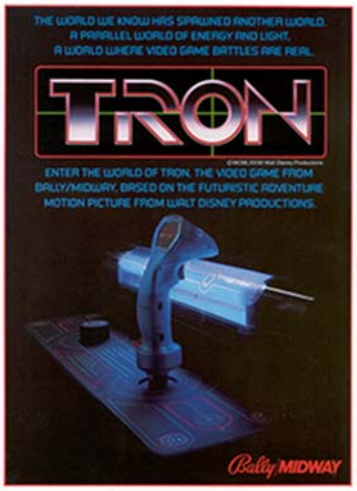
TRON

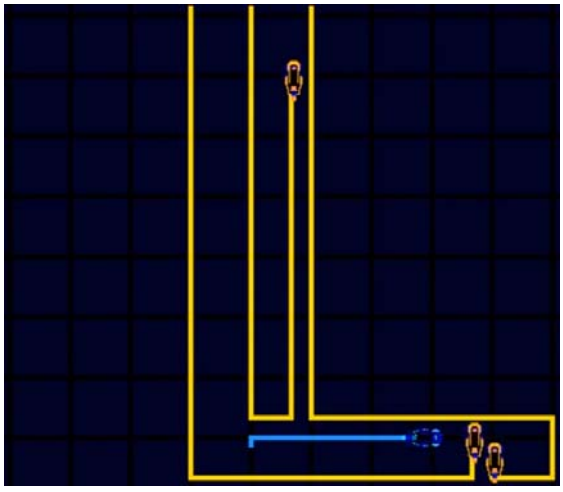
Tron est un jeu vidéo d'arcade commercialisé en 1982 adapté du film sorti la même année. Il a par la suite été porté sur le Xbox Live Arcade et a connu une suite au cinéma : suite « Tron Héritage » réalisée en 2011 par les studios Disney.

