TD Info 7 : Graphes, jeux et heuristiques
Proposition de correction

Contents

[1]:
```
import networkx as nx
import matplotlib.pyplot as plt
from collections import deque
```

1 Dictionnaires : rappels et compléments

On profite de cette séance pour rappeler quelques notions sur les dictionnaires, et en présenter d'autres complémentaires.

1.1 Utilisation de dictionnaires

Création d'un dictionnaire :

[2]:
```
Stocks = dict()
Stocks["oranges"] = 18
Stocks["pommes"] = 3
Stocks["kiwis"] = 41
Stocks
```

[2]:
```
{'oranges': 18, 'pommes': 3, 'kiwis': 41}
```

Un dictionnaire est une *table d'association* : on associe, à chaque **clef** du dictionnaire (un fruit dans l'exemple étudié ici) une **valeur** (nombre de fruits en stock ici). On peut ensuite modifier la valeur d'une clef :

[3]:
```
Stocks["kiwis"] -= 5
Stocks
```

[3]:
```
{'oranges': 18, 'pommes': 3, 'kiwis': 36}
```

On peut obtenir les clefs, la valeurs et les couples (clef, valeur) en utilisant les commandes suivantes :

[4]:
```
print(Stocks.keys())
print(Stocks.values())
print(Stocks.items())
```

```
dict_keys(['oranges', 'pommes', 'kiwis'])
dict_values([18, 3, 36])
dict_items([('oranges', 18), ('pommes', 3), ('kiwis', 36)])
```

Pour parcourir un dictionnaire, plusieurs possibilités, la plus courante et simple est la suivante :

[5]:
```
for clef in Stocks : # raccourci pour for clef in Stocks.keys()
    print(clef, Stocks[clef])
```

```
oranges 18
pommes 3
kiwis 36
```

Pour tester si une clef existe :

[6]:
```
print("kiwis" in Stocks)
print("mandarines" in Stocks)
```

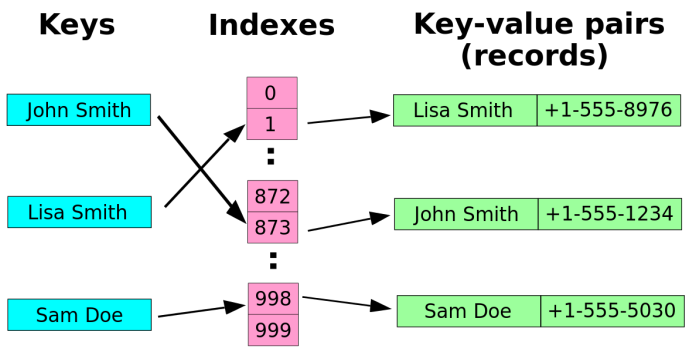
```
True
False
```

De nombreuses fonctions existent pour les dictionnaires, dont un certain nombre similaires à celles pour les listes (`len`, `copy`).

1.2 Quelques explications théoriques : table de hachage

Du point de vue de la machine, les données sont stockées à des emplacements mémoires, à des adresses données. On peut modéliser simplement ces adresses par un nombre entier, un indice. Il est donc nécessaire, pour construire une structure de donnée de type *dictionnaire*, d'avoir une correspondance entre les *clefs* et les *indices* : cette correspondance est assurée par une **fonction de hachage**.

Sur l'exemple ci-dessous, pour créer un dictionnaire associant des numéros de téléphone à des noms, on définit une table d'indices. La fonction de hachage est la fonction permettant de passer des noms à ces indices :



Pour notre exemple de dictionnaire `Stocks`, essayons la fonction de hachage suivante :

[7]:
```
import string

# Création dictionnaire lettre : indice
alphabet = list(string.ascii_lowercase)
Dict_alphabet = dict()
for i in range(len(alphabet)) :
    lettre = alphabet[i]
    Dict_alphabet[lettre] = i

def fonction_h(chaine) :
    """ Fonction de hachage : retourne pour chaque mot 'chaine' un nombre␣
 ↪entier
    qui est la somme des places dans l'aphabet des lettre constituants␣
 ↪'chaine'"""
    somme = 0
    for lettre in chaine :
        somme += Dict_alphabet[lettre]
    return somme
```

[8]:
```
Dict_alphabet
```

[8]:
```
{'a': 0,
 'b': 1,
 'c': 2,
 'd': 3,
 'e': 4,
 'f': 5,
 'g': 6,
 'h': 7,
 'i': 8,
 'j': 9,
 'k': 10,
 'l': 11,
 'm': 12,
```

```
 'n': 13,
 'o': 14,
 'p': 15,
 'q': 16,
 'r': 17,
 's': 18,
 't': 19,
 'u': 20,
 'v': 21,
 'w': 22,
 'x': 23,
 'y': 24,
 'z': 25}
```

