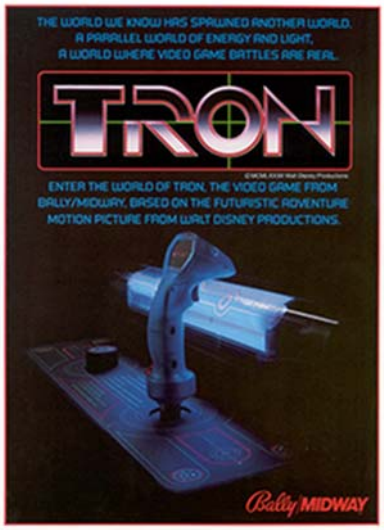
TRON

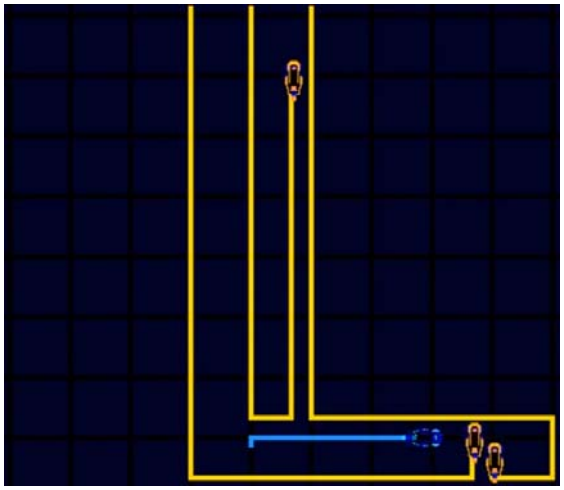
Tron est un jeu vidéo d'arcade commercialisé en 1982 adapté du film sorti la même année. Il a par la suite été porté sur le Xbox Live Arcade et a connu une suite au cinéma : suite « Tron Héritage » réalisée en 2011 par les studios Disney.



Jaquette du jeu 1982

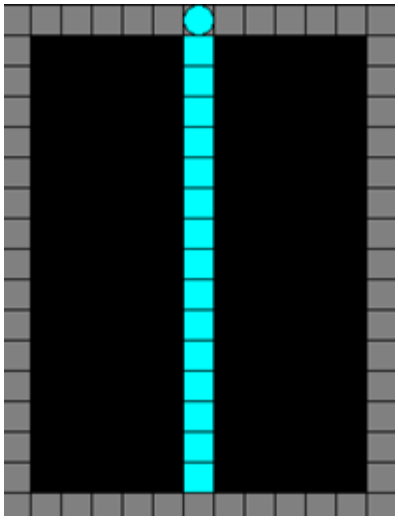
Dual entre motos lumineuses

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.

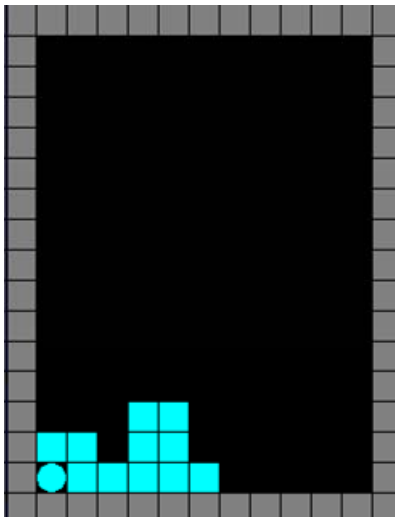


Ouvrez le fichier exemple du projet et lancez-le. Le joueur actuel se déplace uniquement vers le haut jusqu'à rentrer en collision avec le mur de l'arène. A chaque fois que le joueur avance d'une case son score augmente de 1. Nous lançons en tout trois parties, qui vont être identiques. L'intérêt à terme est d'examiner les différents comportements du joueur IA et d'obtenir une sorte de note de qualité de cette IA en moyennant les scores obtenus sur plusieurs parties.

