Chapitre 10

THÉORIE DES JEUX (PARTIE 2)

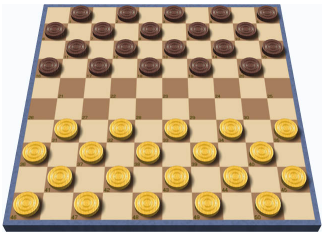
Dans cette seconde et dernière séance sur la théorie des jeux, le but va être double :

- *comprendre la notion d'**heuristique**, déjà rencontrée lors des parcours de graphe avec l'algorithme $A^\star$, et ce qu'elle représente pour la théorie des jeux ;*
- *Savoir programmer l'**algorithme min-max** qui a comme but de maximiser l'heuristique.*

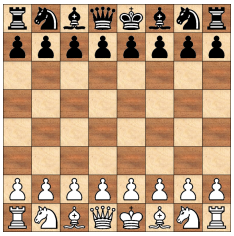
Nous allons ici essayer de trouver un algorithme qui permet de maximiser les gains dans un jeu d'accessibilité sans avoir à parcourir tout le graphe comme pour le calcul des attracteurs.

Dans le cas d'un jeu à deux joueurs plus complexe, le calcul des attracteurs n'est pas possible. L'algorithme écrit dans la première séance a une complexité en $\mathcal{O}(n^3)$, où n est le nombre de sommets du graphe, autrement dit le nombre de positions que l'on peut rencontrer lors d'une partie.

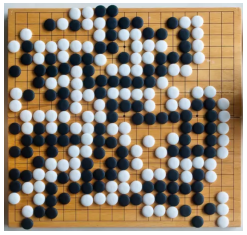
Cependant, cet entier n est souvent extrêmement grand : il est estimé de l'ordre de 10^{32} pour les dames, entre 10^{43} et 10^{50} pour le jeu d'échecs, de l'ordre de 10^{100} pour le jeu de go. Il devient donc nécessaire de s'appuyer non plus sur une évaluation exacte de la position, mais sur une estimation de la valeur de la position atteinte.



Jeu de dames



Jeu d'échecs



Jeu de go

Table des matières

10 THÉORIE DES JEUX (PARTIE 2) 1

I HEURISTIQUE 2

II MAXIMISER L'HEURISTIQUE : L'ALGORITHME MIN-MAX 3

III EXERCICE : LE JEU DE CHOMP (SUITE) 7

I HEURISTIQUE

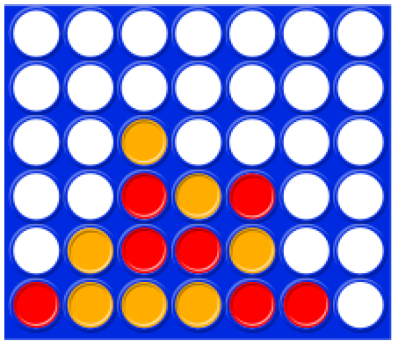
Considérons un jeu d'accessibilité et cherchons à établir une stratégie « intelligente », c'est à dire une stratégie qui maximise les chances de victoire, c'est à dire qu'étant donnée une position dans le jeu, le joueur choisit un coup possible qui lui semble le meilleur parmi tous les choix possibles qui s'offrent à lui.

Une stratégie intelligente : elle doit, en particulier, être meilleure que la stratégie au hasard, autrement dit sur un grand nombre de parties, la stratégie intelligente doit battre très souvent la stratégie aléatoire. Pour trouver une telle stratégie, l'idée est, étant donné une position du jeu, d'estimer la « valeur » de tel ou tel coup, ou plutôt la valeur de la variation entre deux positions de jeu.

Une méthode consiste à attribuer à une position de jeu un nombre de sorte que plus le nombre est grand plus on est susceptible de gagner. On cherche donc une fonction notée h qui à chaque position du graphe/de l'arène attribue un nombre. Par convention, si on est à une position gagnante, on pose $h(p) = +\infty$ et si on est à une position perdante, on pose $h(p) = -\infty$

Exemple.

