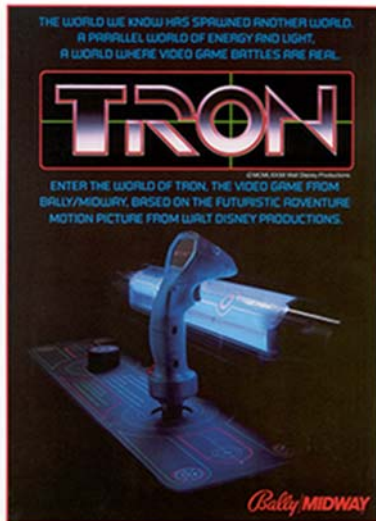


TRON

Tron est un jeu vidéo d'arcade commercialisé en 1982 adapté du film sorti la même année. Il a par la suite été porté sur le Xbox Live Arcade et a connu une suite au cinéma : suite « Tron Héritage » réalisée en 2011 par les studios Disney.



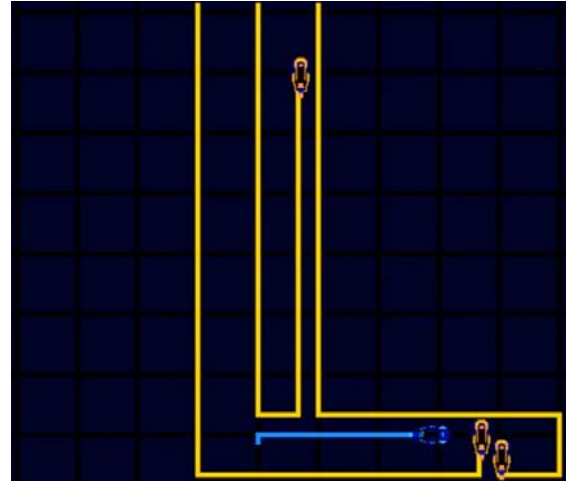
Jaquette du jeu 1982



Dual entre motos lumineuses

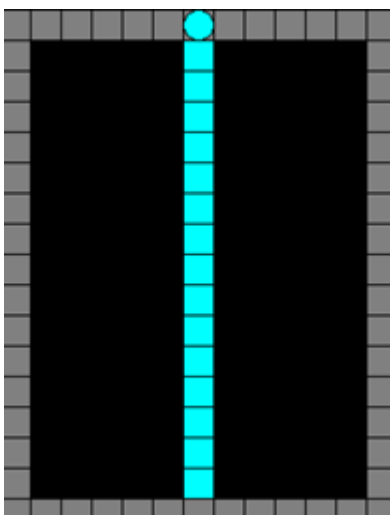


Film Disney 2011



La version borne d'arcade 1982

Il s'agissait d'un jeu où le joueur challengeait des joueurs IA dans une variante du jeu Snake. Le joueur pilote une moto bleue qui sur son passage laisse un mur de lumière bleue sur lequel peuvent s'écraser les motos ennemies de couleur orange. Les IA ennemis marquent aussi l'arène en laissant un mur de lumière orange infranchissable. L'objectif du jeu consiste à survivre le plus longtemps sans s'écraser sur un mur.



Ouvrez le fichier exemple du projet et lancez-le. Le joueur actuel se déplace uniquement vers le haut jusqu'à rentrer en collision avec le mur de l'arène. A chaque fois que le joueur avance d'une case son score augmente de 1.

Partie 0 : choix technique

Nous avons choisi une structure de données optimisée, ainsi nous stockons la grille de jeu à l'intérieur d'un tableau Numpy. Cette librairie est très efficace et relativement complexe. Rassurez-vous, nous n'utiliserons qu'une infime partie de cette librairie et principalement pour nous faciliter le codage.

Ouvrez le fichier **Tron.py**. Le repère que nous utilisons dans le jeu Tron correspond au repère cartésien habituel : l'origine (0,0) est en bas à gauche. Nous utilisons la fonction `numpy.array([...])` pour construire le tableau Numpy à partir d'une liste Python. Nous choisissons un tableau fait d'octets, `np.uint8`, ceci afin de limiter la taille mémoire qui reste une contrainte forte sur les performances.

```
Data =
[ [1,1,1,1,1,1,1,1,1,1,1,1],
  [1,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,0,1],
  [1,1,0,0,0,0,0,0,0,0,0,1],
  [1,1,1,1,1,1,1,1,1,1,1,1] ]

GInit = np.array(Data,dtype=np.int8)
GInit = np.flip(GInit,0).transpose()
```

Pour stocker les données de la partie, nous créons une classe/structure qui va nous permettre de regrouper la grille, la position du joueur et le score de partie dans un même objet. Cet objet facilite de plus l'accès aux données par l'écriture : `PartieCourante.Grille`. La fonction `copy()` permet de recopier toutes les données de la partie dans un autre objet.

```
class Game:
    def __init__(self, Grille, PlayerX,    PlayerY, Score=0):
        self.PlayerX = PlayerX
        self.PlayerY = PlayerY
        self.Score = Score
        self.Grille = Grille
```

```
def copy(self):
    return copy.deepcopy(self)

GameInit = Game(GInit,3,5)
```

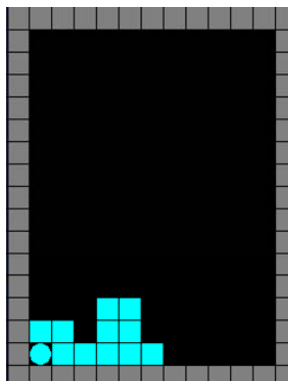
Examinez le code source fourni. Les routines d’affichage en Tkinter peuvent être ignorées car vous n’avez pas à intervenir sur cette partie du code. Il faut cependant comprendre la logique de fonctionnement de la fonction `Play ()`. Car vous allez commencer à coder à l’intérieur de cette fonction.

Partie 1 – Jeu aléatoire

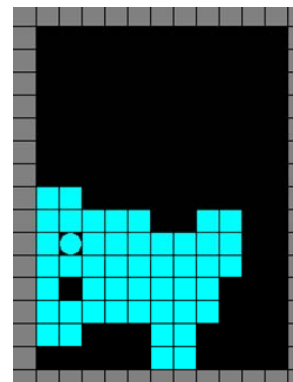
Créez une fonction qui depuis la position courante du joueur retourne la liste des déplacements possibles. Par exemple, si le joueur peut aller uniquement à gauche et à droite, alors la fonction retournera : `[(-1,0), (1,0)]`.

Ensuite, au lieu de toujours déplacer le joueur vers le haut, nous allons choisir au hasard une direction parmi celles disponibles. Pour cela, utilisez la fonction `random.randrange(a)`, qui retourne un entier compris entre 0 et `a-1`. Pensez à importer le package `random` pour pouvoir l’utiliser : `import random`.

Lancez les parties et examiner le score obtenu.