[9]:
```
for fruit in Stocks:
    print('Indice de '+fruit+' donné par la fonction de hachage :␣
 ↪'+str(fonction_h(fruit)))
```

```
Indice de oranges donné par la fonction de hachage : 72
Indice de pommes donné par la fonction de hachage : 75
Indice de kiwis donné par la fonction de hachage : 66
```

Cette fonction fonctionne bien pour notre exemple simple, car les trois fruits correspondent à trois indices différents : on dit qu'il n'y a pas de **collisions** (cela correspond au fait que la fonction, sur l'espace de départ considéré, est injective !). Mais si on veut ajouter un légume :

[10]:
```
fonction_h('carotte') == fonction_h('pommes')
```

[10]:
```
True
```

Collision ! Notre fonction de hachage ne sera alors pas suffisante pour ce dictionnaire. Notre fonction de hachage possède bien d'autres défauts : elle ne fonctionne qu'avec des mots *simples* (pas de tiret, pas d'accent, d'apostrophes, …) sans majuscules. Python propose sa propre fonction de hachage, `hash` :

[11]:
```
for fruit in Stocks:
    print('Indice de '+fruit+' donné par la fonction de hachage :␣
 ↪'+str(hash(fruit)))
```

```
Indice de oranges donné par la fonction de hachage : -5572143226191969059
Indice de pommes donné par la fonction de hachage : -3287300018984216149
Indice de kiwis donné par la fonction de hachage : -4508270496678838083
```

C'est évidemment une fonction bien plus complexes et puissante. Vous pouvez d'ailleurs remarquer qu'elle ne retourne pas les mêmes résultats selon l'ordinateur considéré.

Un autre intérêt du principe de la table de hachage, qui la rend nécessaire : elle permet un accès accès en un temps $\mathcal{O}(1)$ à la bonne adresse mémoire. Il n'est en effet pas nécessaire de parcourir toutes les clefs précédentes pour trouver la clef voulue, grâce il suffit d'appliquer une fois la fonction de hachage.

Enfin il est important de noter que le principe de la table de hachage impose que les clefs doivent être des objets non modifiables (*non mutables*), comme des nombres, des chaînes de caractères, des tuples, … mais pas des listes !

2 Jeux d'accessibilité à 2 joueurs

On s'intéresse à des jeux respectant les conditions suivantes :

- deux joueurs jouent à tour de rôle, en connaissant l'ensemble de la situation (pas de *cartes cachées* par exemple);
- sans mémoire : la stratégie à adopter à un moment donné du jeu ne dépend pas de l'*histoire* passée ;
- sans hasard.

Exemples de tels jeux : dames, échecs, go, …

Ces jeux peuvent être modélisés par des graphes orientés finis : chaque sommet est une situation (on parle aussi de position) et chaque arête correspond à une décision, un coup possible. On peut alors voir le jeu comme le déplacement d'un jeton sur ce graphe : on démarre sur un sommet donné et à tour de rôle les joueurs déplacent le jeton en suivant des arêtes. On s'intéressera plus particulièrement aux jeux d'*accessibilité* : pour gagner, chaque joueur a pour but d'atteindre un ou plusieurs sommets particuliers.

Dans un premier, nous allons nous intéresser à un jeux *simple* pour lequel il est possible de déterminer une stratégie gagnante, puis nous verrons, pour les jeux les plus complexes, comment bâtir une stratégie à l'aide d'une heuristique.

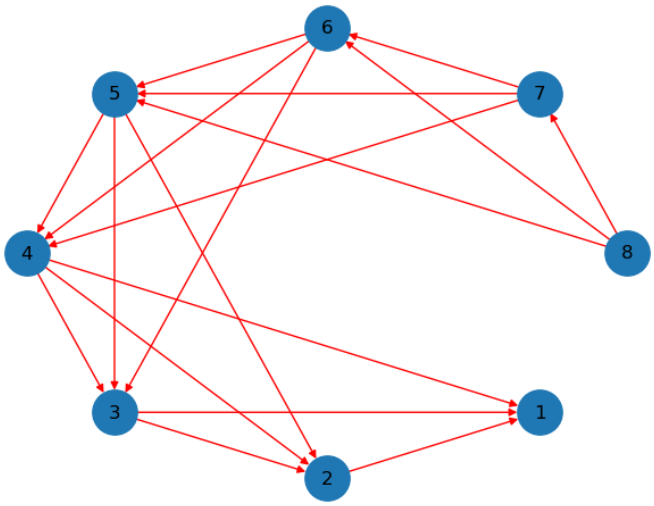
2.1 Présentation du jeux de Nim

Le jeux de Nim est devenu populaire en France à travers l'émission *Fort Boyard*, qui en présente une version. D'autres versions existent, avec d'autres règles.