Jaquette du jeu 1982



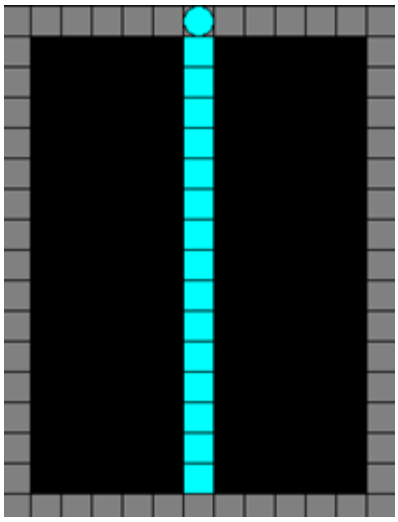
Dual entre motos lumineuses

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 bleu 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],
[1,0,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,1],
[1,1,0,0,0,0,0,0,0,0,0,0,1],
[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)
```

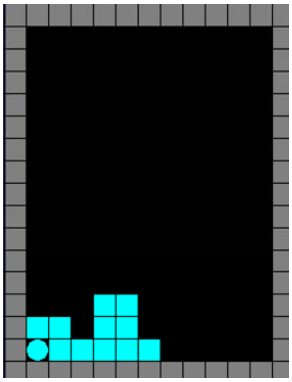
Examinez le code source fourni. Les routines d'affichage en Tkinter peuvent être ignorées car vous n'avez pas à intervenir sur cette partie du code. Il faut cependant comprendre la logique de fonctionnement de la fonction Play () . Car vous allez commencer à coder à l'intérieur de cette fonction.

Partie 1 – Jeu aléatoire

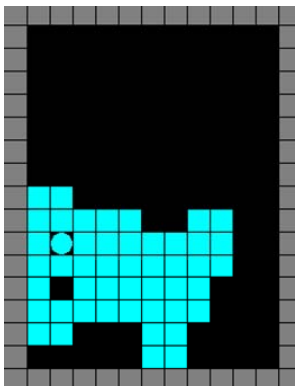
Créez une fonction qui depuis la position courante du joueur retourne la liste des déplacements possibles. Par exemple, si le joueur peut aller uniquement à gauche et à droite, alors la fonction retournera : [(-1,0), (1,0)].

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 veille 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

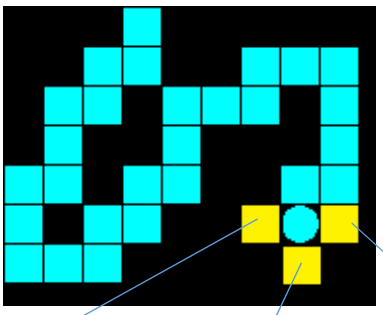
L'algorithme que nous allons mettre en place stocke en mémoire la grille de jeu actuelle et la position du/des joueurs. Il ne conserve aucune information relativement aux coups précédemment joués.

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.

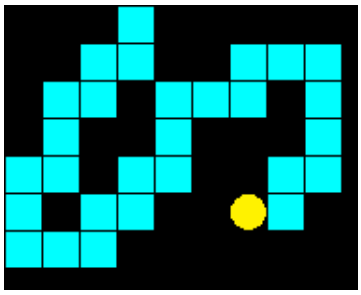