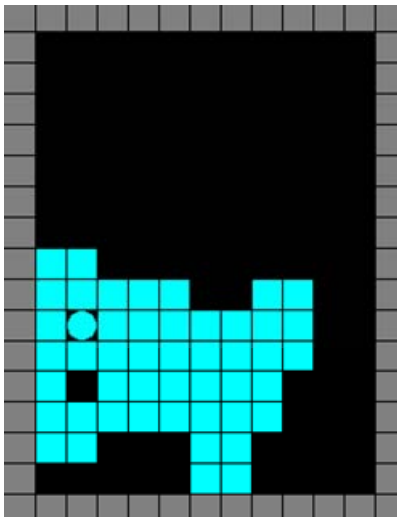
Partie 1 – Jeu aléatoire

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 bas, 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.



Une partie qui n'a pas eu de chance !



Une partie intéressante mais qui n'a pas exploité tout le potentiel disponible

Lancez les parties et examiner le score obtenu.

Un peu d'histoire – pas si veille l'histoire !!

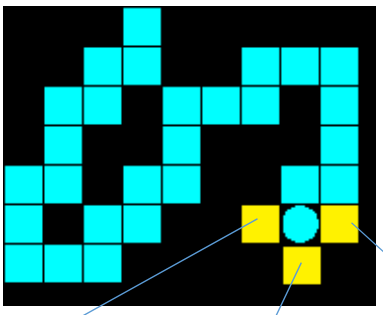
L'algorithme que nous vous proposons d'étudier est une version simplifiée de l'algorithme MCTS (Monte Carlo Tree Search). Cette approche type Monte Carlo fut le centre des premières versions du programme AlphaGo mis en place par l'entreprise Google DeepMind en 2012. En octobre 2015, AlphaGo devient le 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. AlphaGo Zero en octobre 2017 atteint un niveau supérieur en jouant uniquement 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 sociologiquement.

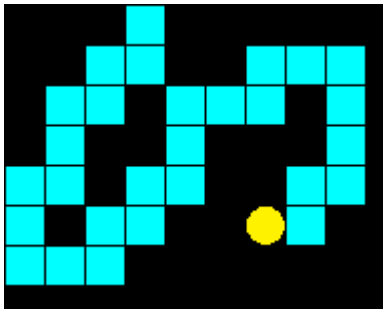
Partie 2 – Algorithme de type Monte Carlo

Cet algorithme n'a pour connaissance que la grille de jeu actuelle et la position du/des joueurs. Il ne tient pas compte de comment les coups précédents ont été effectués. Comment va-t-il prendre sa décision ? En premier lieu, il liste l'ensemble des coups possibles depuis la grille actuelle. Il va alors étudier chacun d'entre eux et choisir le meilleur. Pour cela, lorsqu'il étudie un coup à jouer, il va alors simuler un très grand nombre de parties, ces parties démarrant depuis le coup choisi. Il va récupérer le score de chacune de ces simulations et les moyenner pour obtenir la note associée au coup étudié. Il suffit alors de choisir la meilleure note sur l'ensemble des coups possibles et de jouer ce coup précis.

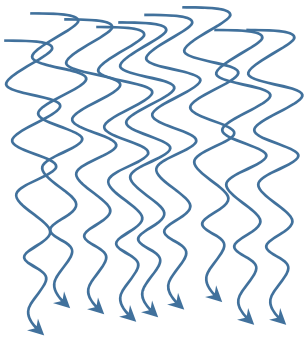
Etat de la grille de jeu et position du joueur



Option numéro 1



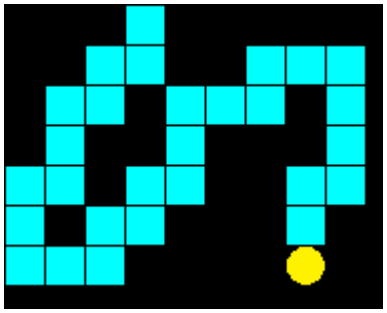
Simulation de milliers de parties indépendantes à partir de ce point



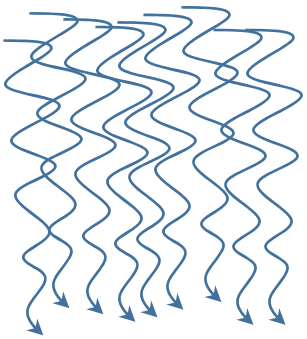
17 42 55 43 62 71 79

Score Moyen : 72.4

Option numéro 2



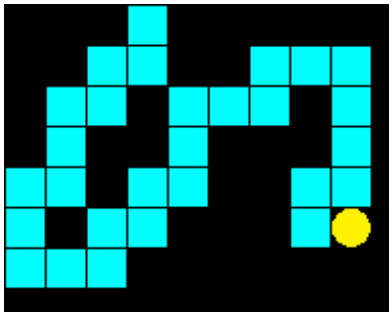
Simulation de milliers de parties indépendantes à partir de ce point



