Gendarmes et voleurs

1 Introduction

Le jeu des gendarmes et du voleur est un jeu à deux joueurs qui se joue sur un graphe non orienté. On déplace sur ce graphe un ou plusieurs pions représentant des gendarmes, et un pion représentant un voleur. Un joueur contrôle les gendarmes, l'autre joueur contrôle le voleur.

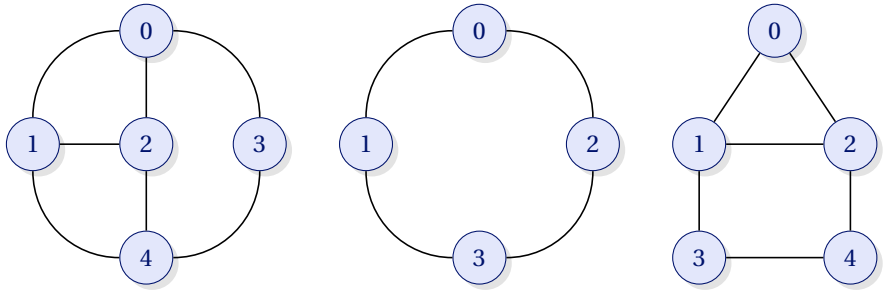
Initialement, le joueur contrôlant le(s) gendarme(s) décide du placement de son ou ses pions, puis l'autre joueur place son pion. Ensuite, les joueurs jouent tour à tour, le joueur contrôlant les gendarmes jouant en premier. Chacun déplace son (ou l'intégralité de ses) pion(s) le long d'une des arêtes du graphe. Le joueur contrôlant les gendarmes gagne s'il parvient à placer un quelconque de ses pions sur le sommet occupé par le voleur. Le voleur gagne s'il échappe indéfiniment aux gendarmes.

Un graphe est dit *cop-win* pour un nombre de gendarmes donné si le joueur jouant le(s) gendarme(s) dispose d'une stratégie gagnante. Le *cop-number* est le nombre minimal de gendarmes nécessaires pour qu'un graphe donné soit cop-win.

2 Construction des positions gagnantes

On se place tout d'abord dans le cas d'un jeu avec un unique gendarme. On décrit les graphes par des listes de listes d'adjacence. En particulier, les graphes ci-dessous sont décrits par les listes suivantes :

```
G1 = [[1, 2, 3], [0, 2, 4], [0, 1, 4], [0, 4], [1, 2, 3]]
G2 = [[1, 2], [0, 3], [0, 3], [1, 2]]
G3 = [[1, 2], [0, 2, 3], [0, 1, 4], [1, 4], [2, 3]]
```



On décrit l'état du jeu par un triplet de deux entiers i et j et un booléen. i désigne le sommet occupé par le gendarme, j le sommet occupé par le voleur, et le booléen vaut `True` si et seulement si c'est au tour du joueur contrôlant le gendarme de déplacer son pion. Par exemple, `(1, 3, False)` signifie que le gendarme occupe le sommet 1, le voleur le sommet 3 et que c'est le voleur qui doit se déplacer.

1. Proposer une fonction `suivant(G, pos)` qui prend en argument un tel triplet et renvoie la liste des triplets que l'on peut obtenir si un coup est joué (dans un ordre quelconque). Par exemple, `suivant(G, (1, 3, False))` devra renvoyer `[(1, 0, True), (1, 4, True)]`.

2. Écrire une fonction `victoire(pos)` qui prend en argument un triplet et renvoie un booléen indiquant si le gendarme a attrapé le voleur.

On souhaite construire la liste des positions pour lesquelles le joueur jouant le gendarme a une stratégie gagnante. Pour ce faire, on rappelle

- toutes les positions pour lequelles `victoire` renvoie `True` sont gagnantes ;
- si c'est au tour du gendarme de jouer, la position est gagnante s'il existe au moins un coup menant à une position gagnante ;
- si c'est au tour du voleur de jouer, la position est gagnante si la *totalité* des coups menent à une position gagnante.

On souhaite donc construire un dictionnaire `gagnantes` telle que `pos` soit une clé si et seulement si l'on a déterminé que la position `pos` est gagnante. La valeur associée à cette clé n'a ici pas d'importance [1], on utilisera `True`. Initialement, le dictionnaire est vide.

3. Proposer une fonction `sait_gagnante(G, suivant, victoire, pos, gagnantes)` prenant en argument la fonction déterminant les coups possibles et celle déterminant le gain [2], une position et le dictionnaire et renvoyant `True` si l'on peut affirmer qu'il existe une stratégie gagnante depuis cette position, et `False` sinon. **On peut répondre directement si `pos` figure parmi les clés du dictionnaire, sinon on essaiera de déterminer son état avec les règles énoncées ci-dessus (avec l'aide de `capture` et `suivant`), une position étant considérée gagnante si elle est déjà présente dans le dictionnaire**. La fonction `sait_gagnante` ne modifie pas le dictionnaire.