Comment estimer le potentiel d'un coup à jouer C de LC ? La méthode consiste à copier la grille actuelle G et à jouer le coup en question C pour obtenir un nouvel état de la grille de jeu : G'. Ensuite, depuis cet état G', on va simuler un très grand nombre de parties, ces parties démarrant toute depuis 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 process.

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

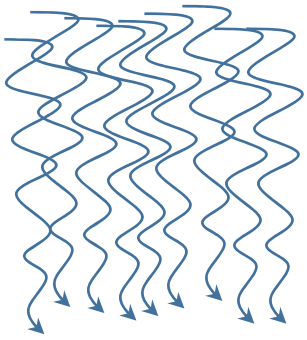
Etat de la grille de jeu et position du joueur



Option numéro 1



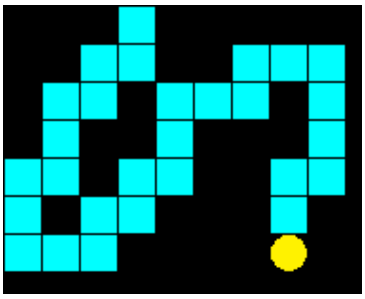
Simulation de milliers de parties indépendantes à partir de ce point



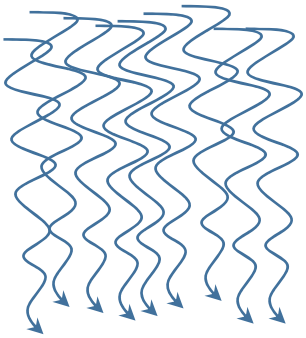
17 42 55 43 62 71 79

Score Moyen : 72.4

Option numéro 2



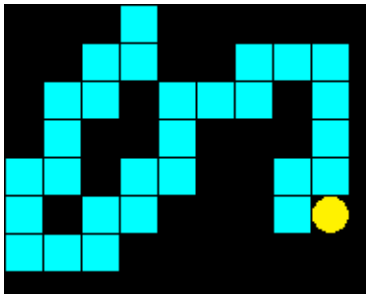
Simulation de milliers de parties indépendantes à partir de ce point



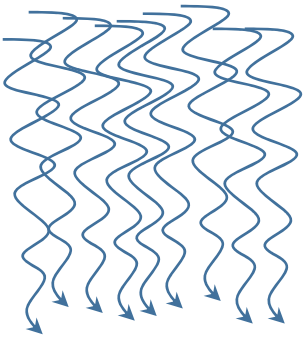
23 33 45 55 43 66 54

Score Moyen : 44.5

Option numéro 3



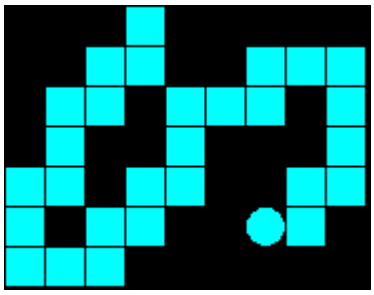
Simulation de milliers de parties indépendantes à partir de ce point



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 sommons 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.

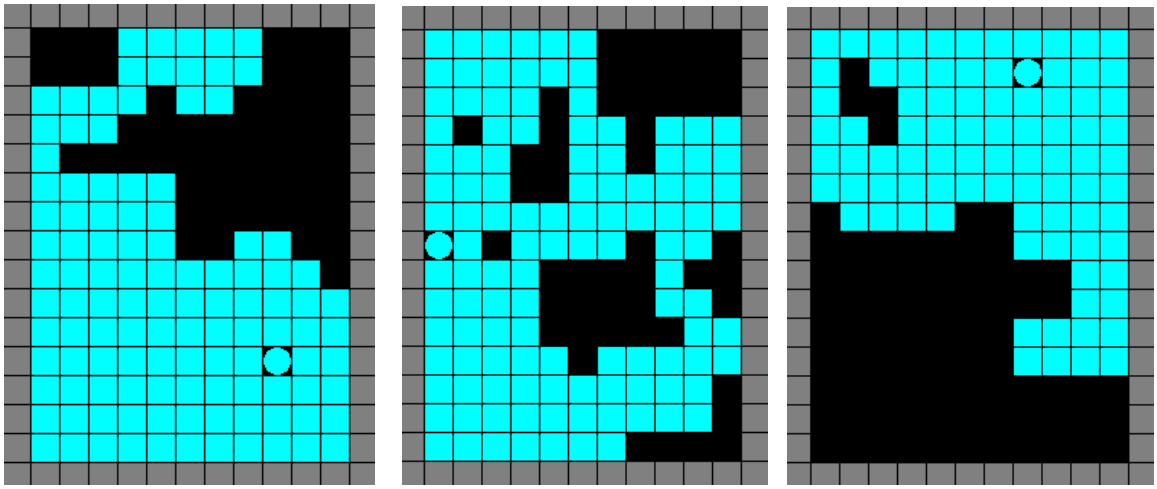
Attention, cet algorithme fonctionne à deux niveaux :

- 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.

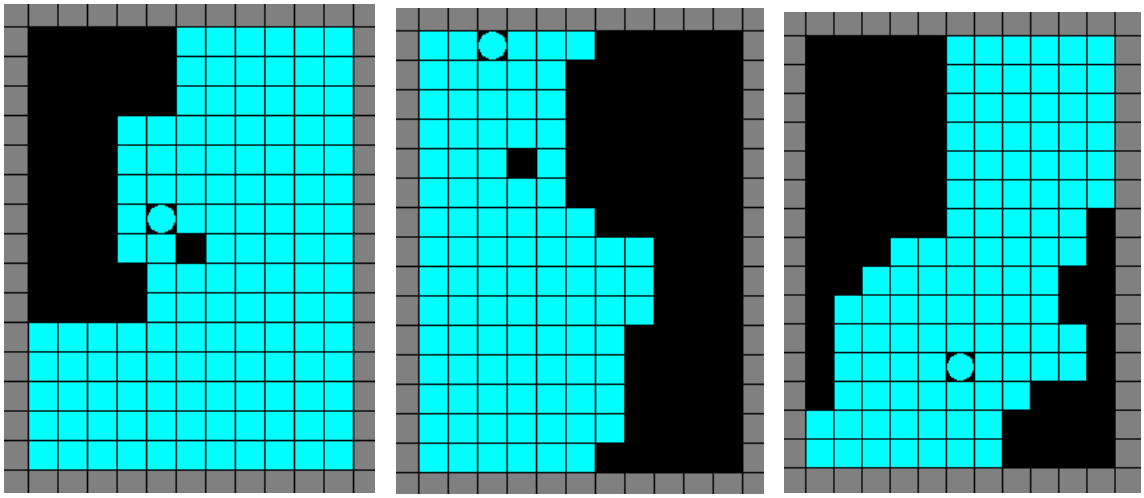
Résultats obtenus avec l'algorithme de Monté Carlo

Pour 10 parties simulées



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