- Aux échecs, on peut attribuer à chaque pièce une valeur (classiquement 1 par pion, 3 par fou et cavalier, 5 par tour, 9 par dame et 0 pour le roi), et faire la différence entre ses points et ceux de l'adversaire. A priori, plus vous avez un score élevé et plus vous êtes susceptibles de gagner puisque vous avez plus de possibilité de mouvement et de prise de pièces adversaires.
- Au puissance 4, on peut attribuer une valeur à chaque case, par exemple le nombre d'alignements potentiels de quatre pions lorsqu'on place un pion à cet emplacement. Puis on somme les cases occupées (positivement pour les pions jaunes, négativement pour les pions rouges).



Exemple de position de puissance 4

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

Valeur des cases pour l'heuristique

Par exemple, si on convient que les pions jaunes sont ceux du joueur auquel on s'intéresse, la valeur de cette position de jeu est égale à $h = 4 + 5 + 7 + 6 + 8 + 13 + 11 - 3 - 5 - 4 - 8 - 10 - 11 - 11 = 2$.

Évidemment, ces exemples de fonctions ne reflètent que très partiellement la réalité. On peut gagner une partie d'échecs avec moins de pièces que son adversaire si celles-ci sont bien placées. De même, à puissance 4, on peut très bien gagner avec un jeton dans le coin tandis que l'adversaire, qui a joué au milieu, perd.

Nous ne cherchons à décrire que partiellement la réalité en faisant des choix sur ce que l'on privilégie et modélise. (Une description parfaite de la réalité reviendrait à étudier toutes les positions de jeu...)

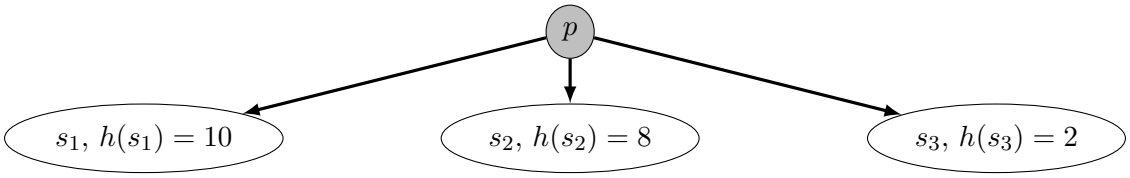
Pour cette raison, on dit que la fonction h s'appelle une **heuristique**, elle va coder notre compréhension très partielle du jeu, souvent issue de nos propres expériences. Autrement dit, c'est une façon de coder nos préjugés sur le jeu en question.

Faisons le lien avec l'heuristique avec l'algorithme $A^\star$ vu en PCSI : par rapport à l'algorithme de Dijkstra, en utilisant une heuristique (la distance à vol d'oiseau), on force l'algorithme à aller explorer certains sommets à priori plus intéressants que d'autres. De même ici : l'heuristique va nous pousser à privilégier certaines positions car la valeur/l'heuristique de ces sommets est plus « élevé ».

II MAXIMISER L'HEURISTIQUE : L'ALGORITHME MIN-MAX

Supposons que l'on soit dans un jeu d'accessibilité à deux joueurs (Morgan et Malo), à une position p, et qu'on dispose d'une heuristique h telle que plus h est grande, plus Morgan a des chances de gagner.

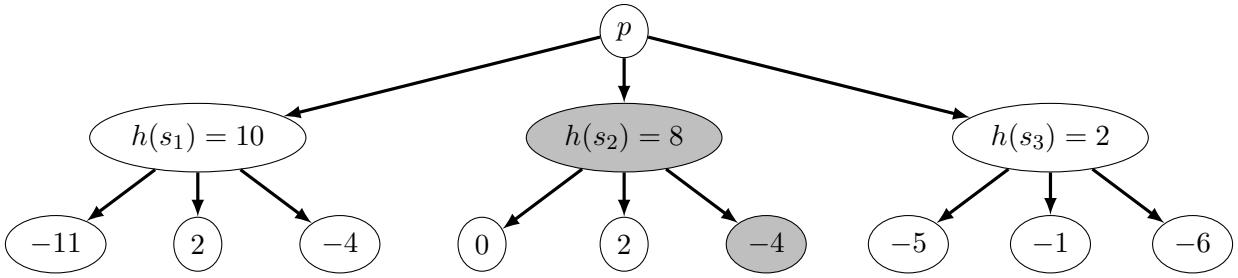
Au moment où l'un des deux joueurs doit jouer, plusieurs possibilités s'offrent à lui (entre une et sept pour le puissance 4). Une solution simple pour choisir le coup à jouer consiste à calculer l'heuristique correspondant à chacune des configurations atteignables, et à jouer le coup d'heuristique maximale (pour Morgan) ou minimale (pour Malo).