4. À l'aide de la fonction précédente, en déduire une fonction `maj_gagnante(G, suivant, victoire, pos, gagnantes)` qui ne fait rien si `pos` est un clé du dictionnaire ou si l'on ne peut pas déterminer que `pos` est gagnante pour le moment. Si elle le peut avec les règles précédente, on ajoutera `pos` comme clé du dictionnaire, avec comme valeur associée `True`.

Il reste à présent à appeler la fonction suffisamment de fois pour déterminer toutes les positions gagnantes. Dans un premier temps, on va utiliser une méthode simple : on va appeler la fonction `maj_gagnantes` sur l'*ensemble* des $2n^2$ états du jeu possibles (les triplets `(i, j, b)` où i et j sont des entiers entre `0` et `n-1` et `b` un booléen). Si à l'issue de

1. On pourrait en profiter pour mémoriser un coup gagnant identifié, de façon à calculer en même temps une stratégie !
2. Nous verrons plus tard pourquoi nous les passons en argument !

ces $2n^2$ appels le dictionnaire possède plus de clés qu'avant, on recommence avec les $2n^2$ états. Sinon, on s'arrête (on admettra que les états du jeu absents du dictionnaire sont des situations gagnantes pour le voleur).

5. Proposer une fonction `resout(G)` qui construit et renvoie un dictionnaire des positions gagnantes.

6. À partir du dictionnaire ainsi construit, comment déterminer si le joueur jouant le gendarme a une stratégie gagnante ? On rappelle qu'il choisit en premier où placer son pion, que le voleur choisit en second (un sommet différent) puis que le gendarme se déplace en premier.

7. En déduire une fonction `copwin(G)` prenant en argument un graphe et renvoyant une liste(possiblement vide) des positions de départ pouvant être choisies par le gendarme et lui garantissant une stratégie gagnante pour ce graphe.

8. Essayer avec les graphes G_1, G_2 et G_3.

On fournit ci-dessous une fonction construisant une grille de taille $n \times p$ où chaque case est reliée à ses quatres voisines (si elles existent) :

```
def grid(n, p):
    G = [[] for i in range(n*p)]
    for i in range(n):
        for j in range(p):
            if i>0: G[p*i+j].append(p*i+j-p)
            if i<n-1: G[p*i+j].append(p*i+j+p)
            if j>0: G[p*i+j].append(p*i+j-1)
            if j<p-1: G[p*i+j].append(p*i+j+1)
    return G
```

9. Vérifier sur quelques exemples que pour tout $n > 1$ et $p > 1$, le joueur déplaçant le gendarme n'a pas de stratégie gagnante.

3 Cas de deux gendarmes

Nous allons maintenant supposer que le joueur déplaçant les gendarmes dispose de deux pions. Une position du jeu sera dorénavant décrite par un quadruplet `(g1, g2, v, b)` où `g1` et `g2` sont les positions des gendarmes, `v` celle du voleur, et `b` un booléen indiquant si c'est aux gendarmes de se déplacer.

10. Proposer une fonction `suivant2(G, pos)` et une fonction `capture2(pos)` adaptées aux nouvelles règles.

On remarque que `sait_gagnante` et `maj_gagnantes` n'ont pas besoin d'être modifiées, il faudra juste les appeler avec les deux fonctions précédentes.

11. Proposer des fonctions `resout2(G)` et `copwin2(G)` adaptées au cas de deux gendarmes.

12. Les gendarmes ont-ils une stratégie gagnante s'ils sont deux sur une grille $n \times p$ avec $n > 1$ et $p > 1$?

4 Graphes aléatoires

On peut se demander à quel point il est probable qu'un graphe donné dispose d'une stratégie gagnante pour les gendarmes. On propose la fonction suivante qui construit un graphe connexe à n sommets où l'on place une arête entre chaque paire de sommets avec une probabilité `prob` :

```
def random_graph(n, prob):
    from random import random
    G = [[] for i in range(n)]
    for i in range(n):
        for j in range(i+1, n):
            if random()<prob:
                G[i].append(j)
                G[j].append(i)
    return G
```

13. En faisant une centaine d'essais, déterminer la probabilité qu'un graphe à 20 sommets possède une stratégie gagnante pour les gendarmes pour `prob`= 0.1. Faire de même pour 0.05 et 0.2.