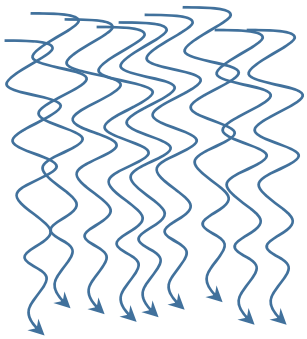
23 33 45 55 43 66 54

Score Moyen : 44.5

Option numéro 3



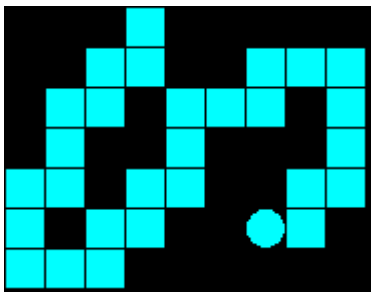
Simulation de milliers de parties indépendantes à partir de ce point



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 la simulation de parties. En effet, nous avons besoin d'une fonction qui permet à partir d'une grille donnée et d'une position de retourner la liste des directions possibles pour la position considérée. Ainsi cette fonction doit recevoir en paramètre une grille et deux coordonnées x et y indiquant la position du joueur. La grille utilisée pour la simulation est une copie de la grille principale. Elle va servir de « brouillon » pour dérouler cette partie puis elle sera oubliée à la fin de cette simulation. A ce niveau, aucune simulation ne doit modifier la grille principale de jeu.

Ensuite, nous allons écrire la fonction de simulation qui à partir d'une grille et d'une position de jeu joue une partie au hasard. Cette fonction, une fois la partie simulée terminée, retourne la quantité de cases parcourues :

```
SimulationPartie (GrilleTemp,x,y) :
   Boucle Infinie
        L = DirectionsPossibles(GrilleTemp,x,y)
        Si L est vide Retourner le nombre de cases parcourues
        Choisir une direction au hasard
        Déplacer le Joueur
        Créer le mur
```

L'approche de Monte Carlo consiste à simuler plusieurs parties pour une position de jeu donnée et à retourner la somme des scores obtenus :

```
MonteCarlo(Grille,x,y,nombreParties)
      Total = 0
      Pour i allant de 0 à nombreParties
          Grille2 = Copie(Grille)
          Total += SimulationPartie(Grille2,x,y)
      Retourner Total
```

Pour effectuer une copie de la grille de jeu et envoyer cette grille à la simulation de partie, veuillez utiliser la fonction :

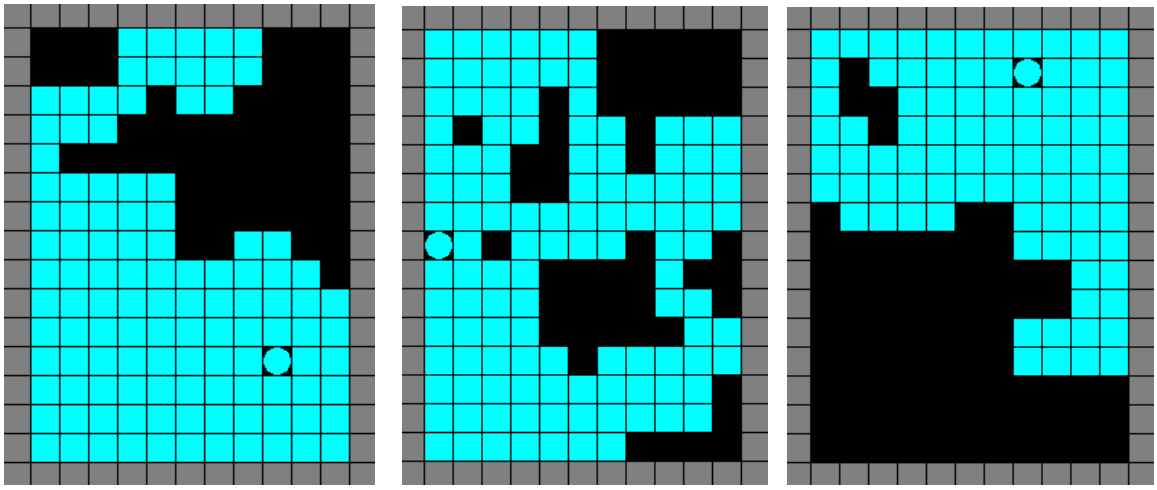
```
Import copy

CopieGrille = copy.deepcopy(MaGrille)
```