C'est à Morgan de jouer, trois coups sont possibles.

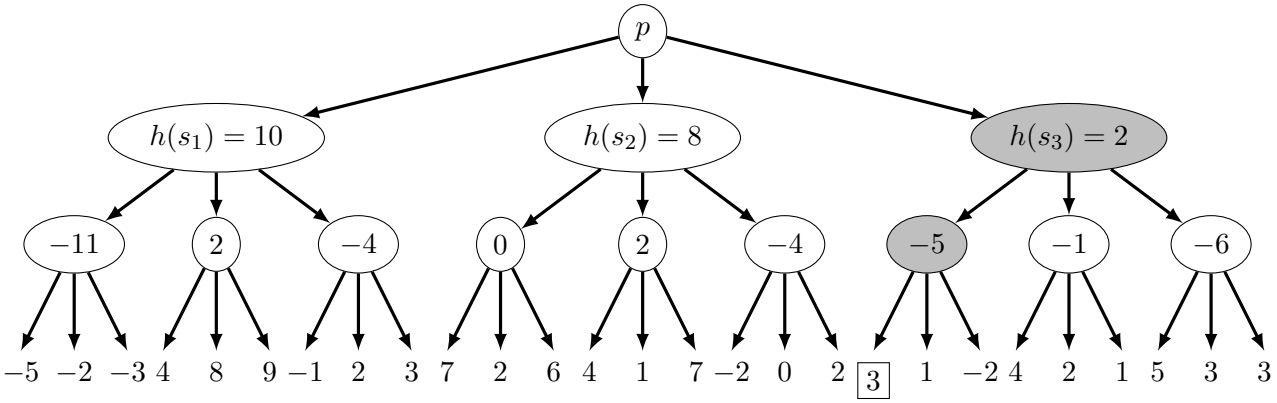
Ainsi, dans l'exemple précédent, Morgan choisirait le coup le plus à gauche, correspondant à une évaluation égale à 10, car c'est elle qui a la plus grande heuristique. Mais ce serait jouer de façon court-termiste.

Morgan peut aussi tenir compte du coup que va jouer Malo ensuite (peut-être qu'il va pouvoir jouer à partir de s_1 un coup qui fera diminuer drastiquement l'heuristique), et donc calculer l'heuristique de chacune des positions que Malo pourra atteindre (à partir du coup qu'il va jouer). Si on observe la figure ci-dessous, on constate qu'il vaut mieux pour Morgan jouer le coup central, en partant du principe que Malo joue au mieux son coup (Morgan doit donc choisir le coup qui permettra que l'heuristique de Malo soit maximal, parmi les choix qu'il peut avoir).



C'est à Morgan de jouer, mais on tient compte du meilleur coup suivant que jouera Malo.