Une partie qui n’a pas eu de chance !



Partie intéressante, mais peut mieux faire !

Un peu d’histoire – pas si vieille l’histoire !!

L’algorithme que nous vous proposons d’étudier est une version simplifiée de l’algorithme MCTS (Monte Carlo Tree Search). Cette approche type Monte Carlo fut le centre des premières versions du programme AlphaGo mis en place par l’entreprise Google DeepMind en 2012. En octobre 2015, AlphaGo devient le premier programme à battre un joueur professionnel de Go sur une grille de taille normale (19×19) sans handicap. Il s’agit d’une étape symboliquement forte puisque programmer une IA pour le jeu de Go était alors un défi complexe de l’intelligence artificielle. En mars 2016, AlphaGo bat Lee Sedol, un des meilleurs joueurs mondiaux (9e dan professionnel). Le 27 mai 2017, il bat le

champion du monde Ke Jie. Cet algorithme sera encore amélioré dans les versions suivantes. Finalement, naît AlphaGo Zero en octobre 2017 qui atteint un niveau supérieur uniquement en ayant appris à jouer en simulant des parties contre lui-même. En décembre 2017, il surpasse largement, toujours par auto-apprentissage, le niveau de tous les joueurs humains et logiciels, non seulement au Go, mais aussi aux échecs.



Pour beaucoup de joueurs asiatiques, le Go est un mode de vie. Dans l'Asie ancienne, le Go était l'un des quatre piliers nobles de l'accomplissement intellectuel, avec la musique, la peinture et la poésie. C'est pourquoi le fait qu'une intelligence artificielle réussisse à battre les meilleurs joueurs est à la fois bouleversant au niveau de la recherche mais également au niveau sociologique pour les asiatiques.

Partie 2 – Algorithme de type Monte Carlo

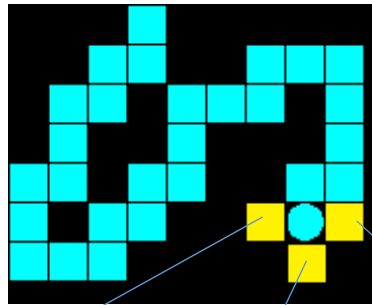
L'algorithme que nous allons mettre en place stocke en mémoire la grille de jeu actuelle et la position du/des joueurs. Il ne conserve aucune information relativement aux coups précédemment joués.

Pour décider quel coup jouer, l'algorithme va comme habituellement rechercher les différents coups possibles et les stocker dans une liste (LC). Ensuite, il va étudier le potentiel de chacun d'entre eux.

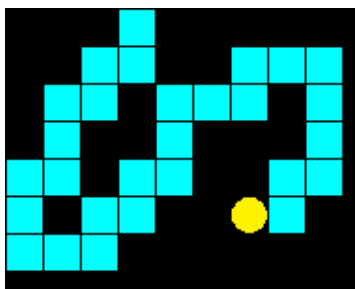
Comment estimer le potentiel d'un coup à jouer C de LC ? La méthode consiste à copier la grille actuelle G et à jouer le coup en question C pour obtenir un nouvel état de la grille de jeu : G'. Ensuite, depuis cet état G', on va simuler un très grand nombre de parties, ces parties démarrant toutes depuis G'. Chacune des parties simulées à partir du coup C choisi va retourner un score. Ainsi, en moyennant les scores de ces simulations, on obtient une sorte de gain potentiel associé au coup C en question. Pour chaque coup à jouer dans LC, on recommence ce processus.

Pour choisir quel coup jouer dans la partie principale (grille G), nous sélectionnons le coup associé au plus fort potentiel. Nous illustrons ce processus dans le schéma ci-dessous :

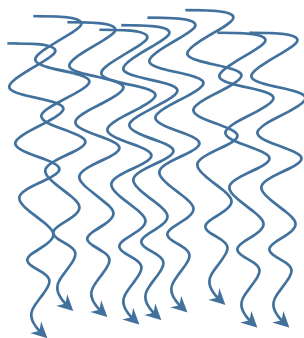
Etat de la grille de jeu et
position du joueur



Option numéro 1



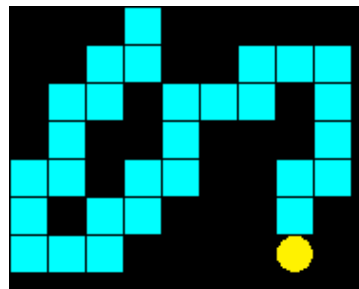
Simulation de milliers de parties
indépendantes à partir
de ce point



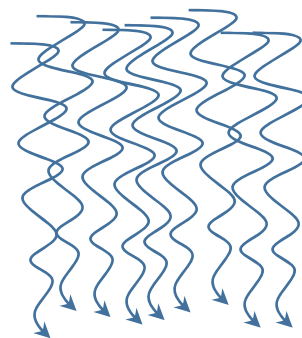
17 42 55 43 62 71 79

Score Moyen : 72.4

Option numéro 2



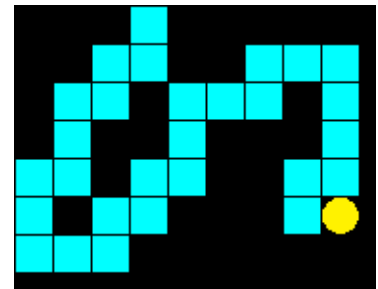
Simulation de milliers de parties
indépendantes à partir
de ce point



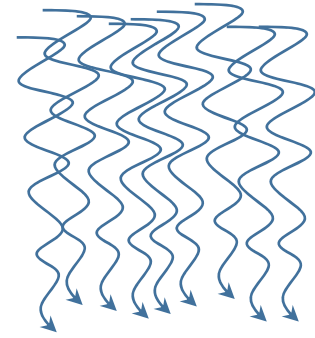
23 33 45 55 43 66 54

Score Moyen : 44.5

Option numéro 3



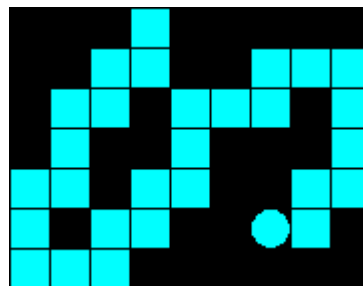
Simulation de milliers de parties
indépendantes à partir
de ce point



14 53 54 66 38 54 99

Score Moyen : 65.4

Option 1 retenue



Pour la mise en place, nous allons réutiliser la méthode précédente « du jeu aléatoire » pour simuler les parties :

```
SimulationPartie (G : Game) :
  Boucle Infinie
    L = DirectionsPossibles(G)
    Si L est vide => Retourner le nombre de cases parcourues
    Choisir une direction au hasard dans L
    Créer le mur
    Déplacer le Joueur
```