Grâce à cette opération, CopieGrille et MaGrille sont deux objets qui contiennent des informations identiques MAIS qui existent indépendamment l'un de l'autre, comme des jumeaux. Si l'on modifie l'un, il n'y a pas de report des modifications dans l'autre objet.

Il vous reste à modifier la boucle de jeu principale pour récupérer les scores associés à chaque option. Une fois le meilleur score identifié, le joueur est déplacé vers la case correspondante.

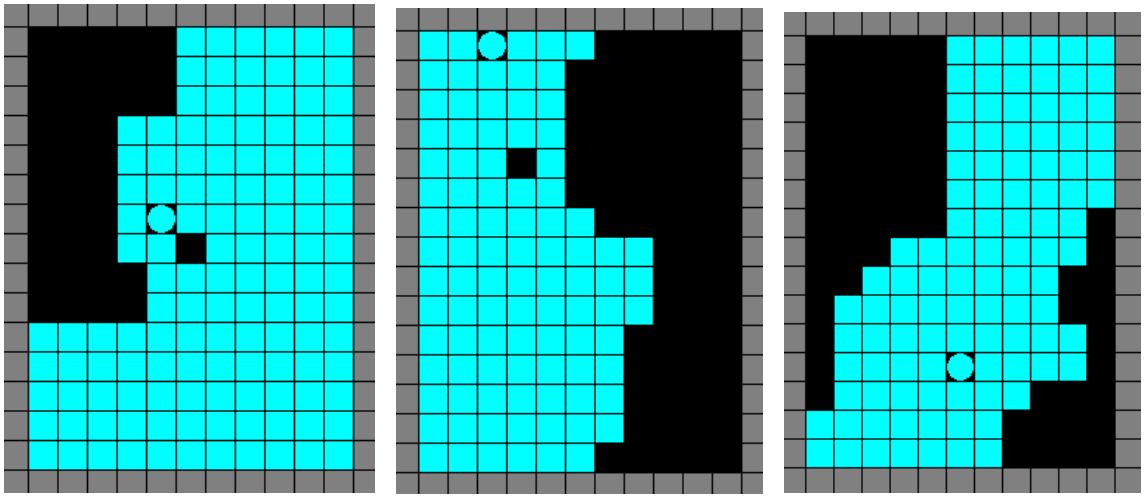
Voici les résultats obtenus avec l'algorithme de Monté Carlo