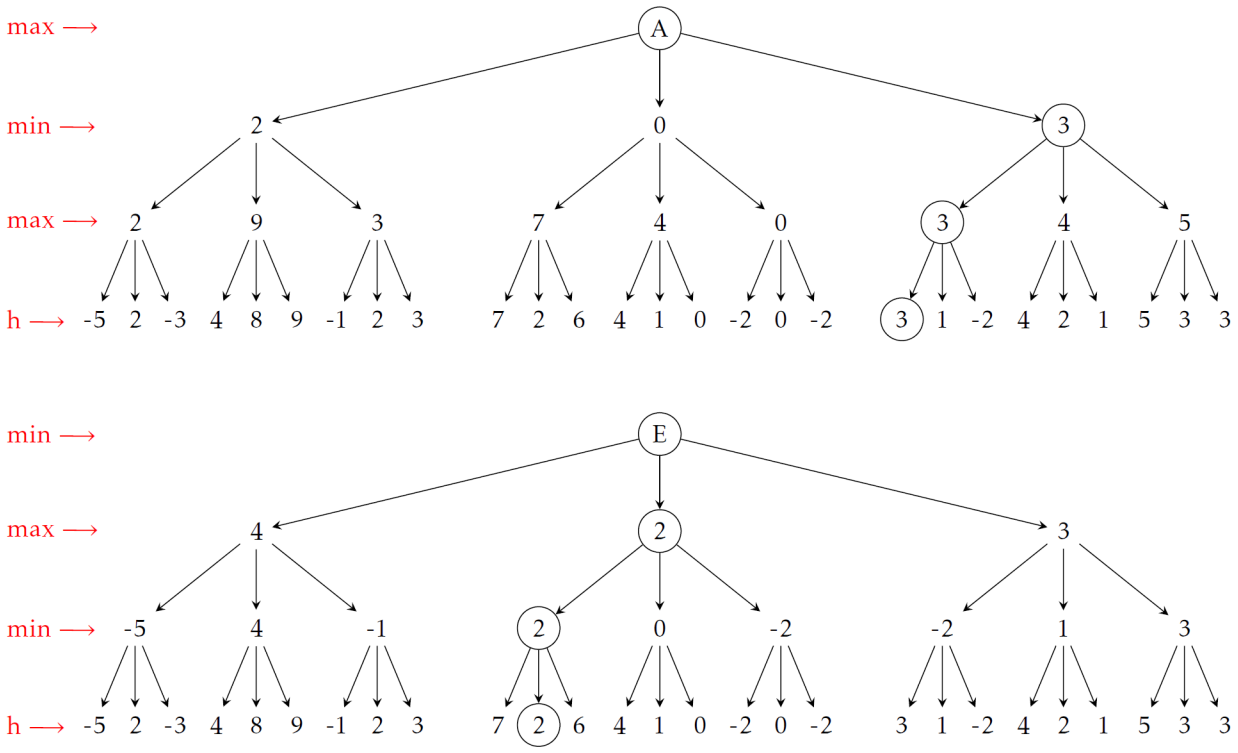
Bien évidemment, on peut réitérer ce raisonnement et tenir compte du coup suivant, joué cette fois par Morgan. La figure ci-dessous montre qu'en tenant compte des deux coups suivants, Morgan a en fait intérêt à jouer le coup le plus à droite.



C'est à Morgan de jouer, mais on tient compte des deux coups suivants.

Pour calculer le meilleur coup de Morgan en tenant compte des n coups suivants, il faut donc commencer par calculer la valeur de l'heuristique de toutes les positions atteignables en n coups (les feuilles de l'arbre/successeurs de la position initiale). Si n est pair, ce sera à Malo de jouer le dernier des n coups. Le père de chacune de ces feuilles se verra donc attribuer le minimum des valeurs de ses fils. À l'inverse, si n est impair le père de chacune de ces feuilles se verra attribuer la valeur maximale de ses fils (car Morgan aura joué en dernier).

Ainsi, de proche en proche chaque position de l'arbre se verra attribuer une valeur, qui ne sera pas forcément égale à sa propre heuristique, mais déduite des heuristiques des coups suivants (figure ci-dessous).



Résultats de l'algorithme min-max à une profondeur de 2.
(premier arbre si c'est à Morgan de jouer, le second si c'est à Malo.)

Nous allons écrire deux fonctions :

- `maximin(p,n)` (destinée à Morgan) va chercher à maximiser l'heuristique après `n` coups en partant de la position `p`, en supposant que son adversaire joue au mieux ;
- `minimax(p,n)` (destinée à Malo) va chercher à minimiser l'heuristique après `n` coups en partant de la position `p`, en supposant que son adversaire joue au mieux.

Ces deux fonctions sont **mutuellement récursives** : pour calculer `maximin(p,n)` on calcule pour chaque position `pi` atteignable en un coup à partir de `p` la valeur `vi` de `minimax(pi,n-1)` et on choisit la position `pi` conduisant à la valeur `vi` maximale.

De manière symétrique, pour calculer `minimax(p,n)` on calcule pour chaque position `pi` atteignable en un coup à partir de `p` la valeur `vi` de `maximin(pi,n-1)` et on choisit la position conduisant à la valeur `vi` minimale.