Pour lancer une série de simulations à partir d'un état de jeu, nous effectuons à chaque simulation une copie de cet état de jeu que nous utilisons pour la simulation. Une fois la partie simulée terminée, nous récupérons la quantité de cases parcourues. Nous sommes cette quantité pour obtenir le gain potentiel. La grille de jeu simulée est abandonnée une fois la simulation terminée. Nous recommençons la simulation suivante à partir d'une nouvelle grille copie de la grille originale. A ce niveau, aucune simulation ne doit modifier la grille principale de jeu :

```
MonteCarlo(G : Game, nombreParties)
  Total = 0
  Pour i allant de 0 à nombreParties
    Game2 = G.copy()
    Total += SimulationPartie(Game2)
  Retourner Total
```

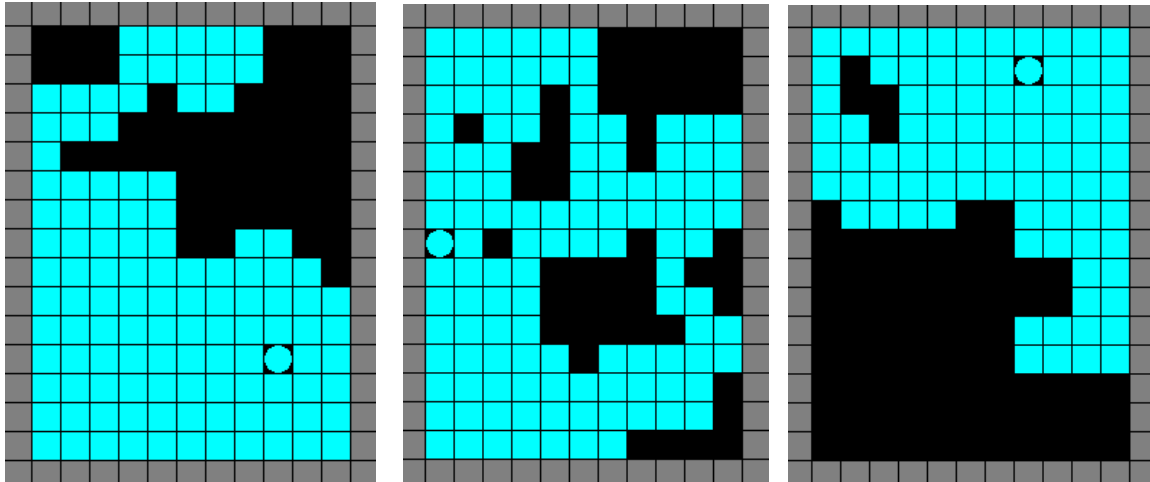
Il vous reste à écrire la fonction qui détermine le coup à jouer pour la partie courante.

Attention, cet algorithme fonctionne à deux niveaux :

- Le premier niveau correspond à la partie courante, on y analyse les coups possibles, on détermine leurs potentiels et on joue le coup le plus prometteur. On recommence jusqu'à ce qu'il n'ait plus de coups possibles.
- Le deuxième niveau consiste à simuler des parties depuis le coup à étudier fourni par la partie principale. Il y a aussi une grille de jeu, construite par copie de la partie principale, et sans influence sur cette dernière.

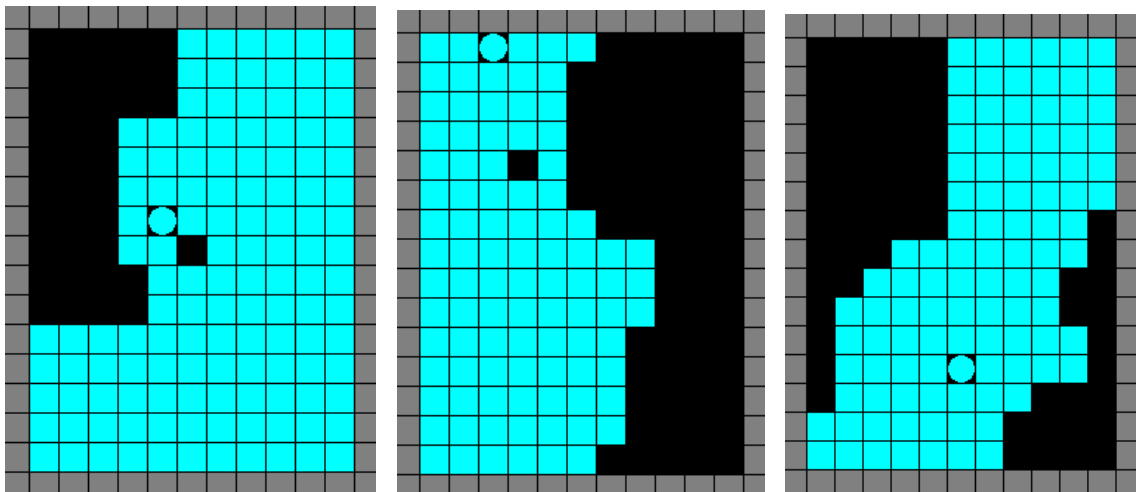
Résultats obtenus avec l'algorithme de Monté Carlo

Pour 10 parties simulées



Score moyen des parties : 95 – on sent que l'IA est devenue plus « professionnelle »

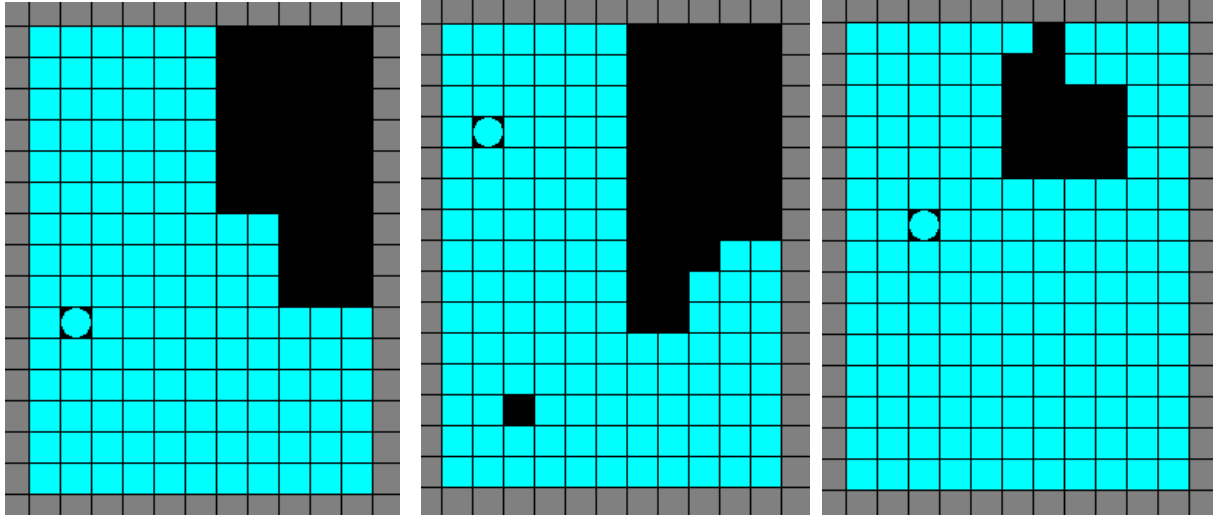
Pour 100 parties simulées



Le score moyen des parties passent à 110. C'est mieux ! Certes, mais ce qui est le plus impressionnant est que l'IA commence à se déplacer de manière tactique. Elle suit les bords, finit de remplir une zone avant de la quitter. Elle devient méthodique ! On peut s'apercevoir sur les résultats finals que les zones non peintes sont rares et souvent réduites à 1 carré. Cependant, il reste encore des zones non explorées qui font perdre des points !

