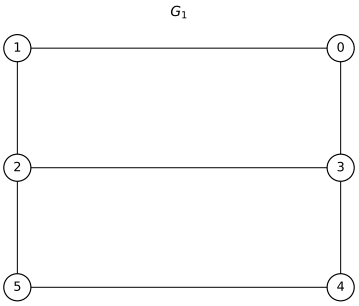
Graphes bipartis et couplages

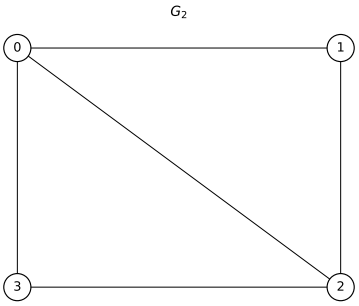
Graphe biparti

1.1

On obtient les figures suivantes :



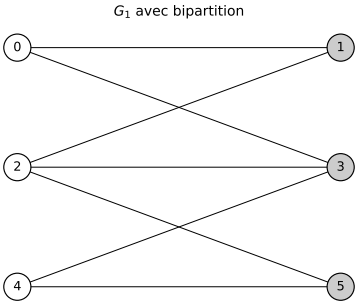
Graphe G1



Graphe G2

1.2

- Le graphe G_1 **est biparti**, par exemple la bipartition $S_0 = \{0, 2, 4\}$ et $S_1 = \{1, 3, 5\}$ convient.



Graphe G1 avec bipartition

- Le graphe G_2 **n'est pas biparti**.

En effet si par l'absurde il existe une bipartition S_0, S_1 alors quitte à intervertir on peut supposer que $0 \in S_0$. Alors comme $\{0, 2\}$ est une arête, $2 \in S_1$.

Et comme $\{0, 1\}$ est une arête, $1 \in S_1$ et de même comme $\{1, 2\}$ est une arête, $1 \in S_0$. Absurde.

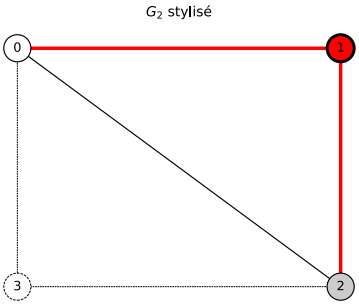


Illustration de l'absurdité

1.3

On obtient le code suivant:

In [1]:
```python
def est_bipartition(L, B):
    """Vérifie que B est une bipartition de L."""
    n = len(L)
    for u in range(n):
        for v in L[u]:
            if B[u] == B[v]:  # u et v sont dans la même partie
                return False
    return True
```

1.4

On obtient le code suivant

In [2]:
```python
def ecriture_binaire(k, n):
    """Retourne la représentation binaire de k sur n bits."""
    r = k
    B = []
    for i in range(n):
        # print(r * 2**i + sum(B[j] * (2**j) for j in range(i)))
        B.append(r % 2)
        r = r // 2
    return B
```

on a une version récursive suivante

In [3]:
```python
def ecriture_binaire_recursive(k, n):
    """Retourne la représentation binaire de k sur n bits (version récursive)."""
    if n == 0:
        return []
    else:
        return [k % 2] + ecriture_binaire_recursive(k // 2, n - 1)
```

1.5

Terminaison

- Pour la version itérative, il s'agit d'une boucle `for` .
- Pour la version récursive la variable `n` est un variant de boucle (strictement décroissant et à valeurs dans les entiers naturels).

Correction

On note r_i la valeur de la variable `r` au début de l'itération i et B_j la valeur de `B[j]` (fixée à l'itération j). À l'itération i de la boucle, ou bien au i-ème appel récursif, on a l'invariant de boucle suivant:

$$\boxed{k = r \cdot 2^i + \sum_{j=0}^{i-1} B[j] \cdot 2^j}.$$

En effet:

- Avant la première itération ($i = 0$), on a $r_0 = k$ et la somme est vide, donc $k = r_0 \cdot 2^0 + 0$.
- Si l'invariant est vérifié à l'itération i, en notant avec les indices i et $i+1$ les variables aux itérations i et $i+1$, on a

$$\begin{aligned} r_{i+1} \cdot 2^{i+1} + \sum_{j=0}^{i} B_{i+1}[j] \cdot 2^j &= (r_i//2) \cdot 2^{i+1} + (r_i\%2) \cdot 2^i + \sum_{j=0}^{i-1} B_i[j] \cdot 2^j \\ &= (2(r_i//2) + (r_i\%2)) \cdot 2^i + \sum_{j=0}^{i-1} B_i[j] \cdot 2^j \\ &= r_i \cdot 2^i + \sum_{j=0}^{i-1} B_i[j] \cdot 2^j \\ &= k \quad \text{(par hypothèse de récurrence)}. \end{aligned}$$

