Chapitre 8

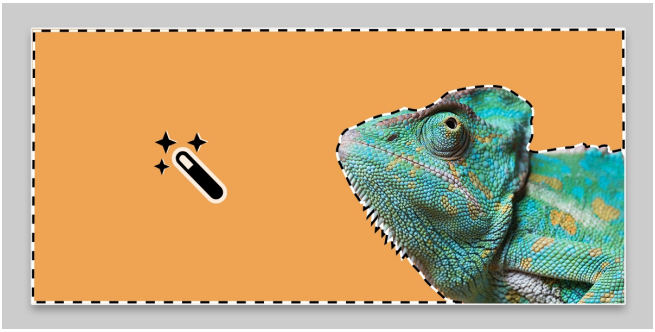
GRAPHES ET THÉORIE DES JEUX

Le but de cette séance d'introduction à la théorie des jeux est de réviser les graphes vus en PCSI, leurs différentes modélisations en fonction des contextes, ce que représentent les sommets et les arêtes mais aussi les parcours (en profondeur et en largeur) des graphes.

Les graphes sont l'outil indispensable pour la modélisation des jeux finis à deux joueurs au programme de PSI, afin d'établir des stratégies permettant la maximisation des chances de victoire.

Le premier exercice modélisera le jeu de Nim (ou jeu des bâtonnets) et le second exercice permettra de recréer (de façon naïve) l'outil de sélection « baguette magique » sous Photoshop.

Le jeu des bâtonnets (Jeu de Nim)



Outil « Baguette magique » sous Photoshop

Table des matières

I LE JEU DE NIM

Présentation du jeu de Nim

Dans l'émission Fort Boyard, le **duel de bâtonnets** (aussi appelé historiquement **jeu de Nim**) avec les maîtres du temps est un jeu à deux joueurs. Au départ, un tas comporte 20 bâtonnets, et à son tour un joueur peut en enlever 1, 2 ou 3. Le perdant est celui qui prend le dernier bâtonnet. Voici un exemple de partie où ni le maître du jeu, ni la joueuse n'ont étudié la théorie des jeux : `https://www.youtube.com/watch?v=10CUpulWxww`.

On peut modéliser ainsi un exemple de partie entre deux joueurs a et b :

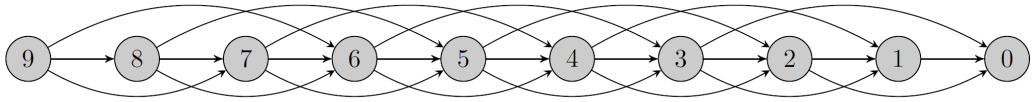
$$20 \xrightarrow{a} 17 \xrightarrow{b} 16 \xrightarrow{a} 15 \xrightarrow{b} 12 \xrightarrow{a} 10 \xrightarrow{b} 8 \xrightarrow{a} 7 \xrightarrow{b} 4 \xrightarrow{a} 3 \xrightarrow{b} 1 \xrightarrow{a} 0$$

Modélisation par un graphe simple

À un tel jeu, on peut associer un **graphe orienté** $G = (S, A)$ qui permet de modéliser toute partie envisageable :

- les **sommets** S de G sont les positions atteignables dans le jeu. Pour le jeu précédent, on pourrait utiliser 21 sommets numérotés de 0 à 20, indiquant le nombre de bâtonnets restants.
- les **arcs** (arêtes orientées) A de G indiquent quel sommet est atteignable depuis un autre en jouant un seul coup.

Exemple. Les arcs issus du sommet 7 pointent vers 6, 5 et 4, comme on le voit sur la figure ci-dessous :



Un jeu de Nim avec seulement 9 bâtonnets au départ

Une partie sur un tel graphe est un chemin où le premier joueur, depuis le sommet de départ, suit un arc, et ensuite chaque joueur suit à tour de rôle un arc, si c'est possible.

Remarque. Le problème sur un tel graphe est qu'il faut noter quel joueur est en train de jouer...

Une partie est *finie* si ce chemin termine sur un sommet sans successeur/voisin, *infinie* sinon.