Pour 1000 parties simulées

Ça commence à ralentir ☺

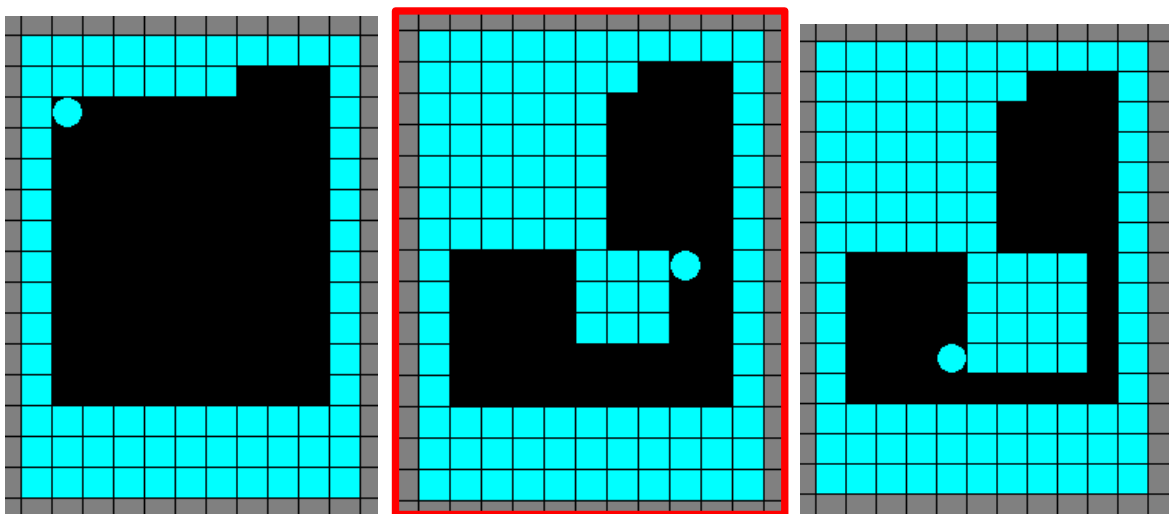


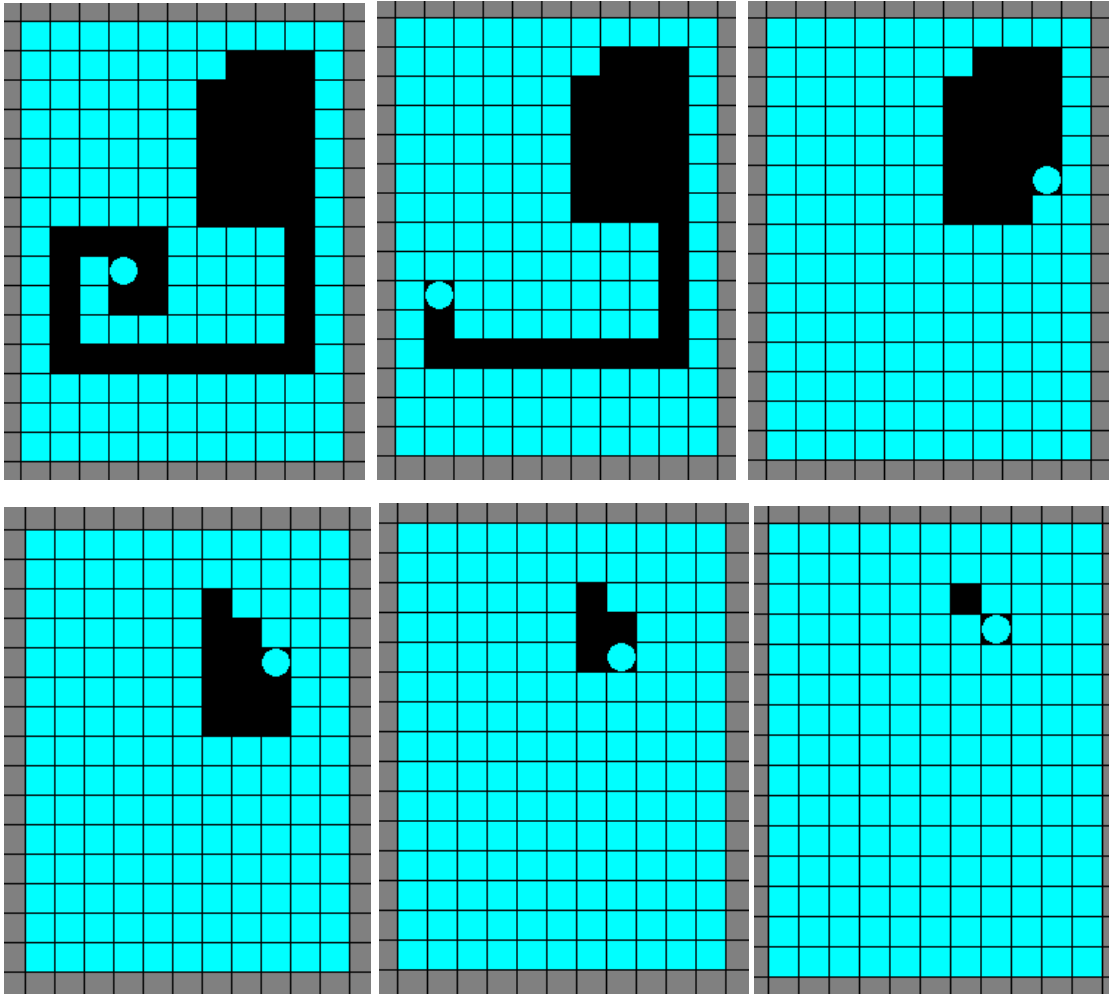
Le score moyen passe à 120, on semble avoir encore progressé. Parfois, le joueur a un choix à faire : explorer une cavité au risque de s'y enfermer ou l'abandonner pour rester dans la zone plus « ouverte ». Mais construire de manière aléatoire un chemin entrant et ressortant d'une cavité semble statistiquement peu probable, alors notre IA va décider d'éviter les cavités. Aller au-delà de 1000 parties simulées semble trop lent, il va falloir changer de style de programmation !