Question 1 : Compléter le code suivant pour représenter le graphe de ce jeux : chaque sommet est une situation possible (qu'on numérote selon le nombre de bâtonnets restants), et chaque arc un coup possible. On se limite ici à la fin du jeu, avec au plus 8 bâtonnets restants. On utilisera ensuite le code fourni pour représenter le graphe.

[10]:
```
Dict_Nim = {'8': ['7', '6', '5'],
            '7': ['6', '5', '4'],
            '6': ['5', '4', '3'],
            '5': ['4', '3', '2'],
            '4': ['3', '2', '1'],
            '3': ['2', '1'],
            '2': ['1'],
            '1': []}

# Création du graphe en networkx :
Graphe_Nim = nx.DiGraph(Dict_Nim)
```

```
# Représentation graphique à l'aide de Matplotlib :
nx.draw_circular(Graphe_Nim, node_size=800, edge_color='red', with_labels=True)
plt.show()
```



Ce graphe orienté est nommé *arène* . Ce jeux fait partie des jeux d'*accessibilité* car le graphe associé ne comporte pas de cycle (ce qui assure que toute partie est finie), et il est déterminé par un ensemble de positions (les cibles) qui sont sans successeurs. Ainsi, dans le jeu de Nim, le nœud 1 du graphe associé est la seule cible du jeu, et l'atteindre signifie la fin de la partie (et la défaite pour celui qui l'atteint).

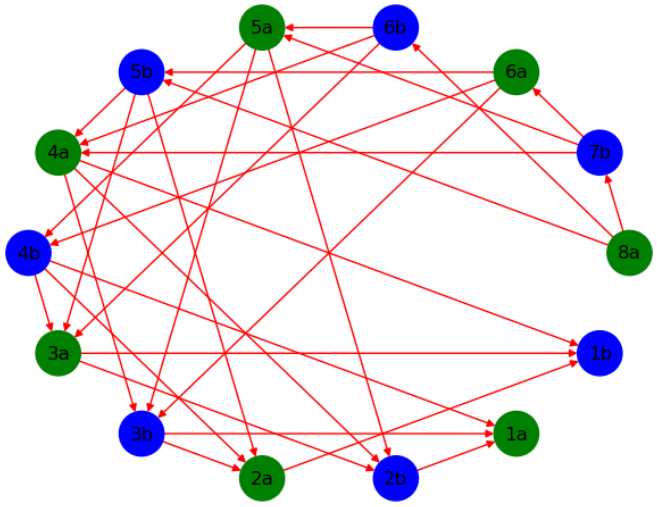
2.2 Graphe biparti

L'inconvénient de de la représentation précédente est qu'elle ne permet pas de distinguer entre les coups réalisés par le premier joueur (nommé joueur **a**) et ceux réalisés par le second (joueur **b**). Pour pallier ce problème, on ne va pas seulement représenter les positions possibles mais aussi différencier les sommets selon s'ils correspondent à un tour du premier ou du second joueur. Pour ce faire, on va en quelque sorte *dédoubler* ce graphe (sauf la position de départ qui ne peut être jouée que par le premier joueur) : on créera ainsi un **graphe biparti**.

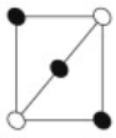
Question 2 : Compléter le code suivant permettant de définir un nouveau dictionnaire d'adjacence correspondant à un graphe biparti. Pour cela, les noms des sommets seront indexés par la lettre a si c'est au tour du premier joueur de retirer des bâtonnets, b sinon. On utilisera ensuite le code fourni pour représenter le graphe.

[11]:
```
Dict_B_Nim = {'8a': ['7b', '6b', '5b'],
              '7b': ['6a', '5a', '4a'],
              '6a': ['5b', '4b', '3b'],
              '6b': ['5a', '4a', '3a'],
              '5a': ['4b', '3b', '2b'],
              '5b': ['4a', '3a', '2a'],
              '4a': ['3b', '2b', '1b'],
              '4b': ['3a', '2a', '1a'],
```