Pour 10 parties simulées



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

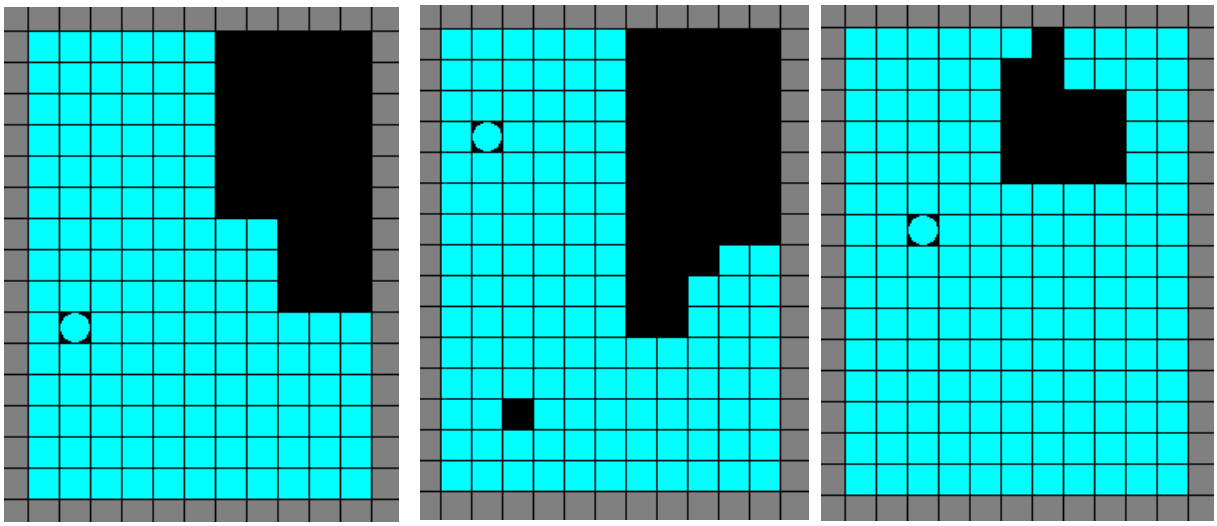
Pour 100 parties simulées



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

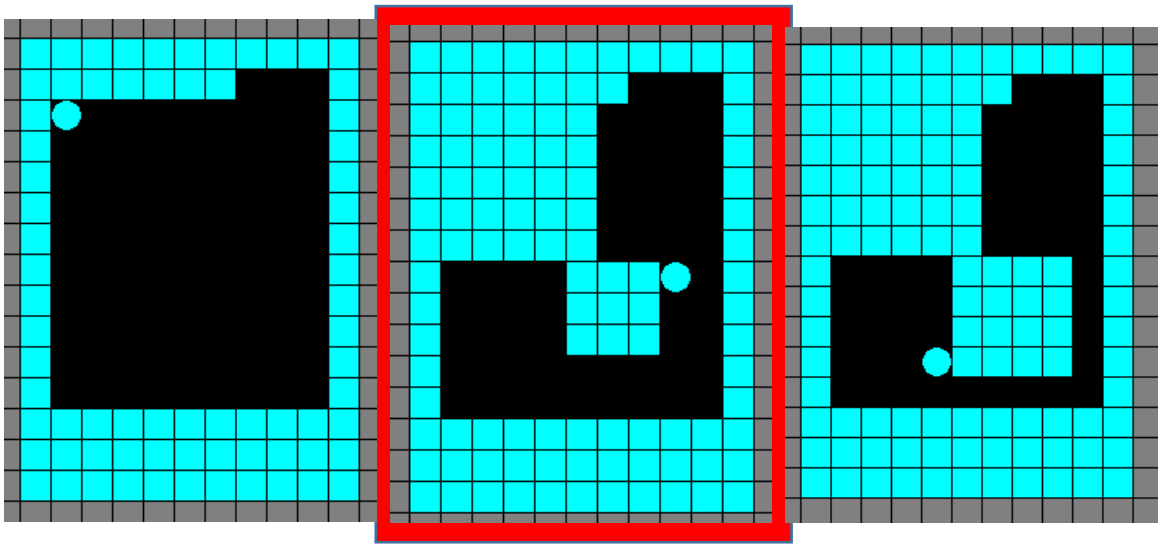
Ça commence à ralentir ☺

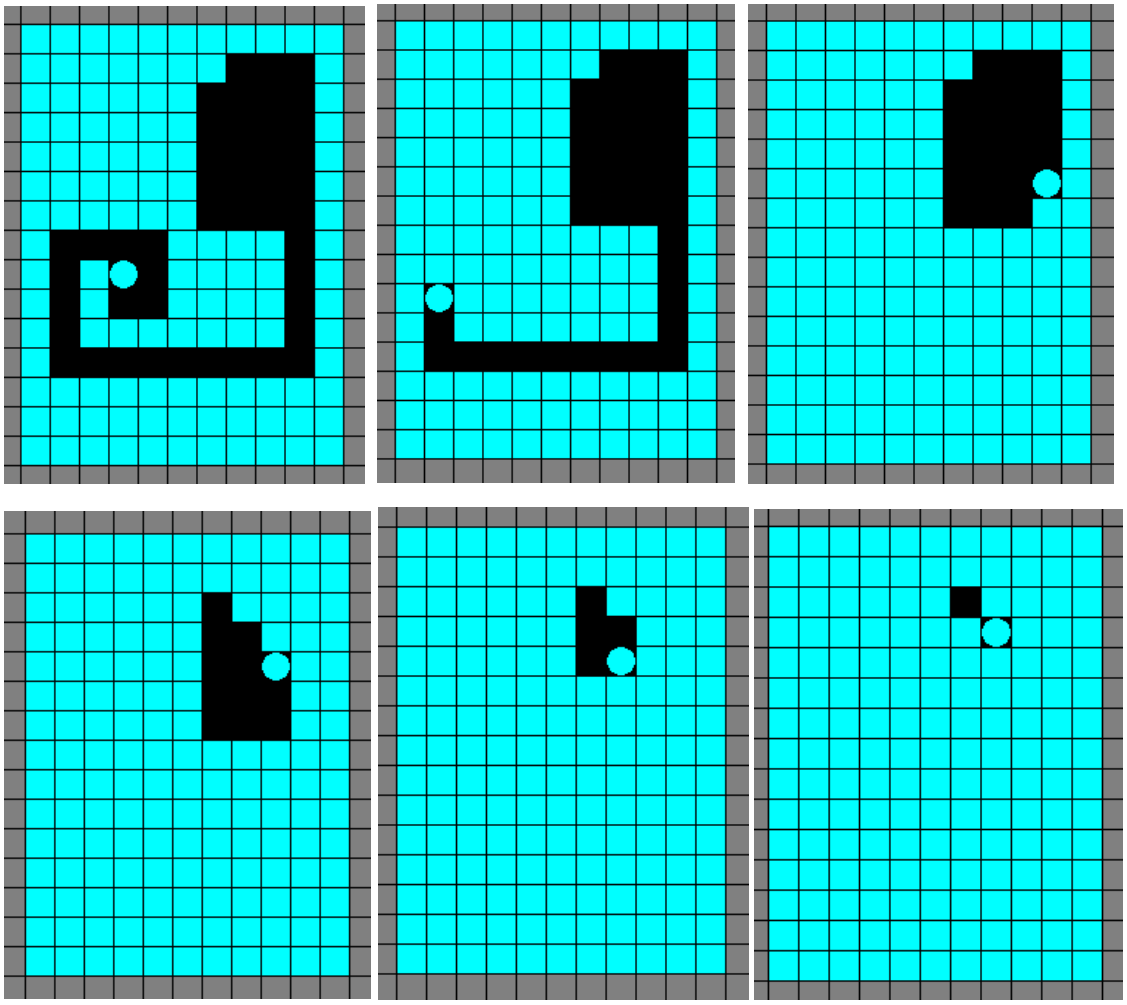


Le score moyen passe à 120, on semble avoir encore progressé. L'algorithme contient encore un défaut, il a peur des cavités. Une fois 40% à 60% du terrain rempli, le joueur se retrouve parfois dans la situation suivante : il a un choix à faire entre entrer dans une cavité ou explorer la zone restante plus « ouverte ». Il va alors choisir d'éviter la cavité car construire un chemin entrant et ressortant de cette cavité semble statistiquement difficile.