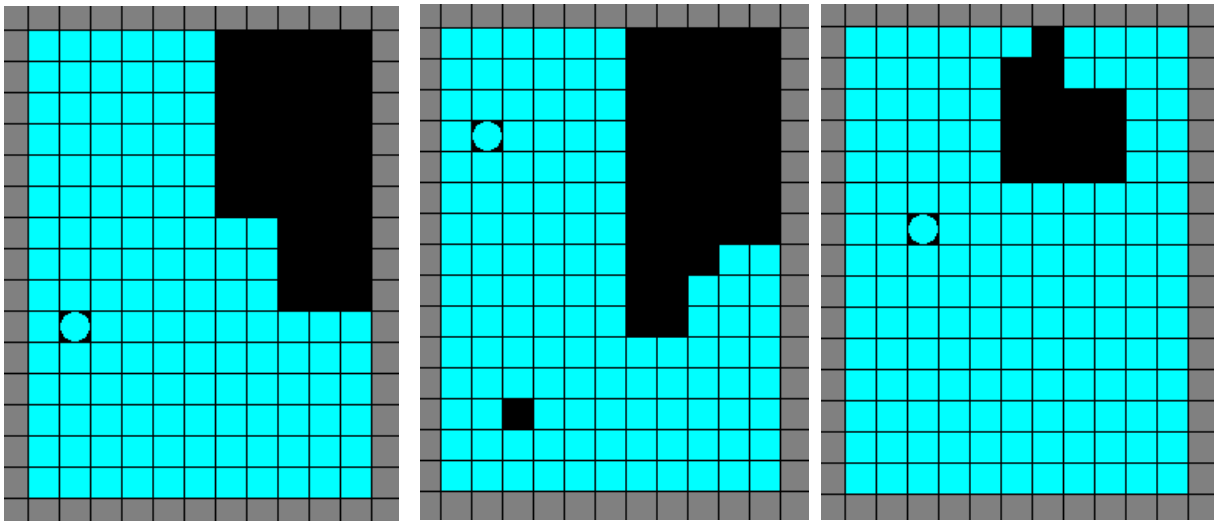
Pour 100 parties simulées



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 ☺

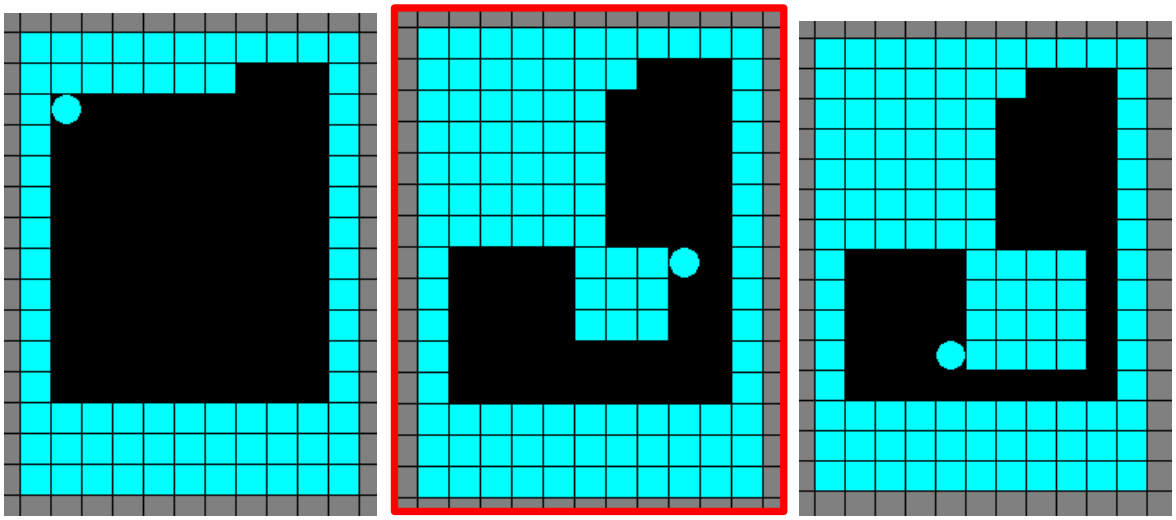


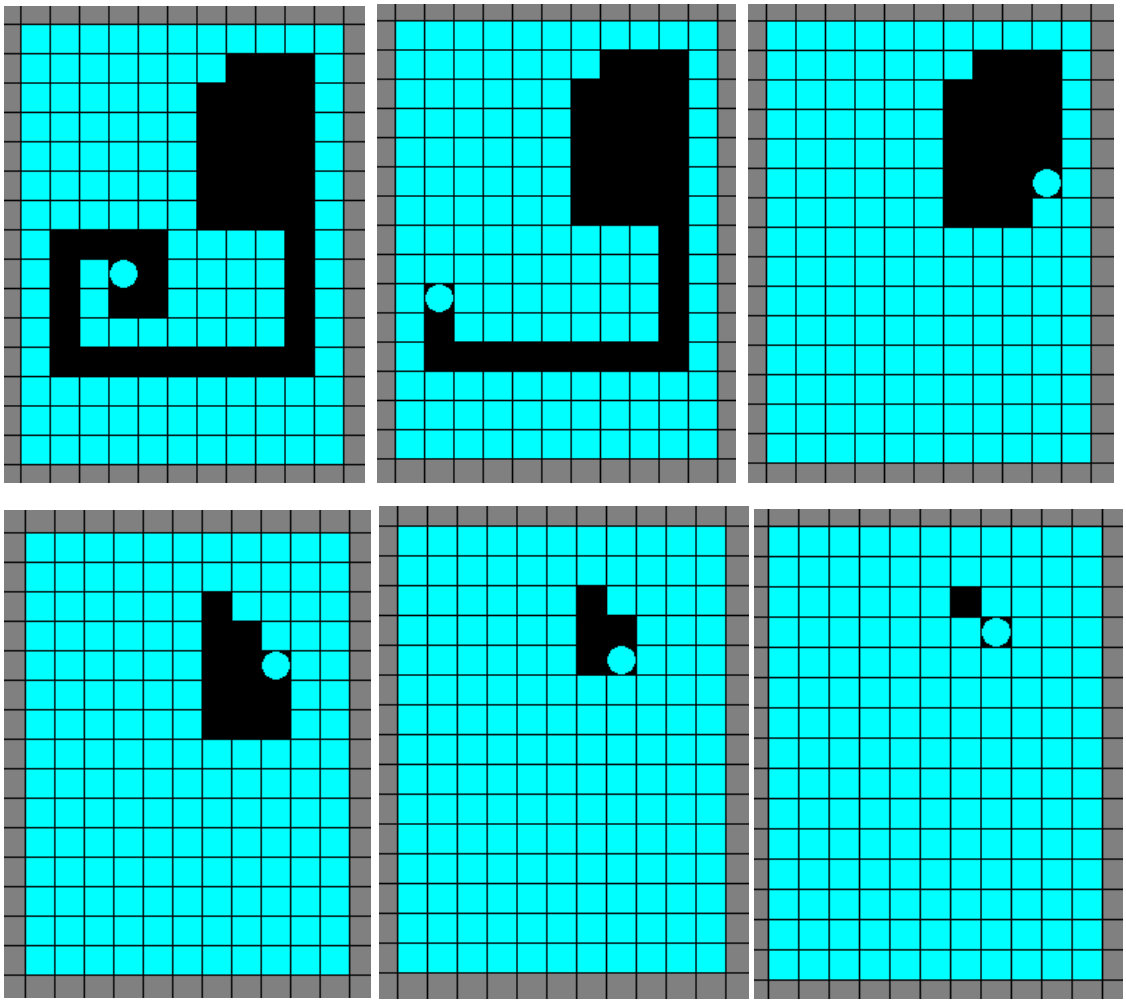
Le score moyen passe à 120, on semble avoir encore progressé. Parfois, le joueur a un choix à faire : explorer une cavité au risque de s'y enfermer ou l'abandonner pour rester dans la zone plus « ouverte ». Mais construire de manière aléatoire un chemin entrant et ressortant d'une cavité semble statistiquement 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 !

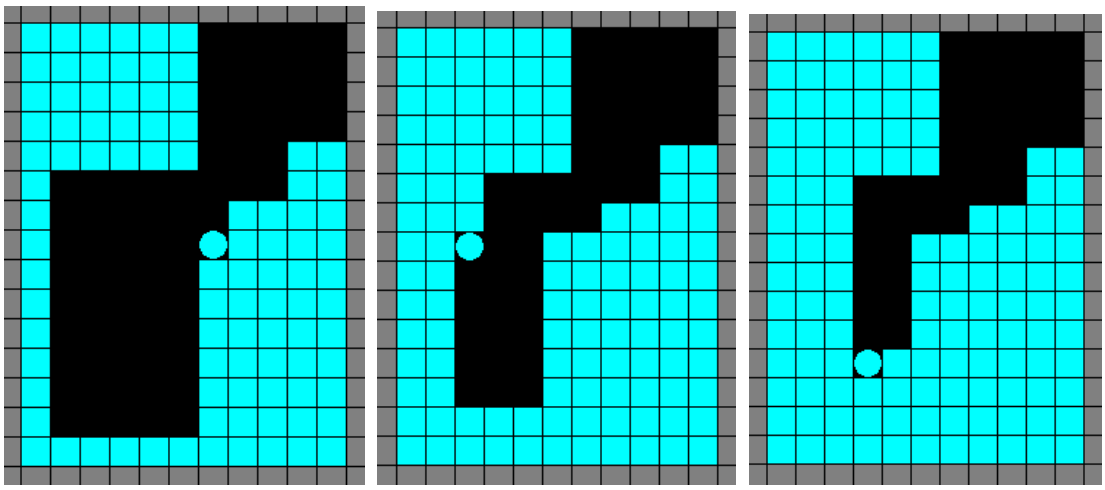
Pour 10 000 parties simulées

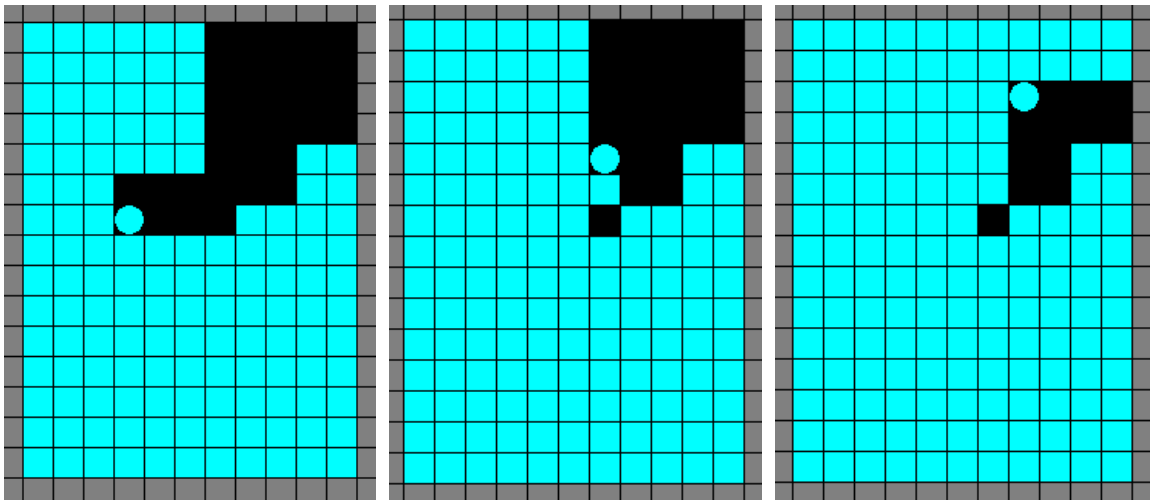
Nous vous présentons le déroulement d'une partie :

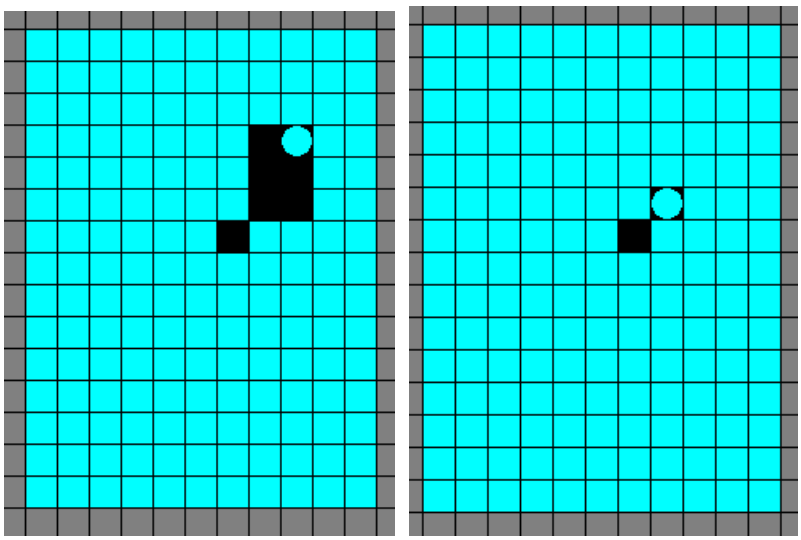




Le comportant de l'IA a véritablement changé. Elle effectue des parcours plutôt en horizontal / vertical pour effectuer le remplissage. Si deux zones à explorer se forment, elle va laisser un chemin assez large (2 cases) pour aller explorer la première zone et revenir par ce chemin pour ensuite explorer 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 librairies 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 librairies, 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 librairies disponibles permet de couvrir 95% des besoins. Développer autours 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 boite 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 