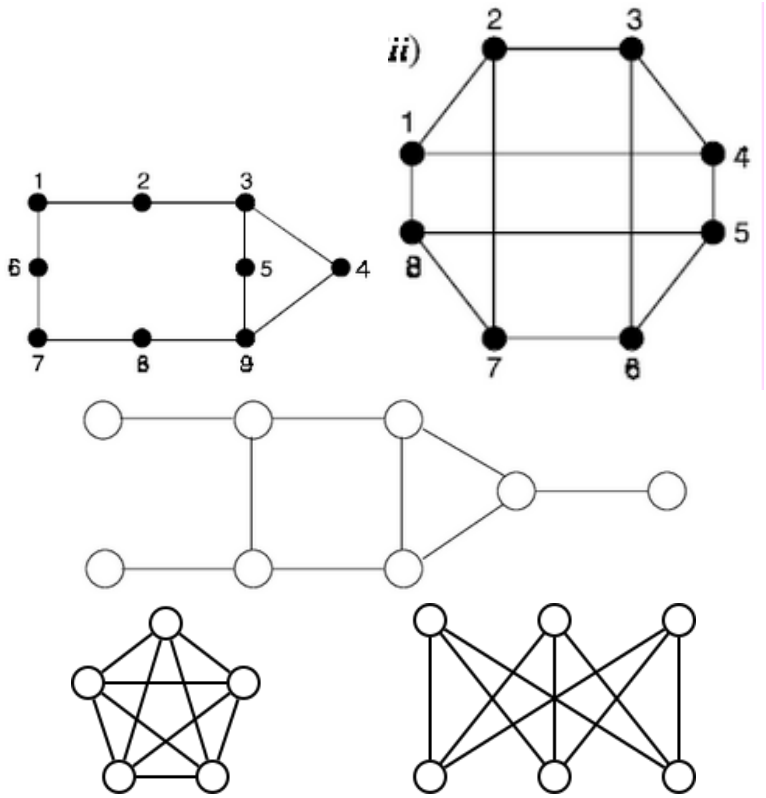
```
                '3a': ['2b', '1b'],
                '3b': ['2a', '1a'],
                '2a': ['1b'],
                '2b': ['1a'],
                '1a': [],
                '1b': []}

# Création du graphe en networkx :
Graphe_B_Nim = nx.DiGraph(Dict_B_Nim)

# Représentation graphique à l'aide de Matplotlib :
color_map = []
for node in Graphe_B_Nim:
    if 'a' in node :
        color_map.append('green')
    else:
        color_map.append('blue')
nx.draw_circular(Graphe_B_Nim, node_color=color_map, node_size=800,␣
 ↪edge_color='red', with_labels=True)
plt.show()
```



Dans ce graphe, comme dans tout graphe biparti à 2 joueurs, un arc ne peut relier qu'un sommet a à un sommet b et réciproquement. Tous ces graphes **bipartis**, comme leur nom l'indique, sont composés de *deux parties*, ici de différentes couleurs. On peut même en faire une définition : *Un graphe est biparti si et seulement si on peut colorer ses sommets avec seulement deux couleurs sans que deux sommets voisins n'aient la même couleur.* On peut le voir sur l'exemple ci-dessous :

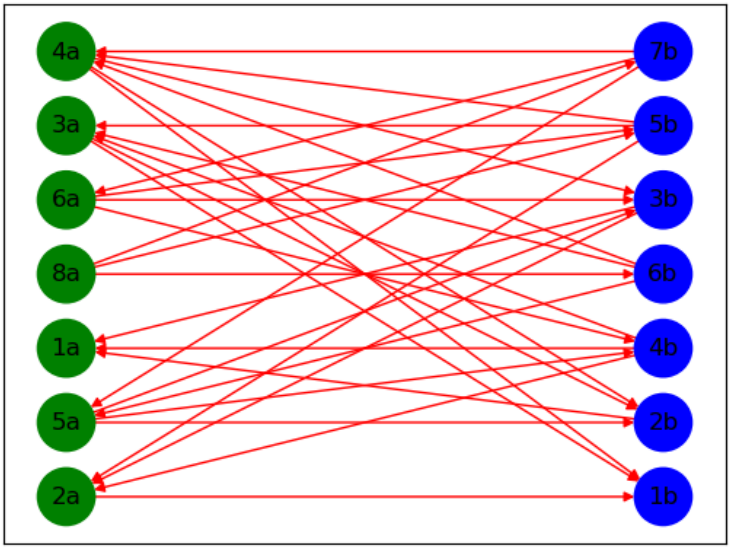


Question 3 : Pour les graphes ci-dessous, dire s'ils sont biparti ou non. Dans le cas affirmatif, donner la partition (c'est-à-dire les sommets formant les deux *sous-graphes*).



Remarque : Un graphe biparti peut être représenté en deux colonnes ou deux lignes (comme le dernier ci-dessus), chaque ligne ou colonne correspondant à une couleur, de sorte qu'il n'y ait pas d'arrête au sein d'une même ligne ou colonne.