1.6

En utilisant les fonctions suivantes, on obtient

In [4]:
```python
def est_biparti_naif(L):
    """Vérifie si L est un graphe biparti en essayant toutes les bipartitions possibles."""
    n = len(L)
    for k in range(2**n):
        B = ecriture_binaire(k, n)
        if est_bipartition(L, B):
            return (True, B)
    return (False, None)
```

1.7

- La complexité de la fonction `ecriture_binaire` est $O(n)$ (une boucle de n itérations).
- La complexité de la fonction `est_bipartition` est $O(n+p)$, en effet on parcourt chaque sommet et chaque arête une fois.
- La complexité finale est donc de

$$\boxed{O(2^n(n+p))}.$$

1.8

En complétant on obtient le code suivant

In [5]:
```
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 B[s] is None:
            B[s] = 0
            file = deque([s])
            # On effectue un parcours en largeur à partir de s
            while file:
                u = file.popleft()
                for v in L[u]:
                    if B[v] is None:
                        B[v] = 1 - B[u]
                        file.append(v)
                    else:
                        if B[u] == B[v]:
                            return (False, None)
    return (True, B)
```

1.9

Dans le parcours en largeur on visite chaque sommet au plus une fois et chaque arête au plus une fois.

La complexité est donc

$$\boxed{O(n+p)}.$$

1.10

La fonction `est_biparti_bfs` est beaucoup **plus efficace** que la fonction `est_biparti_naif` .

Couplage dans un graphe biparti

2.11

- On a $|C_1| = 2$ et il existe des couplages de taille 3 par exemple C_2.

Donc C_1 **n'est pas un couplage maximum**.

- On a $|C_2| = 3$. De plus pour tout couplage C comme à chaque arête du couplage on peut associer deux sommets de manière injective on a (lemme des bergers) $2|C| \leq |S|$.

Donc ici $2|C| \leq 6$ d'où $|C| \leq 3$.

Donc C_2 **est un couplage maximum**.

2.12

On obtient

In [6]:
```
def est_couplage(L, C):
    """Vérifie que C est un couplage de L."""
    n = len(L)
    for u in C:  # On parcourt les clés du dictionnaire C
        # On vérifie que C[u] est un voisin de u
        est_voisin = False
        for v in L[u]:
            if v == C[u]:
                est_voisin = True
        if not est_voisin:
            return False
        # On vérifie que C[u] = v ssi C[v] = u
        if C[C[u]] != u:
            return False
        # Automatiquement aucune arête du couplage n'a un sommet en commun car C est un dictionnaire avec une seul valeur pour chaque c
    return True
```

2.13

En suivant l'algorithme glouton de l'énoncé on obtient

In [7]:
```python
def couplage_glouton(L, B):
    """Retourne un couplage de L à partir d'une bipartition B"""
    n = len(L)
    C = {}
    est_sature = [False for u in range(n)]
    for u in range(n):
        if (B[u] == 0) and (not est_sature[u]):  # u est dans la partie 0 et libre
            est_sature[u] = True
            for v in L[u]:
                if (B[v] == 1) and (
                    not est_sature[v]
                ):  # v est dans la partie 1 et est libre
                    est_sature[v] = True
                    C[u] = v
                    C[v] = u
                    break  # On arrête de parcourir les voisins de u
    return C
```

Sans utiliser de break on peut aussi récupérer le premier sommet libre de S_1 voisin de u ou bien faire une fonction annexe.

In [8]:
```python
def voisin_S1_libre(L, B, u, est_sature):
    """Retourne un voisin libre de u dans S1 ou None s'il n'en existe pas."""
    for v in L[u]:
        if (B[v] == 1) and (not est_sature[v]):
            return v
    return None


def couplage_glouton_sans_break(L, B):
    """Retourne un couplage de L à partir d'une bipartition B sans utiliser de break."""
    n = len(L)
    C = {}
    est_sature = [False for u in range(n)]
    for u in range(n):
        if (B[u] == 0) and (not est_sature[u]):  # u est dans la partie 0 et libre
            est_sature[u] = True
            v = voisin_S1_libre(L, B, u, est_sature)
            if v is not None:
                est_sature[v] = True
                C[u] = v
                C[v] = u
    return C
```

2.14

On parcourt chaque arête et chaque sommet au plus une fois, la complexité est donc $\boxed{O(n+p)}$.