fournir par la fonction time() n'est pas précis du tout, il fonctionne par pas de 1/16ème de seconde. Donc même si vous 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 opérations vectorielles sur ces données. Qu'est-ce qu'une opération vectorielle ? Dans une opération arithmétique classique nous ajoutons deux variables a et b représentant des nombres. Ainsi si a et b valent respectivement 3 et 7, le résultat de $a + b$ vaut 10. Une opération vectorielle va consister à écrire dans le code une seule opération mais cette opération va être effectuée sur plusieurs éléments. Ainsi si A et B représentent cette fois des vecteurs, l'opération A + B va donner comme résultat :

$$\begin{vmatrix}5\\6\\7\end{vmatrix} + \begin{vmatrix}10\\20\\30\end{vmatrix} = \begin{vmatrix}15\\26\\37\end{vmatrix}$$

Pour comprendre l'intérêt d'une opération vectorielle, il faut imaginer que les vecteurs présents contiennent un million de nombres. Ainsi, la librairie qui effectue l'opération vectorielle A + B décide comment elle va l'optimiser :

- En utilisant le multithreading : elle peut utiliser les différents cœurs du processeur pour paralléliser le calcul.
- Elle peut utiliser les instructions AVX2 pour effectuer 4 additions en 1 cycle d'horloge
- Elle peut utiliser le GPU pour paralléliser les calculs

Si vous n'utilisez pas l'écriture vectorielle mais que vous écrivez une boucle :

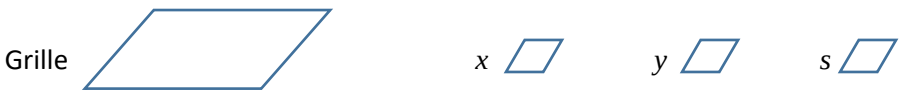
```
Pour i = 0 à n-1