[17]:
```
nx.draw_networkx(Graphe_B_Nim,pos = nx.drawing.layout.
↪bipartite_layout(Graphe_B_Nim, {'8a', '6a', '5a', '4a', '3a', '2a', '1a'}),␣
↪node_color=color_map, node_size=800, edge_color='red', with_labels=True)
```



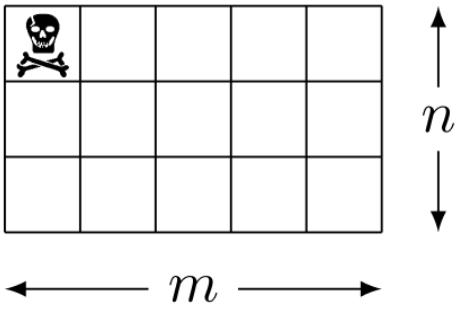
Un **état (ou position) final** correspond à un sommet duquel ne part aucun arc : il correspond à une fin de partie. Ces états peuvent est classé en trois catégories : les états gagnants pour le joueur **a**, les états gagnants pour le joureur **b**, et les états de match nul.

Question 4 : Dans le graphe biparti du jeu de nim, combien existe-t-il d'états finaux ? Les classer.

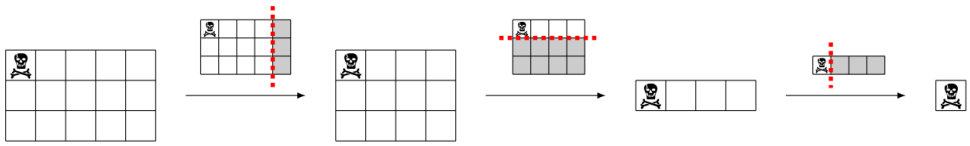
Il existe 2 états finaux : le sommets 1b, état gagnant pour le joueur **a**, et le sommet 1a, état gagnant pour le joueur **b**. Il n'y a aucun état de match nul.

2.3 Un autre exemple : le jeu de Chomp

On considère une version simplifiée de ce jeux dans laquelle le "plateau" est similaire à une tablette de chocolat de n lignes et m colonnes, et dont le carreau situé en haut à gauche (première ligne et première colonne) est empoisonné.

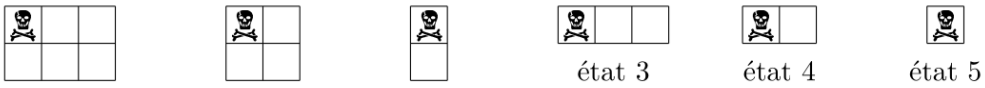


À tour de rôle, les deux joueurs coupent la tablette suivant une verticale ou une horizontale et mange les colonnes à droite ou les lignes en dessous de cette découpe. La partie se termine lorsqu'il ne reste que le carré empoisonné et que le joueur doit donc le manger. Voici un exemple de partie :



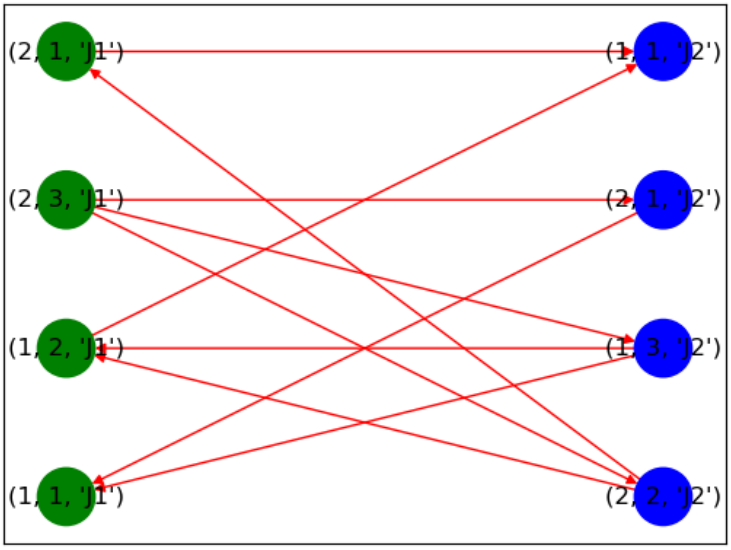
Question 5 : On part d'une tablette composée de 2 lignes et 3 colonnes. Quels sont les différents états possibles ? Quels sont les états finaux ? Écrire alors le code permettant de définir le dictionnaire `Dict_Chomp` du graphe biparti correspondant à ce jeux. On nommera les états (sommets du graphe) par un tuple `(i, j, Jk)` avec `i` et `j` le nombre de lignes et colonnes restantes de la tablette, et `J1` ou `J2` le joueur. Les valeurs associées seront une liste de tuples (chaque tuple étant un sommet voisin). On utilisera ensuite le code fourni pour représenter le graphe.