Partie 3 – Geek Time !

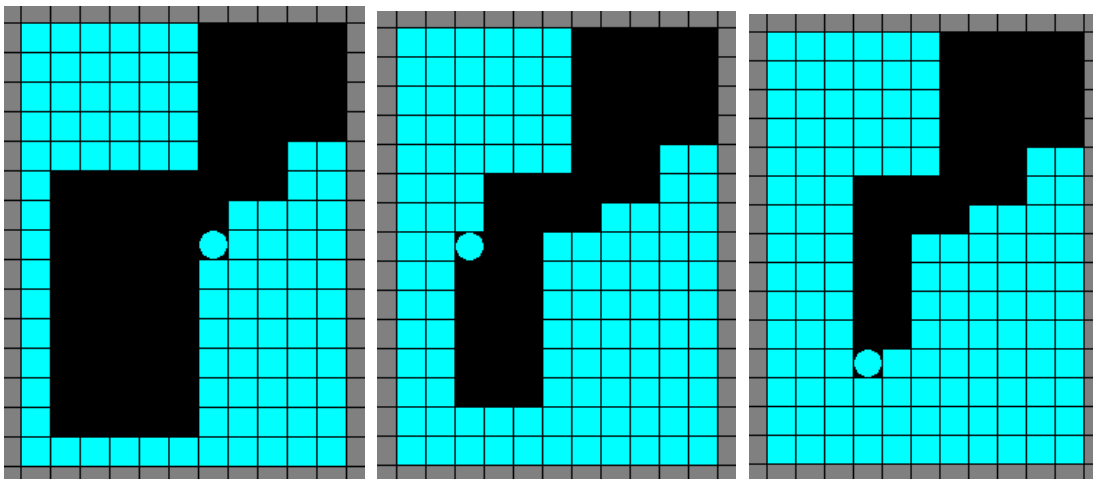
Pour 10 000 parties simulées

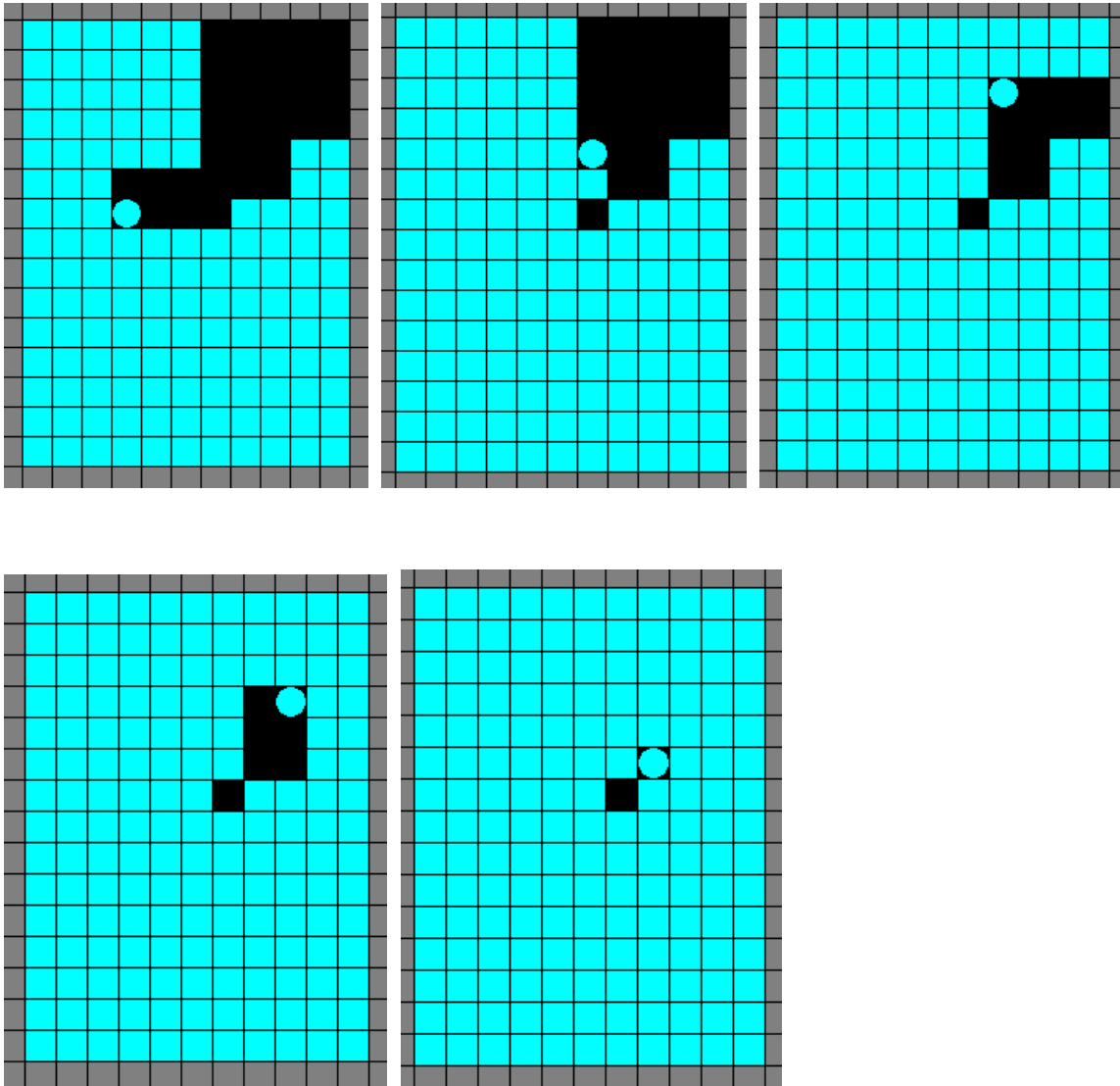
Nous vous présentons le déroulement d'une partie :





Le comportement de l'IA a véritablement changé. Elle effectue des parcours plutôt en horizontal / vertical pour effectuer le remplissage. Si deux zones à explorer se forment, elle va laisser un chemin assez large (2 cases) pour aller explorer la première zone et revenir par ce chemin pour ensuite explorer la seconde, voir le dessin encadré en rouge ci-dessus. Le parcours est parfait, au sens où il ne reste plus qu'une case inexplorée dans le pire cas. Vous allez dire : oui mais c'était évident, il fallait parcourir la grille en balayant en horizontal/vertical. Oui. Mais ce qu'il faut comprendre, c'est que finalement l'IA est devenue aussi maline que vous ! Autre exemple au moment critique :





Le score moyen des parties est de 156 sur un total maximum de 165 ! C'est plutôt très bien.