Lorsque la profondeur `n` vaut 0, alors on évalue la position directement grâce à l'heuristique `h`. Sans oublier les cas de fin de parties (victoires/défaites/nulles).

Pour la rédaction de l'algorithme, on suppose définies :

- la fonction `h(p)` qui prend pour argument une position du jeu et renvoie la valeur de son heuristique,
- la fonction `successeurs(p)` qui renvoie la liste des positions atteignables à partir de la position `p`.

On a alors toutes les cartes en main pour programmer l'algorithme `Min-Max` en Python :

```
def minimax(p,n):
  if n == 0 or successeurs(p) == []:
    return h(p)
  mini = np.inf
  for pk in successeurs(p):
    s = maximin(pk,n-1)
    if s < mini:
      mini = s
  return mini
```

```
def maximin(p,n):
  if n == 0 or successeurs(p) == []:
    return h(p)
  maxi = -np.inf
  for pk in successeurs(p):
    s = minimax(pk,n-1)
    if s > maxi:
      maxi = s
  return maxi
```

Remarque.
L'algorithme peut être long à exécuter pour deux raisons :

- suivant le type de jeu, il peut y avoir beaucoup de coups possibles à regarder à chaque étape.
- plus la profondeur `n` choisie est grande, plus la complexité sera importante (elle est même exponentielle).

La mémoïsation pourrait être utile car on peut aboutir à la même position avec des ordres de coups différents, mais cela complexifierait encore l'algorithme.

Remarque.
Il existe aussi d'autres méthodes qui n'explorent que certaines parties de l'arbre (élagage alpha-beta), mais celles-ci sont hors programme.

Remarque.
Si on ne fixe pas de profondeur `n` mais qu'on continue jusqu'à atteindre des positions sans successeur (fin de partie), et qu'on prend la fonction `h` sur ces positions terminales qui vaut 1 si Morgan a gagné la partie, -1 si Malo a gagné, et 0 si match nul, alors l'algorithme min-max nous donne un calcul exact des positions gagnantes et la stratégie gagnante correspondante. Cependant pour la plupart des jeux, le temps de calcul serait trop long.

Exemple.
Pour un jeu comme Puissance 4, pour explorer l'ensemble du jeu, comme une partie peut durer 42 coups, il faudrait une profondeur de 42, ce qui est beaucoup trop.

Le **choix de la profondeur** sur laquelle lancer l'algorithme min-max résulte d'un **compromis** entre qualité du résultat et temps de calcul : lorsque la profondeur augmente, la stratégie obtenue est meilleure mais le temps de calcul augmente (et peut devenir impraticable).

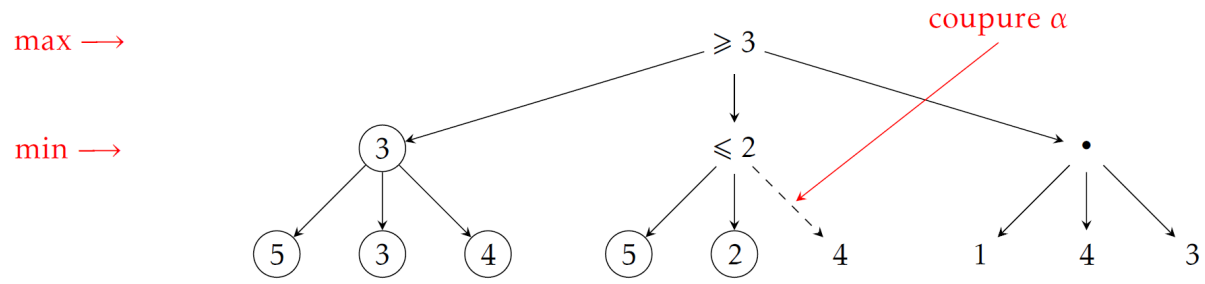
Enfin, une difficulté est de **trouver une heuristique** qui soit la plus fidèle possible à la probabilité de gagner (afin d'obtenir une bonne stratégie). Dans les faits, on peut faire s'affronter plusieurs heuristiques entre elles et prendre celle qui gagne le plus souvent contre les autres (les heuristiques utilisées en pratique sont d'ailleurs souvent des méthodes issues de l'expérimentation).