      T[i] = a[i] + b[i]
```

Aucune de ces trois optimisations ne peut se déclencher. De plus, en l'écrivant en script Python, les performances ne seront pas du tout au rendez-vous. En utilisation une opération vectorielle de Numpy, cela permet de faire appel à du code C qui a été écrit de manière ultra-optimisé.

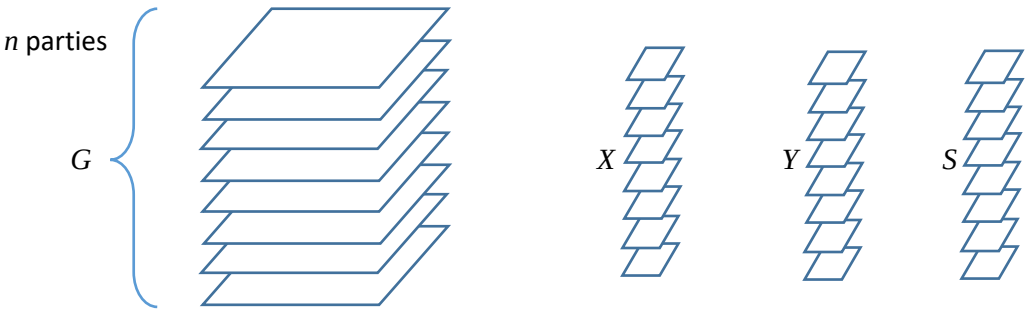
Nous allons laisser de côté le code mis en place dans Tron.py. Vous allez maintenant ouvrir le fichier Tron Parallel.py. Tout le travail qui suit sera maintenant implémenté dans ce fichier. Nous allons mettre en place le traitement parallèle des simulations avec Numpy. Uniquement une fois votre code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

En début de simulation, les paramètres de partie sont : la grille de jeu, la position (x,y) du joueur et le score s :

Partie :



Pour gérer n parties en vectoriel, nous allons dupliquer n fois ces informations. La grille de partie est un tableau à deux indices, en empilant plusieurs grilles, nous obtenons ainsi un tableau à trois indices. La position x du joueur est une valeur, en empilant plusieurs fois cette information, nous obtenons un vecteur X. De la même manière, nous construisons un vecteur Y et un vecteur S :



Remarque : nous utilisons indifféremment les termes tableau et vecteur. En effet, ces deux termes désignent des containeurs de données avec 1 indice. Nous n'avons pas de notions de vecteur vertical ou horizontal comme en algèbre linéaire.

La tableau tridimensionnels G est indicé de la manière suivante : G[id_partie][x][y], ainsi en écrivant G[4] vous accédez à la grille de jeu de la 5ième partie. Idem, en écrivant X[4] et Y[4] vous accédez aux coordonnées du joueur de la 5ième partie.

La création des différents tableaux G, X, Y et S est effectuée par la fonction tile que nous présentons ci-dessous. L'import numpy as np permet d'utiliser toutes les fonctions de la librairie Numpy en écrivant np.ma_function(…) :