Voici les 5 états possibles, avec l'état de départ (état 0) et l'unique état final (état 5, état doublé si on considère les deux joueurs):



[19]:
```
Dict_Chomp = {(2, 3, 'J1'): [(2, 2, 'J2'), (2, 1, 'J2'), (1, 3, 'J2')],
              (2, 2, 'J2'): [(2, 1, 'J1'), (1, 2, 'J1')],
              (1, 3, 'J2'): [(1, 2, 'J1'), (1, 1, 'J1')],
              (2, 1, 'J2'): [(1, 1, 'J1')],
              (2, 1, 'J1'): [(1, 1, 'J2')],
              (1, 2, 'J1'): [(1, 1, 'J2')],
              (1, 1, 'J2'): [],
              (1, 1, 'J1'): []}
```

[21]:
```
# Création du graphe en networkx :
Graphe_Chomp = nx.DiGraph(Dict_Chomp)

# Représentation graphique à l'aide de Matplotlib :
color_map = []
for node in Graphe_Chomp:
    if 'J1' == node[2] :
        color_map.append('green')
    else:
        color_map.append('blue')
nx.draw_networkx(Graphe_Chomp,pos = nx.drawing.layout.
 ↪bipartite_layout(Graphe_Chomp, {(2, 3, 'J1'),(2, 1, 'J1'), (1, 2, 'J1'), (1,␣
 ↪1, 'J1')}), node_color=color_map, node_size=800, edge_color='red',␣
 ↪with_labels=True)
plt.show()
```



2.4 Application du parcours d'un graphe : bipartisme

On rappelle l'algorithme de parcours en largeur d'un graphe :

[16]:
```
def creer_file():
    return deque()

def est_vide(F):
    """ Retourne un booléen : True si la file F est vide, False sinon """
    return len(F) == 0

def enfiler(F,v):
    """ Ajoute l'élément v à la fin de la file F, v devient le dernier élément,␣
 ↪celui tout à droite"""
    F.append(v)

def defiler(F):
    """ Enlève l'élément au début de la file F, soit son premier élément, celui␣
 ↪tout à gauche"""
    if est_vide(F):
        raise ValueError("Erreur : file vide")
    else :
        F.popleft()
    return(F)

def début(F):
    """ Retourne le début de la file F, soit son premier élément """
    if est_vide(F):
```

```
        raise ValueError("Erreur : file vide")
    else :
        return F[0]
```

Question 6 : Adapter ou utiliser la fonction précédente pour écrire une fonction `est_biparti` qui prend en argument un graphe (sous forme d'un dictionnaire d'adjacence) et un sommet de départ, et retourne un booléen (`True` si le graphe est bien biparti, `False` sinon).

[17]:
```
def est_biparti(Graphe, sommet_dep):

    Visités = [] # Liste des sommets déjà visités
    A_visiter = creer_file() # File des sommets à visiter
    enfiler(A_visiter, sommet_dep)

    Dict_coul = dict() # clefs : sommet / valeur associée : couleur
    Couleur = ('bleu', 'vert') # deux couleurs possibles
    Dict_coul[sommet_dep] = Couleur[0] # on colorie le sommet de départ

    while est_vide(A_visiter) == False :

        # On considère un sommet de la file encore non visité :
        if début(A_visiter) in Visités :
            defiler(A_visiter)
        else :
            sommet_vis = début(A_visiter)
            # On indique qu'on l'a visité :
            defiler(A_visiter)
            Visités.append(sommet_vis)

            # Couleur du sommet actuellement visité :
            coul = Dict_coul[sommet_vis]

            # On change la couleur pour les voisins :
            nouv_coul = Couleur[0]
            if nouv_coul == coul :
                nouv_coul = Couleur[1]

            # On met à jour la file des sommets à visiter, à partir des voisins␣
↪:
            for sommet_adj in Graphe[sommet_vis] :
                # on attribue la nouvelle couleur aux voisins encore non␣
↪coloriés
                if sommet_adj not in Dict_coul :
                    Dict_coul[sommet_adj] = nouv_coul
                # si un voisin est déjà colorié, avec la "mauvaise" couleur
                elif Dict_coul[sommet_adj] != nouv_coul :
                        return False
```

```
                if sommet_adj not in Visités :
                    enfiler(A_visiter, sommet_adj)

    print(Dict_coul)
    return True
```