2.15

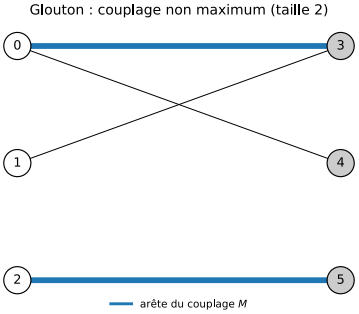
On a les étapes suivantes:

- $u = 0$ (dans S_0, libre) : voisins $[3, 4]$. Le sommet 3 est libre, on l'apparie : $C = \{0:3,\ 3:0\}$. Les sommets 0 et 3 sont saturés.
- $u = 1$ (dans S_0, libre) : voisins $[3]$. Le sommet 3 est saturé ; 1 **reste libre**.
- $u = 2$ (dans S_0, libre) : voisins $[5]$. Le sommet 5 est libre, on l'apparie : $C = \{0:3,\ 3:0,\ 2:5,\ 5:2\}$.
- $u = 3$ (dans S_1).
- $u = 4$ (dans S_1).
- $u = 5$ (dans S_1).

On obtient finalement le couplage

$$\boxed{C = \{0:3,\ 3:0,\ 2:5,\ 5:2\}}$$

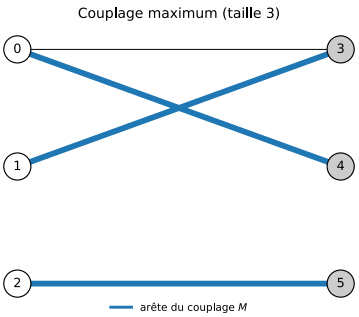
ce qui correspond au schéma



Couplage glouton pour G4

2.16

On voit sur le graphe précédent que le couplage précédent est de taille 2 alors qu'il existe des couplages de taille 3 par exemple



Couplage de taille 3 pour G4

Et donc **le couplage glouton n'est pas toujours couplage maximum**.

2.17

On a k **est impair**.

Comme s_0 et s_k sont libres, on a en particulier $s_0 - s_1 \notin M$ et $s_{k-1} - s_k \notin M$.

Par alternance entre l'appartenance à M et la non appartenance à M on a donc $k-0$ est impair.

2.18

Cardinal de $\widetilde{M}$.

On a vu à la question précédente que comme γ est alternante avec extrémités libres, que:

- les arêtes $\{s_0, s_1\}, \{s_2, s_3\}, \dots, \{s_{k-1}, s_k\}$ (d'indice pair, au nombre de $\frac{k+1}{2}$) sont **hors de** M ;
- les arêtes $\{s_1, s_2\}, \{s_3, s_4\}, \dots, \{s_{k-2}, s_{k-1}\}$ (d'indice impair, au nombre de $\frac{k-1}{2}$) sont **dans** M.

Donc $E(\gamma) \cap M$ contient $\frac{k-1}{2}$ arêtes et $E(\gamma) \setminus M$ en contient $\frac{k+1}{2}$.

D'où

$$|\widetilde{M}| = |M| - |M \cap E(\gamma)| + |E(\gamma) \setminus M| = |M| - \tfrac{k-1}{2} + \tfrac{k+1}{2} = |M| + 1.$$

$\widetilde{M}$ **est un couplage.** Il faut montrer que deux arêtes distinctes de $\widetilde{M}$ ne partagent aucun sommet.

Soit u un sommet de G. On procède par disjonction de cas pour montrer qu'il n'existe pas deux arêtes de $\widetilde{M}$ incidentes à u.

- *1er cas :* $u \notin \gamma$. Alors aucune arête de $E(\gamma)$ n'est incidente à u (sinon u serait un des s_i). Donc les arêtes de $\widetilde{M}$ incidentes à u sont les mêmes que celles de M : au plus une.
- *2ème cas :* $u \in \gamma$, *disons* $u = s_i$ *pour un* $i \in [\![0, k]\!]$.
  - Sous-cas $i = 0$ ou $i = k$ (extrémité libre de γ) : u n'était saturé par aucune arête de M. Et dans γ, u est extrémité d'une unique arête (celle indexée 0 ou $k-1$), qui est hors de M donc passe dans $\widetilde{M}$. Donc u est saturé par **exactement une** arête de $\widetilde{M}$.
  - Sous-cas $0 < i < k$: $u = s_i$ est incident dans γ à deux arêtes $\{s_{i-1}, s_i\}$ et $\{s_i, s_{i+1}\}$, l'une dans M et l'autre hors de M (par alternance). Donc dans $\widetilde{M}$, la première disparaît et la seconde apparaît (ou vice-versa) : u est saturé par **exactement une** arête de $\widetilde{M}$ parmi les arêtes de γ. De plus, u étant déjà saturé dans M par l'unique arête de γ dans M qui lui est incidente, il n'est saturé par aucune autre arête de M. Donc u est saturé par exactement une arête de $\widetilde{M}$.

Dans tous les cas, chaque sommet est saturé par au plus une arête de $\widetilde{M}$, donc $\widetilde{M}$ est bien un couplage.

2.19

Soit $s_0 - \cdots - s_k$ une chaîne augmentable de M.