Comment optimiser les performances

Nous avons constaté que l'IA prenait toute son ampleur à partir de 10 000 simulations, mais à ce niveau-là, nous souffrons d'un manque de performance de calcul. Python fut conçu comme un langage de script, donc lent, mais par contre très évolué, souple et simple. Sa lenteur relative disparaît car le langage dispose de multiples bibliothèques optimisées et très performantes. Ainsi, si le langage Python est utilisé comme chef d'orchestre effectuant peu de traitements et déléguant les tâches lourdes aux bibliothèques, le programme construit autour de cette architecture reste très efficace.

L'environnement Python n'a pas toujours été un écosystème aussi complet. Cependant, aujourd'hui la grande diversité des bibliothèques disponibles permet de couvrir 95% des besoins. Développer autour de Python s'avère maintenant un choix judicieux car le langage est simple, flexible et intuitif. Il permet de coder vite et bien.

Cependant, quelques rares fois notre besoin n'est pas couvert par une librairie existante. Il faut alors programmer tous les calculs en utilisant uniquement Python et, comme vous l'avez constaté, dès que la quantité de traitements est importante, les performances ne sont pas au RDV.

Les années précédentes, nous avons utilisé une solution basée sur la bibliothèque Numba. Cette dernière permet de convertir à la volée une fonction Python en langage bas niveau (équivalent à du C), de la compiler et de la réintégrer dans le programme final. Pour fonctionner, ce mécanisme magique imposait de nombreuses contraintes sur le passage des paramètres en les limitant à des types de bases et en interdisant les listes ainsi que les variables globales. Les performances étaient au RDV et permettaient d'atteindre les 10 000 simulations par coup à étudier. Cependant, étant proposé comme une boîte noire, le mécanisme de conversion/compilation automatique pouvait dysfonctionner et arriver à trouver la source du problème s'avérait très difficile.

Cette année nous vous proposons une autre approche basée sur la librairie Numpy beaucoup plus connue et plus établie que Numba. Elle va nous permettre de mettre en place le traitement parallélisé de nos simulations. Ainsi, au lieu de traiter les simulations les unes après les autres, nous allons traiter toutes les simulations en même temps et jouer un coup en parallèle dans chacune d'elle. Cette approche est utilisée par les cartes 3D GPU, qui avec leur système de cœurs multiples - une carte NVidia contient ~3000 Cuda Cores - permet d'effectuer des calculs massivement parallèles.

Les algorithmes parallèles ou vectoriels sont assez déstabilisants car ils ne correspondent pas à votre méthode habituelle de programmer où en général vous effectuez les traitements de manière séquentielle, c'est-à-dire, les uns après les autres. Cependant, avec la vitesse des processeurs qui n'évolue guère, avec les ordinateurs quantiques/photoniques qui peinent à arriver, les algorithmes vectoriels restent l'implémentation la plus probable lorsque l'on cherche de la performance.

Etape 1 : Timing

Avant de parler d'optimisation, il faut déjà être capable de mesurer le temps mis par l'ordinateur pour effectuer son traitement. Dans la boucle de jeu principale, insérez le code suivant :

```
BoucleDeJeu
...
while (...)
    Tstart = time.time()
    ...
    print(time.time() - Tstart)
    Affichage()
```

Ces deux lignes supplémentaires vont nous permettre de mesurer le temps de calcul en seconde pour chaque coup de jeu. Pensez à importer le package time pour avoir accès à cette fonction. Pour 10 000 simulations, vous devriez obtenir un temps d'attente de l'ordre de 1 à 2 secondes sur un PC moyenne gamme (E5 4 cœurs 3GHz). Le chronomètre fourni par la fonction time() n'est pas précis du tout, il fonctionne par pas de 1/16^{ème} de seconde. Donc même si vous voyez plusieurs chiffres après la virgule, ne vous laissez pas impressionner. On veillera donc pour les comparaisons à avoir toujours un temps d'exécution supérieur à 0.3 seconde.

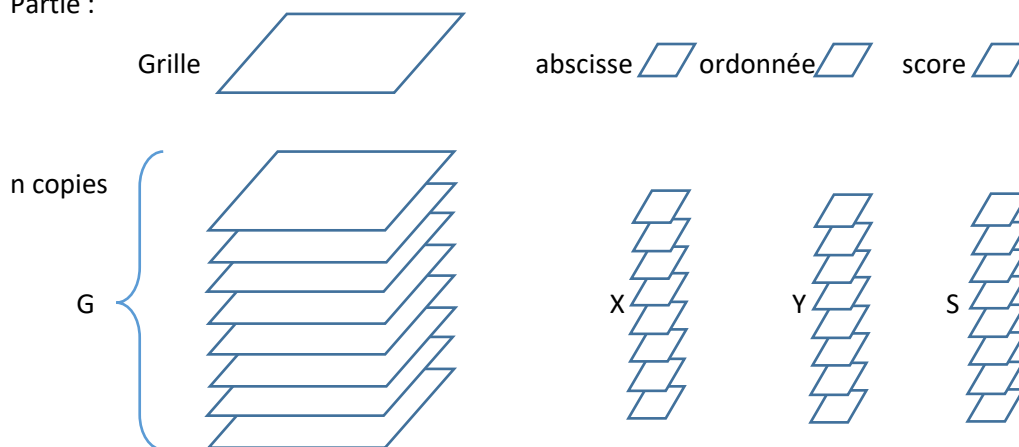
Etape 2 : Numpy introduction

La librairie Numpy permet de stocker des tableaux de données et d'effectuer des traitements très efficacement sur ces données. C'est cette spécificité que nous allons utiliser dans le projet. La librairie Numpy est aussi connue car elle permet d'effectuer des calculs matriciels efficaces, mais nous ne nous intéressons pas à cette partie (Ouf ! dirons certains).

Nous allons laisser de côté le code mis en place dans `Tron.py`. Vous allez maintenant ouvrir le fichier `Tron Parallel.py`. Tout le travail qui suit sera maintenant implémenté dans ce fichier. Nous allons mettre en place le traitement parallèle des simulations avec Numpy. Uniquement une fois votre code des simulations validé, vous l'implémenterez dans le code principal pour permettre au jeu Tron d'atteindre les 30 000 simulations ou plus.