1. Écrire une fonction `GrapheNim(N:int)->{int:list}` qui prend en entrée un nombre `N` de bâtonnets et renvoie le dictionnaire d'adjacence du graphe `G` (clefs = sommets et valeurs = liste des sommets voisins).
2. Écrire des commandes permettant de compter le nombre d'arcs de votre graphe `G` à partir de son dictionnaire d'adjacence. On vérifiera sur plusieurs valeurs de `N` que l'on trouve bien $3\texttt{N} - 3$.
3. Expliquer pourquoi un dictionnaire d'adjacence est plus adapté ici qu'une matrice d'adjacence.
4. Quel type de chemin ne doit surtout pas comporter ce graphe pour être certain que toute partie est finie ?

Stratégie gagnante

Ce jeu est particulièrement simple, et il est facile de voir que l'un des deux joueurs peut toujours gagner s'il suit la bonne stratégie ! (on dit que ce joueur possède alors une **stratégie gagnante**.)

5. Si le joueur a vient de jouer et a laissé seulement 5 bâtonnets au joueur b, expliquer pourquoi le joueur a peut gagner dans tous les cas.
6. Dans le cas général, à chaque tour, combien de bâtonnets le joueur a doit-il laisser au joueur b de sorte à pouvoir toujours être certain de gagner la partie ?
7. Quel joueur possède une stratégie gagnante ? Celui qui commence ou celui qui joue en second ?
   Une **stratégie gagnante** pour un joueur est le fait d'être assuré de gagner toute partie jouée à partir d'une position initiale en jouant intelligemment selon un schéma de jeu bien précis (une **stratégie**).

Propriété (Théorème de Zermelo)**.**

Dans tout jeu tour à tour, fini, à deux joueurs, à information parfaite, et sans hasard, sans match nul, l'un des deux joueurs a une stratégie gagnante.

Démonstration : Par récurrence sur le nombre de sommets du graphe.

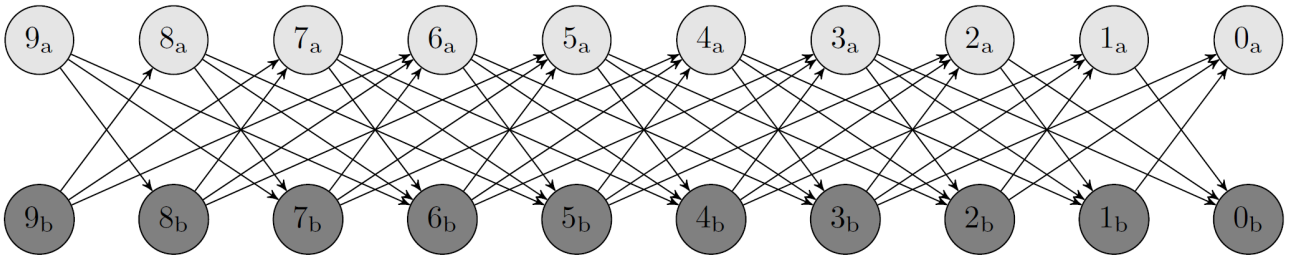
Modélisation par un graphe biparti

Il est utile dans la pratique de « dédoubler » un graphe comme le précédent en un graphe biparti, de sorte qu'une partie dans le jeu se résume simplement à un chemin dans le graphe biparti, sans avoir à distinguer quel joueur joue.

Définition.

Un graphe $G = (S, A)$ est dit **biparti** s'il existe une partition de S en deux sous-ensembles S_a et S_b, telle qu'aucun arc du graphe ne relie deux sommets de S_a, ou deux sommets de S_b.

Exemple. Le graphe biparti associé au graphe de la figure précédente (9 bâtonnets) est le suivant :



Le graphe biparti associé au jeu de Nim avec 9 bâtonnets au départ.

Remarque. Si le joueur a commence (en 9_a) alors le sommet 9_b ne sera jamais atteint.

