_1 (en pointillé), non augmentable car 3 est saturé par C_1.



La chaine $2-4-1-5$ est une chaîne alternante relativement à C_1 (en pointillé), augmentable car 2 et 5 sont libres.

2.17. Soit $\gamma = s_0 - s_1 - \cdots - s_k$ une chaîne alternante augmentable relativement à M. Que dire de la parité de k ?

2.18. Soit $\gamma = s_0 - s_1 - \cdots - s_k$ une chaîne alternante augmentable relativement à M.
On note $E(\gamma) = \{\{s_i, s_{i+1}\}, i \in [\![0, k-1]\!]\}$ l'ensemble des arêtes de γ ainsi que

$$\widetilde{M} = (M \setminus E(\gamma)) \cup (E(\gamma) \setminus M)$$

la différence symétrique entre M et $E(\gamma)$.
Montrer que $\widetilde{M}$ est un couplage de G et que $|\widetilde{M}| = |M| + 1$.

L'opération qui à M et γ associe $\widetilde{M}$ est appelée **augmentation** du couplage M le long de la chaîne γ.
On en déduit le résultat suivant, appelé **lemme de Berge** (il n'est pas demandé de le démontrer):

> **Théorème 1** (Lemme de Berge)
> Un couplage M de G est maximum si et seulement s'il n'existe pas de chaîne alternante augmentable relativement à M.

Le lemme de Berge suggère l'algorithme suivant pour construire un couplage maximum de G :
- Partir d'un couplage M quelconque (par exemple $M = \emptyset$).
- Tant qu'il existe une chaîne alternante augmentable γ relativement à M, augmenter M le long de γ.
- Retourner M.

Pour rechercher une chaîne alternante augmentable, on procède par parcours depuis chaque sommet libre $u \in S_0$: on explore le graphe en n'empruntant que des chaînes alternantes à partir de u. Une telle chaîne est augmentable dès qu'elle atteint un sommet libre de S_1.

2.19. Justifier qu'il suffit de chercher des chaînes augmentables partant d'un sommet libre de S_0 (et non de S_1).

2.20. Écrire une fonction python `chaine_augmentable(L, B, C, u)` qui prend en arguments une liste d'adjacence L d'un graphe biparti, sa bipartition B, un couplage C représenté par un dictionnaire et un sommet libre $u \in S_0$, et qui retourne soit `None` s'il n'existe pas de chaîne alternante augmentable issue de u, soit la liste des sommets d'une telle chaîne.
Indication: On pourra utiliser un parcours en profondeur, en mémorisant pour chaque sommet visité son prédécesseur dans le parcours afin de reconstruire la chaîne.

2.21. Écrire une fonction python `augmenter(C, gamma)` qui prend en arguments un couplage C et une chaîne alternante augmentable γ représentée par la liste de ses sommets, et qui modifie C pour l'augmenter le long de γ.

2.22. En déduire une fonction python `couplage_maximum(L, B)` qui prend en arguments une liste d'adjacence L d'un graphe biparti et sa bipartition B, et qui retourne un couplage maximum de G représenté par un dictionnaire.

2.23. Quelle est la complexité de la fonction `couplage_maximum(L, B)` en fonction de $n = |S|$ et de $p = |A|$? Commenter.