En début de simulation, nous recevons comme paramètre une partie contenant : la grille de jeu, la position du joueur en X et Y et le score. Pour gérer des parties en parallèle, nous allons dupliquer autant de fois que de parties traitées la partie passée en paramètre. La grille de partie est un tableau à deux indices, en empilant plusieurs grilles, nous obtenons ainsi un tableau à trois indices, le premier indiquant le numéro de partie. La position en abscisse du joueur est une valeur, en empilant plusieurs fois cette valeur, nous obtenons un tableau à 1 indice (on dit aussi vecteur dans ce cas précis). L'indice de ce vecteur indique le numéro de partie concernée. Il en est de même pour la position en ordonnée et le score :

Partie :



La création de ces différents tableaux est gérée par la partie de code suivante. L'empilement est mis en place par la fonction `tile` qui reste hors programme. L'import `numpy` as `np` permet d'utiliser toutes les fonctions de la librairie Numpy en utilisant la notation `np.fonction(...)` :

```
G = np.tile(Game.Grille,(nb,1,1))
X = np.tile(Game.PlayerX,nb)
Y = np.tile(Game.PlayerY,nb)
S = np.tile(Game.Score,nb)
```

Etape 3 : Syntaxe Numpy

Numpy fournit des syntaxes très pratiques et très élégantes pour faire des traitements rapides. Par exemple pour un tableau 2D, si je veux écrire n valeurs V_i aux positions (X_i, Y_i) , on écrit habituellement :

Pour $n = 0$ à $n-1$:
 $G[X[i], Y[i]] = V[i]$

Dans la syntaxe Numpy, il suffira d'écrire : $G[X, Y] = V$

Pour que cette écriture fonctionne, il faut que les vecteurs X , Y et V soient exactement de la même taille, ce qui dans notre cas est garanti lors de leur création. Mais, en ce qui nous concerne, nous n'avons pas une seule grille mais plusieurs. Pour faire une modification/lecture dans chaque grille, nous allons utiliser une astuce en stockant dans un vecteur I les valeurs $0, 1, 2, \dots$ jusqu'à $n-1$ avec n correspondant aux nombres de parties simulées. Ainsi l'écriture :

Pour $i = 0$ à $n-1$:
 $G[i, X[i], Y[i]] = V[i]$ \longleftrightarrow Devient en Numpy : $G[I, X, Y] = V$

Il existe plusieurs manières de créer des tableaux Numpy. Nous avons vu la syntaxe `np.array` utilisant une liste Python comme paramètre. Nous allons maintenant présenter les fonctions `np.ones` et `np.zeros` permettant de créer des tableaux initialisés à la valeur 1 et 0 respectivement. Comme paramètres, elles prennent la taille sous forme d'un tuple et aussi le type des éléments du tableau. Nous utilisons des `np.int8` correspondant à un entier non signé sur 8 bits, c'est-à-dire à un nombre compris entre 0 et 255 ce qui suffit largement pour nos besoins. Les `int8` permettent de limiter la mémoire à un octet par case ce qui prend 4x moins d'espace que si nous avions pris des entiers 32 bits et 8x moins que si nous avions pris des entiers 64 bits. Si vous omettez cet attribut, attention, car Numpy crée par défaut des tableaux de flottants et un flottant ne peut être utilisé comme valeur d'indice.

Comme nous allons gérer plusieurs parties en parallèle, certaines vont finir avant les autres. Cependant, comme nous traitons les parties en parallèle et qu'il n'est pas vraiment possible de traiter celles qui ont fini différemment des autres, nous allons ajouter une direction supplémentaire correspondant au « sur place ». Ainsi, les joueurs étant coincés entre 4 murs vont se déplacer « sur place ». Ainsi, nous aurons 5 directions possibles : sur place, à gauche, en haut, à droite et en bas. Nous allons stocker dans deux vecteurs `dx` et `dy` les déplacements pour ces 5 cas :

```
dx = np.array([0, -1, 0, 1, 0], dtype=np.int8)
dy = np.array([0, 0, 1, 0, -1], dtype=np.int8)
```

Dans le code source, tous les joueurs se déplacent vers le haut. Ainsi le vecteur contenant le numéro de déplacement pour chaque vecteur est rempli de 2 :

```
Choix = np.ones(nb,dtype=np.uint8) * 2
```

Ainsi pour l'ensemble des joueurs, nous pouvons construire deux vecteurs DX et DY de longueur n correspondant au déplacement de chaque joueur dans les n parties :

```
DX = dx[Choix]
DY = dy[Choix]
```

Cette syntaxe se lit de la manière suivante :

$DX = dx[K]$ \longleftrightarrow Pour $i = 0$ à $n-1$: $DX[i] = dx[K[i]]$

Les opérations standards sur un tableau Numpy sont permises, ainsi l'écriture :

$U = 2*V - W$ avec des vecteurs Numpy \longleftrightarrow Pour $i = 0$ à $n-1$: $U[i] = 2 * V[i] - W[i]$

Ainsi pour déplacer l'ensemble des joueurs, il suffit d'écrire :

```
X += DX
Y += DY
```

Pour afficher l'ensemble des différentes parties, nous fournissons la fonction AffGrilles(...) qui permet d'afficher plusieurs parties en colonnes dans la sortie texte. Pensez à utiliser un environnement avec une sortie d'affichage assez large, Pyzo le fait par exemple.

Etape 4 : création de l'algorithme vectoriel

Nous allons suivre le même développement que l'algorithme écrit en Python. Tout d'abord nous allons examiner les cases voisines du joueur pour savoir ces options de jeu.

Ecrivez un code permettant de charger, pour chaque grille, la valeur de la case à gauche de la position du joueur. Le résultat obtenu correspond à un vecteur de taille n. Ce vecteur contient la valeur 0 si la case est vide ou d'autres valeurs pour indiquer la présence d'un mur du décor ou d'un mur laissé par le joueur. Pour faciliter les traitements, nous allons transformer ce vecteur en vecteur de valeurs 0 ou 1 : 1 pour une case vide et 0 pour une case occupée. Voici la syntaxe à utiliser :

$V_{gauche} = (V_{gauche} == 0) * 1$

Le test $==$ transforme le vecteur V_{gauche} en vecteur de booléens. L'opération $*1$ transforme les valeurs True en 1 et les valeurs False en 0.

Nous allons maintenant créer n listes servant à stocker les directions possibles pour chaque joueur de chaque partie. Pour cela, nous allons créer un tableau de dimension $n \times 4$ servant à stocker chaque possibilité. Nous aurons aussi un vecteur Indices permettant de savoir la prochaine case disponible dans chaque liste.

```
LPossibles = np.zeros((nb,4),dtype=np.int32)
Indices = np.zeros(nb,dtype=np.int32)
```

Pour tous les joueurs pouvant se déplacer à gauche, il faut ajouter la valeur 1 dans leurs listes respectives. Modifiez ensuite le vecteur Indices pour que chaque insertion fasse avancer de 1 l'indice correspondant. Ce n'est pas du tout évident en première lecture, mais l'opération à mettre en place s'avère relativement simple. Recommencez pour chacune des 4 directions.

Ainsi dans l'exemple :

LPossibles	Indices
1 2 4 0	3
2 3 0 0	2
0 0 0 0	0

Dans la partie 0, le joueur 0 peut se déplacer dans les directions 1 2 et 4 soit à gauche, en haut et en bas. Comme il a 3 choix, la valeur Indices[0] vaut 3. Le joueur 1 a seulement 2 possibilités pour se déplacer : 2 (en haut) et 3 (à droite). Le joueur 2 n'a aucun choix, les 4 valeurs associées à sa liste sont nulles et la valeur Indices[2] reste nulle.