HORS-PROGRAMME : ÉLAGAGE ALPHA-BETA

L'élagage alpha-beta est une amélioration de l'algorithme min-max qui vise à ne pas explorer certaines des branches de l'arbre dont on sait qu'elle n'interviendront pas dans l'évaluation de la position courante.

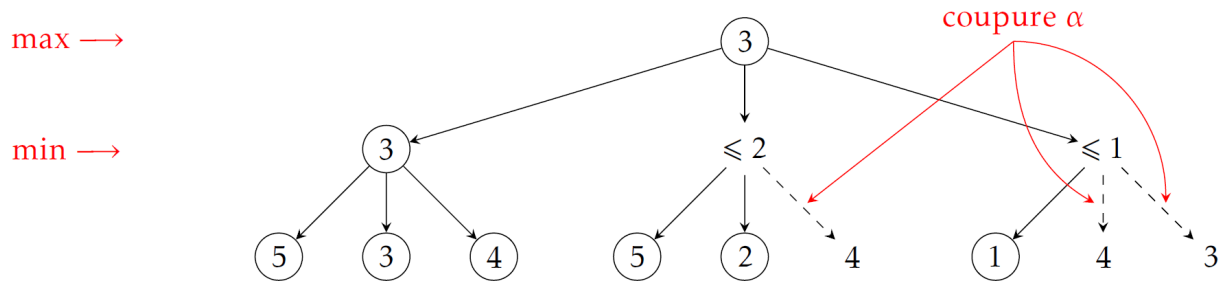
Coupure alpha

Considérons l'exemple ci-dessous, qui concerne une étape de calcul de minimum. L'exploration est en cours, les feuilles et nœuds qui ont été examinés sont marqués d'un cercle.



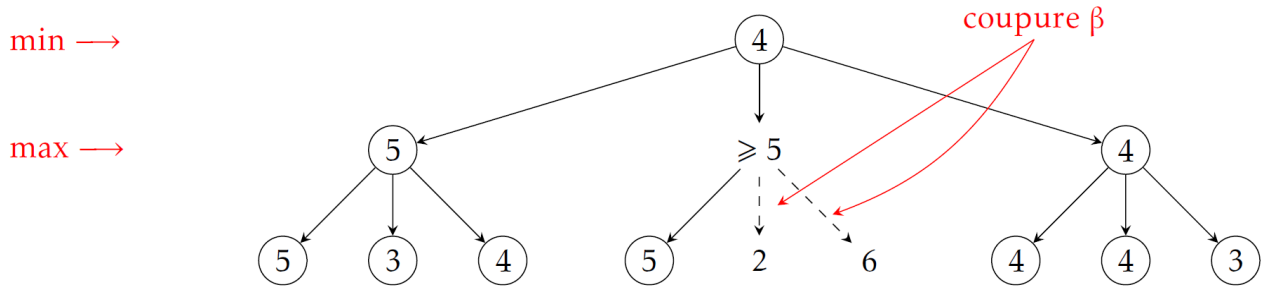
On constate qu'il n'est pas nécessaire de poursuivre l'exploration de la branche centrale : on sait déjà que sa valeur sera inférieure ou égale à 2 donc qu'elle ne sera pas choisie à l'étape suivante qui sera un calcul de maximum.

Une deuxième coupure α se produit après l'exploration du premier fils de la branche de droite : le minimum sera inférieur ou égal à 1 donc ne sera pas pris en compte pour le calcul du maximum de l'étape suivante :



Coupure beta

La situation est bien évidemment symétrique dans le cas d'une étape de calcul de maximum, comme l'illustre l'exemple ci-dessous :

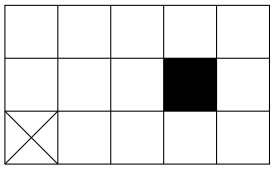


III EXERCICE : LE JEU DE CHOMP (SUITE)

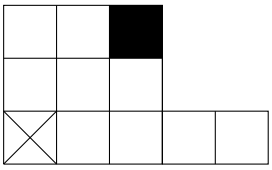
On reprend le jeu de Chomp là où nous nous étions arrêtés à la dernière séance. On rappelle qu'on dispose d'une tablette de chocolat avec `n` lignes et `p` colonnes et que le carré situé tout en bas et tout à gauche est empoisonné. Deux joueurs jouent à tour de rôle. À chaque coup, le joueur choisit l'un des carrés de la plaque et mange ce carré ainsi que ceux situés en haut et à droite de ce carré. Le jeu continue tant qu'il reste des carrés à manger. Celui qui a mange le carré empoisonné perd.