```
import numpy as np

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

Nous présentons une syntaxe vectorielle de Numpy très utile pour le projet. Si nous avons un tableau Numpy T à 3 dimensions et 4 vecteurs A, B, C et D de taille k, alors la syntaxe vectorielle

T[A, B, C] = D

Se lit en code classique :

```
Pour i = 0 à k-1

      T[A[i],B[i],C[i]] = D[i]
```

Remarque : on peut aussi utiliser l'écriture suivante T[A, B, C] = 4.

Nous présentons un autre exemple pour expliquer cette syntaxe qui peut sembler déroutante au premier abord mais qui en fait sert simplement à effectuer une série d'affectations :

```
T = np.zeros((9,9), dtype=np.int32)
T[ 0 , 1 ] = 10
T[ 2 , 5 ] = 19
T[ 6 , 7 ] = 12
T[ 4 , 5 ] = 16
```

```
T = np.zeros((9,9), dtype=np.int32)
A = np.array( [ 0,  2,  6,  4] )
B = np.array( [ 1,  5,  7,  5] )
C = np.array( [10, 19, 12, 16] )
T[A,B] = C
```

Nous pouvons par exemple utiliser cette syntaxe pour mettre un mur à la position (x,y) du joueur. Ainsi, la i-ème valeur stockées dans le vecteur X et le vecteur Y représentent la position (x,y) du joueur dans la i-ème partie. Le code classique que nous écririons serait :

```
Pour i = 0 à nb_parties-1

      G[i,X[i],Y[i]] = valeur_mur
```

On ne peut pas directement transcrire ce code dans la syntaxe vectorielle vue précédemment car l'indice *i* ne représente pas un vecteur. Idéalement, il nous faudrait un vecteur I contenant les valeurs prises par l'indice i : 0, 1, 2, 3… nb_parties-1. Si nous en avons besoin, il suffit de le créer et de le remplir correctement en utilisant une boucle. Ainsi, nous pourrons écrire le code suivant :

```
G[I, X, Y] = valeur_mur
```

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, ces fonctions prennent la dimension du tableau sous forme d'un tuple et aussi le type des données. Nous utilisons des np.int32. Si vous omettez cet attribut, attention, car Numpy crée par défaut des tableaux de flottants et un flottant ne peut être utilisé comme indice d'un tableau.

Nous allons gérer plusieurs parties en parallèle. Certaines parties vont finir avant d'autres. Pour faire en sorte de ne traiter que les parties en cours, il faudrait faire des tests supplémentaires avec des conditions, ce qui n'est pas forcément performant. Nous préférons en fait traiter toutes les parties comme des parties actives et ajouter une direction supplémentaire correspondant à une nouvelle direction « sur place » utilisée par les parties achevées. Ainsi, les joueurs coincés entre 4 murs vont se déplacer « sur place ». Nous aurons 5 directions possibles : sur place, à gauche, en haut, à droite et en bas. Nous stockons dans les vecteurs dX et dY les déplacements associés aux cinq directions, l'indice 0 correspondant à la direction sur place :

```
dX = np.array([0, -1, 0,  1,  0],dtype=np.int32)
dY = np.array([0,  0, 1,  0, -1],dtype=np.int32)
```

Le vecteur Choix correspondant au n° de la direction choisie dans chaque partie. Par exemple, si Choix est le vecteur [2,3,4], cela veut dire, d'après le contenu des vecteurs dX et dY, que le déplacement dans la partie 0 est (0,1), dans la partie 1 : (1,0) et dans la partie 2 : (0,-1). Pour l'ensemble des parties, nous pouvons construire deux vecteurs DX et DY correspondant au déplacement en utilisant la syntaxe :

```
DX = dX[Choix]
DY = dY[Choix]
```

Le code équivalent à cette syntaxe se lit :

```
DX[0] = dX[Choix[0]]
DX[1] = dX[Choix[1]]
DX[2] = dX[Choix[2]]
…
```

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 déterminer 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 dans un vecteur de taille nb_parties-1. 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 :

```
Vgauche = (Vgauche == 0) * 1
```

Le test == transforme le vecteur Vgauche en vecteur de booléens. L'opération *1 transforme les valeurs True en 1 et les valeurs False en 0.

Nous allons maintenant créer *nb_parties* listes servant à stocker les directions possibles pour chaque partie. Pour cela, nous allons créer un tableau de dimension *nb_parties*x4 pouvant stocker au maximum 4 directions possibles pour chaque partie. Nous aurons aussi un vecteur Tailles stockant la

taille de chaque liste, accessoirement, la taille de chaque liste correspond aussi à l'indice où sera inséré le prochain élément dans la liste. Voici un exemple :

LPossibles	Tailles	
1 2 4 0	3	# trois directions possibles : 1 2 et 4
2 3 0 0	2	# deux directions possibles : 2 et 3
4 0 0 0	1	# une direction possible : 4

Voici le code nécessaire pour créer et initialiser ces deux tableaux :

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

Prenons un exemple où nous avons déjà traité les trois premières directions. Nous passons maintenant à la dernière. L'analyse des déplacements vers le bas génère un vecteur Vbas rempli de 0 et de 1. Voici le tableau LPossibles et le vecteur Tailles, avant et après l'insertion :

LPossibles	Tailles	Vbas		LPossibles	Tailles
1 2 0 0	2	1		1 2 4 0	3
2 3 0 0	2	0	=>	2 3 0 0	2
0 0 0 0	0	1		4 0 0 0	1

La syntaxe T[A, B] = C permet de réaliser cette transformation. Le vecteur Tailles permet d'obtenir la position où doit s'effectuer l'insertion. Le vecteur C est obtenu en multipliant le vecteur Vbas par la valeur correspondante. Il faut ensuite mettre à jour le vecteur Tailles. Pour cela, il suffit d'utiliser le vecteur Vbas pour obtenir ses nouvelles valeurs.