Pour choisir au hasard un élément dans chaque liste, il faut tirer un nombre au hasard et lui appliquer un modulo en fonction du nombre d'indices disponibles. Ainsi un `rand()%3` peut retourner les valeurs 0, 1 ou 2. Pour connaître le nombre d'éléments dans la liste, le vecteur Indices est utile.

Cependant en informatique un modulo par la valeur 0 est une erreur. Nous avons donc un problème pour les joueurs entre 4 murs. Nous allons donc modifier les valeurs nulles dans le vecteur Indices pour les remplacer par la valeur 1. Le modulo 1 retournant toujours la valeur 0, cela aura pour effet de retourner la valeur à l'indice 0 dans la liste [0,0,0,0] ce qui désigne le déplacement « sur place » ce qui correspond à ce que l'on veut finalement. Voici la syntaxe à utiliser :

$V[V == a] = b \iff \text{Pour } i = 0 \text{ à } n-1 : \text{ if } V[i] == a : V[i] = b$

Pour créer un vecteur de valeurs aléatoires, il faut utiliser la fonction `np.random.randint(...)` et lui donner en paramètre la plage de valeurs pour les valeurs à créer. Cela est un problème car si nous donnons par exemple la plage 0 à 2, nous sommes coincés pour les listes à deux éléments car il faudrait des nombres de 0 à 1. Pour cela nous allons utiliser une astuce. Comme les listes contiennent 1, 2, 3 ou 4 éléments, nous allons utiliser le ppcm de ces valeurs, c'est-à-dire 12. Ainsi, en utilisant le modulo 1, on obtient toujours 0, en utilisant le modulo 2 sur les valeurs [0,1,2,3,4,5,6,7,8,9,10,11] on obtient [0,1,0,1,0,1,0,1,0,1,0,1], en utilisant le modulo 3 sur les valeurs [0,1,2,3,4,5,6,7,8,9,10,11], on obtient : [0,1,2,0,1,2,0,1,2,0,1,2] et finalement avec le modulo 4 : [0,1,2,3,0,1,2,3,0,1,2,3]. Ainsi chaque valeur potentielle est équiprobable. Finalement, nous utilisons le code suivant :

```
R = np.random.randint(12,size=n)
R = R % NbChoix
```

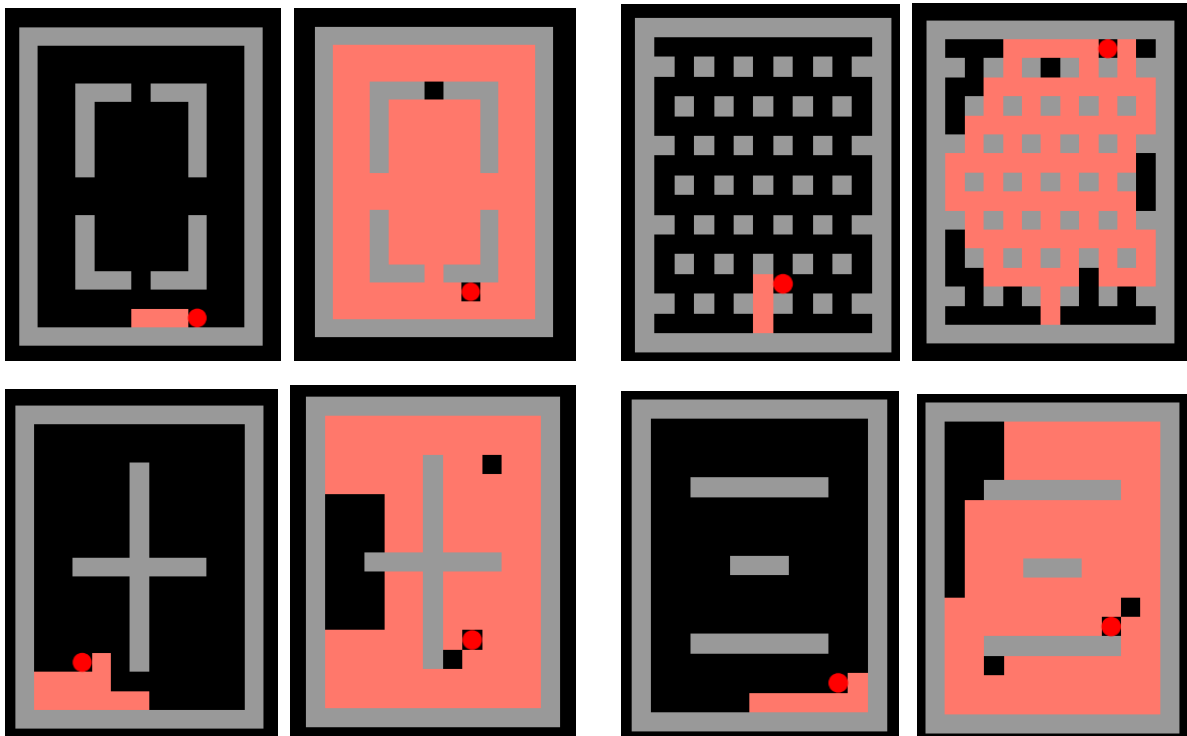

Après cela, il reste à déduire le déplacement choisi pour chaque joueur et à modifier leurs positions. On fait ensuite évoluer leurs scores en fonction. Pour savoir si les simulations sont toutes terminées, nous vous conseillons de vous baser sur les scores, en effet, lorsque tous les joueurs font du « sur place » les scores n'évoluent plus. Ainsi, leur total, que l'on peut obtenir par la fonction `np.sum(...)`, reste le même.

Pour terminer, nous vous conseillons de bien tester cette partie et de vous assurer qu'elle fonctionne parfaitement. Pensez à retourner un score qui va servir de notation pour l'algorithme appelant.

Intégré le code que vous avez créé dans le fichier principal `tron.py` avec un copier-coller. Les formats de données étant identiques, cela devrait bien se passer. Modifiez votre code pour qu'il utilise cette nouvelle fonction et augmentez progressivement le nombre de simulations !

PROJET

Maintenant que nous avons mis en place une IA efficace, testez là dans des décors plus compliqués à résoudre. Elle ne fera peut-être pas le meilleur score, mais, elle va quand même essayer de faire au mieux 😊



Proposez des configurations et essayez de déterminer quel type de situations met en difficulté l'IA et pourquoi.

Vous avez la possibilité de prolonger le sujet TRON en tant que projet pour votre atelier. Pour cela, nous vous proposons les points suivants :

- Mettre en place une TRON Arena avec plusieurs IA qui s'affrontent
- Ajoutez des éléments de décors : cases de téléportation / escaliers
- Faire des affrontements sur 2 ou 3 étages
- Ajoutez un joueur humain 😊