8. Soit `N` le nombre de bâtonnets, quel est maintenant le nombre de sommets du graphe ? Et son nombre d'arcs ?
9. Modifier la fonction `GrapheNim` en une fonction `GrapheNim2(N:int,j1:str,j2:str)->{(int,str):list}` afin qu'elle renvoie le dictionnaire d'adjacence du graphe biparti `G` codant le jeu de Nim entre deux joueurs `j1` et `j2`.
   Les clefs sont les sommets, c'est-à-dire un couple (numéro, nom de joueur) et la valeur associé est toujours la liste des sommets voisins.

Remarque. On pourra utiliser un dictionnaire des opposants `opp = {j1:j2,j2:j1}`.

II BAGUETTE MAGIQUE SOUS PHOTOSHOP

Vous travaillerez dans le fichier `MagicWand.py` fourni.

- Documentez vos fonctions.
- Testez au fur et à mesure les fonctions que vous écrivez ! Pour ce faire, des images sont proposées, mais vous pouvez utiliser tout autre image en couleurs de votre choix.

Une image `img` est codée sous la forme d'une liste de listes de listes à trois éléments d'entiers, qui peut être vue comme un tableau à deux dimensions de triplets.

On notera h le nombre de lignes de l'image, et w son nombre de colonnes (respectivement pour **h**eight et **w**idth). Le pixel d'indice $(0, 0)$ correspond par convention au pixel situé en haut à gauche de l'image.

Chaque pixel de coordonnées $(i, j) \in [\![0, h-1]\!] \times [\![w-1]\!]$ a une couleur codée par une liste d'entiers `[r,g,b]` de trois entiers compris entre 0 et 255, correspondant respectivement à ses niveaux de rouge, vert et bleu (0 pour l'absence de couleur, 255 pour l'intensité maximale). On parle d'espace RGB pour l'ensemble de ces triplets.

Des fonctions permettant de charger et d'afficher une image sont déjà dans le fichier Python `MagicWand.py`.

Conversion en niveaux de gris

Un niveau de gris est obtenu lorsque les trois canaux r, g et b sont à la même valeur. Par exemple, $[0, 0, 0]$ correspond au noir, $[255, 255, 255]$ au blanc, et $[127, 127, 127]$ à un gris moyen.

1. Écrire une fonction `dim(img:[[[int]]])->tuple`, qui prend en argument une image et renvoie le couple d'entiers `(w,h)`, `w` étant la largeur de l'image, et `h` sa hauteur (en nombre de pixels).
   On pourra ainsi dans la suite utiliser `(w,h) = dim(img)` pour récupérer les dimensions d'une image.
2. Écrire une fonction `niveaux_de_gris(img:[[[int]]])->none` qui prend en entrée une image couleur et la transforme en image en niveaux de gris en utilisant la stratégie naïve [1] qui remplace $[r, g, b]$ par $[m, m, m]$, où $m = \left\lfloor \frac{r+g+b}{3} \right\rfloor$.

Deux outils de sélection

On souhaite dans cette partie écrire deux outils de sélection par couleur dans une image. La sélection sera matérialisée par la mise en rouge de chaque pixel sélectionné (il serait également possible, et guère plus compliqué, d'engendrer une nouvelle image en noir et blanc correspondant à un masque de sélection).

Sélection par couleur

Sélection par baguette magique.

Sélection par couleur

3. Écrire une fonction `sq_distance_couleurs(c1,c2)->int` qui prend en argument deux couleurs `c1=[r1,g1,b1]` et `c2=[r2,g2,b2]`, et renvoie le carré de la distance algébrique entre ces deux couleurs dans l'espace RGB [2].
4. Écrire une première fonction `selection_couleur(img:[[[int]]],i:int,j:int,seuil:int)->none` qui prend en argument une image, les coordonnées d'un pixel, et un seuil, et sélectionne (i.e. colorie en rouge) tous les pixels de l'image dont la distance à la couleur du pixel de coordonnées (i, j) est inférieure au seuil. Quelle est sa complexité ?