Une fois les 4 directions traitées, nous interprétons l'exemple précédent ainsi :

- Partie 0 : le joueur 0 peut se déplacer dans les directions 1 2 et 4 correspondant aux directions à gauche, en haut et en bas. Comme 3 choix sont possibles, Tailles [0] vaut 3.
- Partie 1 : le joueur a seulement 2 possibilités pour se déplacer : 2 (en haut) et 3 (à droite).
- Partie 2 : le joueur peut uniquement se déplacer vers le bas, ce choix unique implique que la valeur de Tailles [2] soit égale à 1.

Une fois le tableau LPossibles mis à jour, les listes de directions possibles sont disponibles pour chaque partie. Les listes vides ont une taille égale à 0, nous allons les modifier pour qu'elles soient de taille 1, afin que la direction n° 0, équivalent à rester sur place, soit sélectionnée pour ces parties. Nous changeons donc dans notre vecteur toutes les valeurs à 0 par des valeurs 1. Pour cela, nous utilisons la syntaxe suivante :

T[T==v1] = v2 avec v1 et v2 deux valeurs

Le code équivalent à cette syntaxe est le suivant :

```
if  T[0] == v1 :  T[0] = v2
if  T[1] == v1 :  T[1] = v2
if  T[2] == v1 :  T[2] = v2
…
```

Il nous reste maintenant à tirer au sort la direction possible pour chaque liste. Pour cela nous utilisons la syntaxe suivante :

R = np.random.randint(LPossibles)

LPossibles étant un vecteur, le code équivalent à cette syntaxe est le suivant :

```
R[0] = np.random.randint(0,LPossibles[0])
R[1] = np.random.randint(0,LPossibles[1])
R[2] = np.random.randint(0,LPossibles[2])
…
```

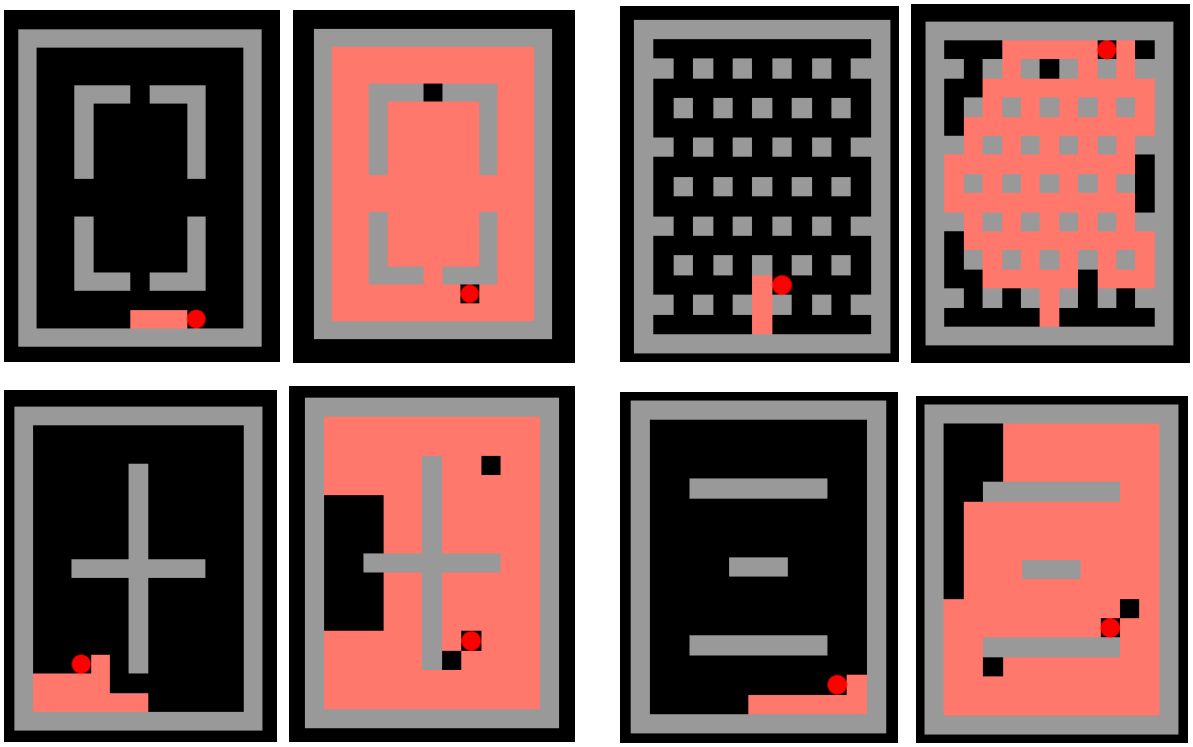
Après cela, il reste à déduire le déplacement choisi pour chaque joueur, à modifier leurs positions et à faire évoluer les scores en fonction. Pour savoir si les simulations sont toutes terminées, vous pouvez sommer les valeurs du vecteur score avec la fonction np.sum(…). En effet, lorsque tous les joueurs font du « sur place » alors la somme des scores n'évolue plus.

Nous vous conseillons de tester votre code et de vous assurer qu'il fonctionne parfaitement en lançant plusieurs simulations.

Intégrez le code que vous avez créé dans le fichier principal tron.py avec un copier-coller. 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 ☺