[18]:
```
est_biparti(Dict_B_Nim, '8a')
```

```
{'8a': 'bleu', '7b': 'vert', '6b': 'vert', '5b': 'vert', '6a': 'bleu', '5a':
'bleu', '4a': 'bleu', '3a': 'bleu', '2a': 'bleu', '4b': 'vert', '3b': 'vert',
'2b': 'vert', '1b': 'vert', '1a': 'bleu'}
```

[18]: True

[19]:
```
est_biparti(Dict_Chomp,((2,3,'J1')))
```

```
{(2, 3, 'J1'): 'bleu', (2, 2, 'J2'): 'vert', (2, 1, 'J2'): 'vert', (1, 3, 'J2'):
'vert', (2, 1, 'J1'): 'bleu', (1, 2, 'J1'): 'bleu', (1, 1, 'J1'): 'bleu', (1, 1,
'J2'): 'vert'}
```

[19]: True

Essayons avec un autre graphe, non biparti :

[20]:
```
est_biparti(Dict_Nim, '8')
```

[20]: False

Remarque : ces algorithmes peuvent reposer peut utiliser n'importe quelle fonction méthode parcours, pas seulement le BFS.

2.5 Stratégie et positions gagnantes

Afin de gagner presque *à coup sur* les joueurs auront intérêt à définir une *méthode* pour déterminer quels coups jouer au fur et à mesure de la partie : c'est l'idée d'une **stratégie**. Une telle **stratégie** détermine de façon univoque et répétable le mouvement suivant à jouer, à partir d'un état ou position.

Une **stratégie** est dite **gagnante** pour un joueur si toute partie jouée en suivant cette stratégie est gagnante, à partir d'une position de départ donnée, quelquesoit la stratégie du joueur adverse.

Une **position** est dite **gagnante** s'il existe une stratégie gagnante à partir de cette position.

Question 7 : Pour le jeu de Nim, déterminer les positions gagnantes pour le joueur 1. En déduire une stratégie gagnante à partir du sommet 8a. En généralisant pour n bâtonnets, donner une indication sur des positions perdantes.

Pour le joueur 1, l'état final 1b est gagnant, et il a pour antécédents 2a, 3a et 4a : ce sont donc aussi des positions gagnantes pour lui. Comme 5b n'est relié qu'à ces trois états, c'est aussi une position gagnante pour le joueur 1. Or le joueur 1 peut atteindre directement 5b à partir de 8a :

8a est donc une position gagnante, et il existe une stratégie gagnante consistant à choisir de passer par 5b puis d'agir "intelligemment" en fonction de ce que joue l'adversaire.

En itérant ce raisonnement, on peut montrer que les positions avec $4n + 1$ bâtonnets ($n \in N$) sont perdantes pour le joueur qui doit jouer. Ainsi si le nombre de bâtonnets au départ est de la forme $4n$, $4n + 2$ ou $4n + 3$, alors le joueur qui commence possède une stratégie gagnante (c'est une position gagnante pour lui); s'il est de la forme $4n + 1$ alors c'est l'autre joueur qui aura une stratégie gagnante. En particulier, à Fort Boyard, la partie débute avec 20 bâtonnets et $20 = 4 \times 5$ donc le joueur qui commence devrait toujours gagner !

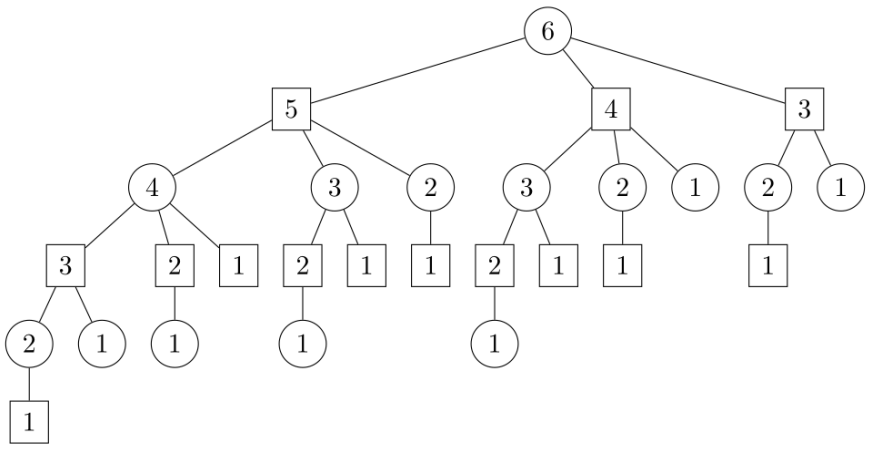
3 Notion d'heuristique

3.1 Complément sur la théorie des jeux : arbre de décision et heuristique

La détermination de stratégies gagnantes repose souvent sur la détermination de positions gagnantes, en partant de l'état final gagnant. Mais cette méthode se heurte généralement à la complexité des jeux :