1. Naïve car l'œil humain n'a pas la même sensibilité aux différents canaux, et que sa sensibilité n'est de plus pas linéaire par rapport aux luminosités des canaux. Cf `https://fr.wikipedia.org/wiki/Niveau_de_gris`
2. Même remarque que ci-dessus, cela n'est pas la métrique la plus pertinente à mettre sur les couleurs...

Sélection par baguette magique

Une seconde méthode, dont sont dérivées les « baguettes magiques » de certains logiciels d'édition d'image, consiste à sélectionner les pixels ayant une couleur proche d'un pixel de départ, mais de façon à créer une zone de pixels **contigus** la plus grande possible vérifiant cette propriété.

Pour cela, on va considérer les pixels comme les sommets d'un graphe. Une arête reliera deux sommets si les pixels sont voisins [3] dans l'image.

L'algorithme consiste alors à effectuer un parcours du graphe à partir du pixel sélectionné, de façon à ce que seules les zones de l'image vérifiant la condition de couleur soient parcourues.

5. Écrire une fonction `voisins(i,j,h,w)` qui prend en argument les coordonnées `i` et `j` d'un pixel dans une image de dimensions `(h,w)`, et renvoie la liste de ses pixels voisins (attention aux bords !).
6. Écrire une fonction `baguette_magique(img,i,j,seuil)` qui implémente la sélection présentée ci-dessus. On pourra utiliser une liste `a_explorer`, contenant initialement le seul pixel initial. Tant que cette liste n'est pas vide, on extraira le dernier élément : un pixel `p` que l'on coloriera en rouge, et on lui rajoutera les voisins de `p` non déjà explorés ou coloriés en rouge et ayant une couleur convenable.
   Quelle est sa complexité ?
   Rappel : On pourra utiliser la commande `L.pop()` qui supprime le dernier élément d'une liste `L`.
   Cette commande renvoie cet élément et a une complexité linéaire en $\mathcal{O}(1)$.
   A contrario `L.pop(i)` supprime et renvoie le `i`-ième élément de `L` et a une complexité linéaire en $\mathcal{O}(\texttt{len(L)})$.
7. Quel algorithme de parcours de graphe a-t-on utilisé ici ? Remplacer la liste `a_explorer` par une **pile**.

Rappel : Utilisation des piles et files

	Files	**Piles**
Importer le module	`from collections import deque`	`from collections import deque`
Créer une file/pile	`file=deque(['A','B','D'])`	`pile=deque(['A','B','D'])`
Enfiler/Empiler	`file.append('F')`	`pile.append('F')`
Défiler le premier arrivé	`file.popleft()`	*Théoriquement pas possible !*
Dépiler le premier arrivé	*Théoriquement pas possible !*	`pile.pop()`

• L'essentiel est de comprendre que la variable `file` créé avec la commande `deque` se gère comme une liste sauf que la commande `file.popleft()` a une complexité en $\mathcal{O}(1)$ tandis que celle de `L.pop(0)` est en $\mathcal{O}(\texttt{len(L)})$.

8. Quel autre algorithme de parcours serait envisageable ? Comment faudrait-il modifier la fonction précédente pour l'implémenter ? Vaut-il mieux alors utiliser des piles ou des files ? Qu'est-ce que cela changerait sur le résultat obtenu ?

III RETOUR SUR LE JEU DE NIM

Suite du **I** sur le jeu de Nim.

10. Coder la création du graphe du jeu de Nim avec un dictionnaire d'adjacence en utilisant un parcours **récursif**, puis avec un parcours en largeur avec une file, puis avec un parcours en profondeur avec une pile.

Remarque. Voici quelques indications pour le code (en récursif notamment) :

- en partant d'un sommet de la forme `(n,j)` on va aller visiter les trois successeurs de ce sommet : `(n-1,o)`, `(n-2,o)`, `(n-3,o)` où `o` est le nom de l'opposant du joueur `j`.
- attention, il faut garder en tête que le nombre de bâtonnets doit toujours être supérieur ou égal à 1.

3. Il y a au plus quatre voisins par pixel. Deux pixels situés en diagonale l'un par rapport à l'autre ne seront pas considérés comme voisins.