On a déjà vu que qu'une chaîne augmentable est de longueur impaire.

Donc $s_0 \in S_0$ et $s_k \in S_1$, ou bien $s_k \in S_0$ et $s_0 \in S_1$.

Dans le cas où $s_0 \in S_1$ alors la **chaîne renversée** $s_k - \cdots - s_0$ est une chaîne augmentable de M qui commence dans S_0 et finit dans S_1, et avec exactement les mêmes arêtes que la chaîne $s_0 - \cdots - s_k$.

2.20

On effectue un **parcours en profondeur** depuis u qui n'emprunte que des chaînes alternantes :

- depuis un sommet $x \in S_0$, on emprunte uniquement les arêtes **hors du couplage** (vers S_1) ;
- depuis un sommet $x \in S_1$, on emprunte uniquement l'arête **du couplage** qui revient dans S_0 (s'il y en a une).

Dès qu'on atteint un sommet libre de S_1, on a trouvé une chaîne augmentable. On mémorise les prédécesseurs `pred[v]` pour reconstruire la chaîne.

In [9]:
```
def chaine_augmentable(L, B, C, u):
    """Renvoie la liste des sommets d'une chaîne augmentable issue de u,
    ou None s'il n'en existe pas."""
```

```
    n = len(L)
    pred = {u: None}
    est_visite = [False for _ in range(n)]
    est_sature = [False for _ in range(n)]
    for s in C:
        est_sature[s] = True
    pile = [u]
    while pile:
        x = pile.pop()
        if B[x] == 0:
            # Depuis S_0 : on suit les arêtes hors couplage
            for v in L[x]:
                if (not est_visite[v]) and not (est_sature[x] and C[x] == v):
                    est_visite[v] = True
                    pred[v] = x
                    if not est_sature[v]:
                        # Si v libre dans S_1 : on reconstruit la chaîne
                        chaine = []
                        cur = v
                        while cur is not None:
                            chaine.append(cur)
                            cur = pred[cur]
                        return chaine[::-1]
                    # Si on a pas renvoyé la chaîne augmentable, on continue à explorer les voisins de x
                    pile.append(v)
        else:
            # Depuis S_1 : on suit l'arête du couplage
            w = C[x]
            est_visite[w] = True
            pred[w] = x
            pile.append(w)
    return None
```

2.21

Augmenter le couplage C selon $\gamma = s_0 - s_1 - \cdots - s_k$ revient à faire les opérations suivantes:

- **ajouter** au couplage les arêtes $\{s_i, s_{i+1}\}$ pour i **pair** (arêtes de γ hors couplage) ;
- **retirer** du couplage les arêtes $\{s_i, s_{i+1}\}$ pour i **impair** (arêtes de γ dans le couplage).

On remarque qu'on peut faire ces deux opérations simultanément avec une seule affectation. En effet, pour i pair, l'arête $\{s_i, s_{i+1}\}$ à **ajouter** et l'arête $\{s_{i+1}, s_{i+2}\}$ à **retirer** partagent le sommet s_{i+1}. Comme dans un couplage chaque sommet est extrémité d'au plus une arête, écrire `C[s[i+1]] = s[i]` **écrase** automatiquement l'ancienne valeur `C[s[i+1]] = s[i+2]` : l'ajout de la nouvelle arête fait disparaître l'ancienne.

Il suffit donc de parcourir les indices i pairs de 0 à $k-1$ et de poser

$$C[s_i] = s_{i+1} \quad \text{et} \quad C[s_{i+1}] = s_i.$$

On obtient donc

In [10]:
```
def augmenter(C, gamma):
    """Augmente le couplage C à l'aide de la chaîne augmentable gamma."""
    k = len(gamma) - 1
    for i in range(0, k, 2):  # Par pas de 2
        u = gamma[i]
        v = gamma[i + 1]
        C[u] = v
        C[v] = u
```

2.22

On obtient donc.

In [11]:
```
def couplage_maximum(L, B):
    """Retourne un couplage maximum de L à partir d'une bipartition B."""
    n = len(L)
    C = {}
    for u in range(n):
        if (B[u] == 0) and (not u in C):
            gamma = chaine_augmentable(L, B, C, u)
            if gamma != None:
                augmenter(C, gamma)
    return C
```

2.23

- La fonction `chaine_augmentable` est un parcours en profondeur et est donc de complexité $O(n+p)$.
- La fonction `augmenter` est de complexité $O(k)$ où k est la longueur de la chaîne augmentable, et donc $O(n)$.

D'où la complexité de `couplage_maximum` est $\boxed{O(n(n+p))}$, en effet dans le pire des cas on trouve une chaîne augmentable pour chaque sommet de S_0 et chacune de ces chaînes est de longueur $O(n)$.