- pour le jeu du morpion, il y a 765 positions possibles,
- pour le jeu du puissance 4, il y a environ $4,5 \times 10^{12}$ positions possibles
- pour les échecs, il y a de l'ordre de 10^{120} parties possibles
- enfin pour le jeu de go, il y a de l'ordre de 10^{170} positions possibles, ce qui donne approximativement 10^{600} parties possibles.

Les parties possibles peuvent être représentées par des **arbres de décision**; comme celui ci-dessous, pour l'exemple du jeu de Nim avec 6 bâtonnets au départ :



Chaque sommet est une position du jeu (ici sous forme de cercle ou carré selon le joueur dont c'est le tour), et les arrêtes les coups possibles. Le déroulement de la partie se lit de haut en bas. Les positions finales, les plus "basses", sont nommées *feuilles* de l'arbre, et leur nombre représente le nombre de parties possibles.

Sur un jeu complexe, plutôt que de chercher une stratégie gagnante (car parcours de l'arbre entier impossible), on peut proposer une **heuristique** : il s'agit d'évaluer quelles sont les positions les

plus *avantageuses* pour le joueur considéré, à l'aide du fonction *simple*, afin de faire un *bon choix*. Il ne s'agit pas forcément du *meilleur* choix, car on ne parcourt pas toutes les parties pour le prouver !

Généralement, une **heuristique** affecte à chaque position une valeur numérique, d'autant plus grande que cette position est estimée meilleur pour le joueur considéré. Par exemple, pour les échec, une heuristique simple attribue à chaque position un nombre entier, somme de la "valeur estimée" des pièces sur un échiquier. Classiquement 1 pour un pion, 3 pour un cavalier ou un fou, 5 pour une tour et 9 pour une dame. Cependant ce n'est pas suffisant en pratique, il faut ajouter des considérations d'occupation de l'espace, de contrôle de cases, etc. La création d'une heuristique de qualité est une tâche difficile). Pour le puissance 4, une heuristique simple consiste à attribuer à chaque case une valeur, par exemple le nombre d'alignements potentiels de quatre pions lorsqu'on place un pion à cet emplacement (visible sur la figure ci-dessous), puis à sommer les cases occupées.

3	4	5	7	5	4	3
4	6	8	10	8	6	4
5	8	11	13	11	8	5
5	8	11	13	11	8	5
4	6	8	10	8	6	4
3	4	5	7	5	4	3

De nombreux algorithme on alors été développés pour faire la meilleure décision possible à partir de ces heuristiques.

3.2 Exemple d'heuristique : problème du sac à dos

Le problème du sac à dos est un problème classique d'optimisation avec contrainte. Voici son énoncé : *On dispose d'un sac pouvant supporter un poids maximal donné et de divers objets ayant chacun une valeur et un poids. Il s'agit de choisir les objets à emporter dans le sac afin d'obtenir la valeur totale la plus grande tout en respectant la contrainte du poids maximal.*

Ce problème peut se résoudre par force brute, c'est-à-dire en testant tous les cas possibles. Mais ce type de résolution présente un problème d'efficacité. Son coût en fonction du nombre d'objets disponibles croît de manière exponentielle. Nous pouvons envisager une stratégie gloutonne. Le principe d'un algorithme glouton est de faire le meilleur choix pour prendre le premier objet, puis le meilleur choix pour prendre le deuxième, et ainsi de suite. Que faut-il entendre par meilleur choix ? Est-ce prendre l'objet qui a la plus grande valeur, l'objet qui a le plus petit poids, l'objet qui a le rapport valeur/poids le plus grand ? Cela reste à définir;

Un algorithme glouton va, **à chaque étape**, donner une solution optimale. Par exemple, pour le problème du sac-à-dos, il va proposer, à chaque étape, de mettre dans le sac-à-dos l'objet de plus grande valeur encore disponible ne faisant pas dépasser le poids maximal. C'est effectivement la meilleure solution, mais uniquement si on s'arrêtait là ! Un algorithme glouton n'assure donc pas une convergence vers un optimum global, soit vers la *meilleure solution du problème* : il s'agit d'une **heuristique**.

Pour traiter ce problème du sac-à-dos, voir l'énoncé CCINP TSI 2021 (ou le sujet 0).