Par exemple, partons d'une tablette avec 3 lignes et 5 colonnes.

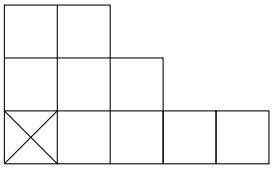
Morgan choisit la case à la 2-ième ligne 4-ième colonne.



Malo choisit alors la case 1ère ligne, 3-ième colonne



La partie continue... tant qu'il reste des carrés à manger...



On suppose que vous avez réussi à écrire les fonctions permettant de déterminer les liste des voisins/successeurs d'une position `s` donnée avec `ListeVoisins(s)` ainsi qu'une fonction `GrapheTab(tab,joueur)` qui permet de créer un graphe représentant le jeu de Chomp pour une tablette `tab` et un premier joueur nommé `joueur` donnés. Si ce n'est pas le cas, récupérez le fichier de correction de ces questions sur cahier de prépa.

Le but est d'une part d'écrire l'algorithme permettant le calcul des attracteurs ainsi qu'un algorithme permettant de définir une stratégie aléatoire, puis un algorithme de stratégie gagnante à l'aide des attracteurs.

Enfin, le but est de programmer la stratégie qui suit l'algorithme min-max.

8. Créer une fonction `Attracteur(joueur)` qui calcule l'attracteur (adapter l'algorithme du cours).
   Vérifier que chaque joueur admet 33 sommets attracteurs. Comment savoir lequel des deux joueurs possède une stratégie gagnante ?
9. Compléter la fonction `StrategieAleatoire(tab,joueur)` afin qu'elle renvoie une position `s` choisie au hasard dans `tab`, excepté le carré empoisonné (sauf si le joueur `joueur` n'a pas le choix.)
   On pourra utiliser la fonction `np.random.randint(n)` qui renvoie un entier choisi aléatoirement entre 0 et `n`-1.
10. Écrire une fonction `StrategieGagnante(tab,joueur)` qui va utiliser le calcul de l'attracteur de la façon suivante : si la position n'est pas dans l'attracteur renvoyer un coup au hasard, sinon renvoyer un coup qui reste encore dans l'attracteur que l'on a calculé.
11. Exécuter la dernière cellule de l'algorithme en enlevant les `"""` sur les lignes de code.
    Comprendre le code proposé et expliquer ce qu'il fait.
12. Coder une fonction heuristique `H(tab)` de la façon suivante :
    - si la tablette ne contient que le carré `(0,0)` renvoyer -1 (synonyme de défaite),
    - si la tablette ne contient qu'une seule ligne ou une seule colonne renvoyer 1 (synonyme de victoire) et 0 dans tous les autres cas (synonyme qu'on ne sait pas trop quel coup jouer).
13. Écrire les algorithmes `Minimax(tab,n)` et `Maximin(tab,n)` avec cette heuristique (s'aider du cours).
14. Modifier les algorithmes `Minimax(tab,n)` et `Maximin(tab,n)` écrits à la question précédente de sorte à ce qu'ils renvoient respectivement le triplet `(mini,i0,j0)` et `(maxi,i0,j0)` où `(i0,j0)` sont les coordonnées du carré à manger dans `tab` pour arriver au sommet suivant du graphe en suivant l'algorithme `Max-Min`.
15. Utiliser l'algorithme `Minimax` avec `n` $= 3$ et `n` $= 7$ pour définir une stratégie qui suit les coups proposés par cet algorithme puis le comparer avec la stratégie hasard et la stratégie gagnante en enrichissant le code proposé à la dernière cellule (celui exécuté à la question **9.**). Qu'en pensez-vous ?