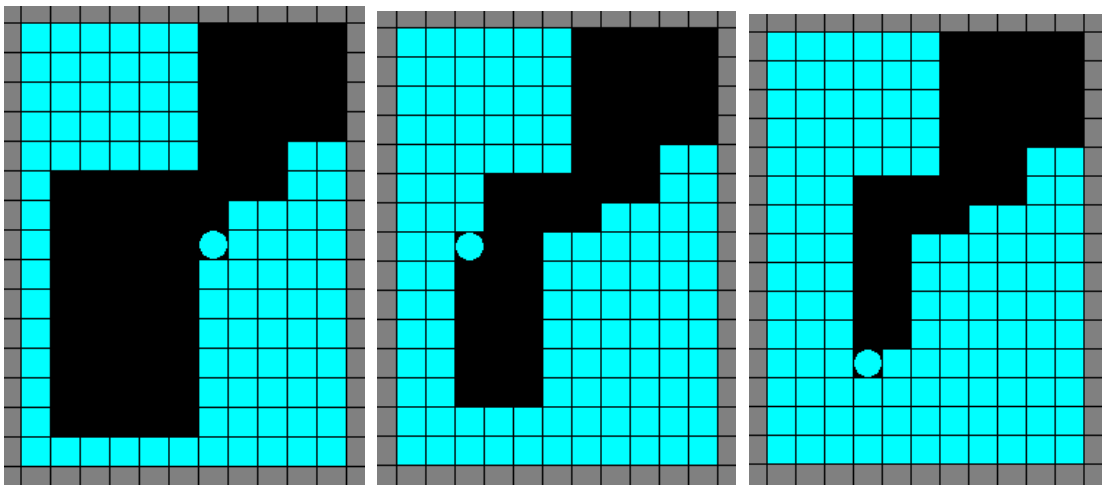
Pour 10 000 parties simulées

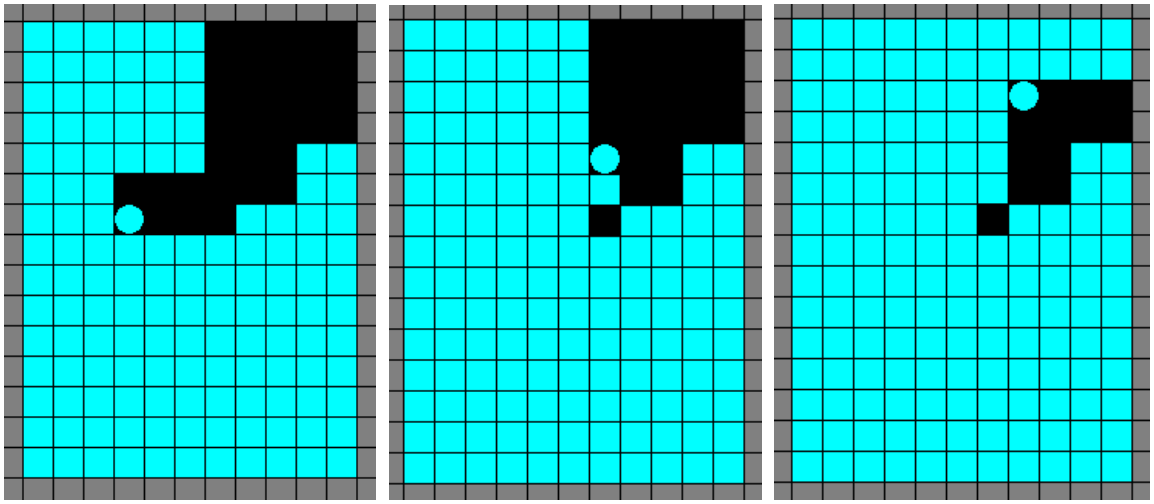
Nous vous présentons le déroulement d'une partie :

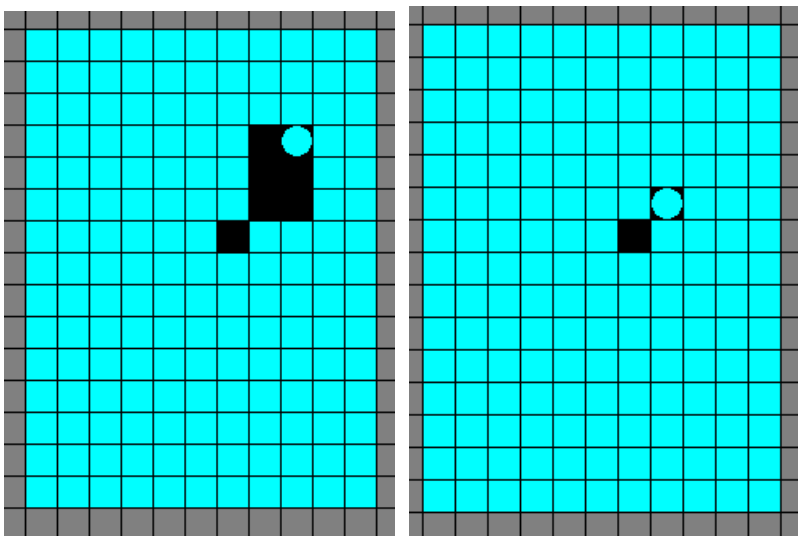




Le comportant de l'IA a véritablement changé. Elle effectue des parcours plutôt en horizontal / vertical pour effectuer le remplissage. Si deux zones à explorer se forment, elle va laisser un chemin assez large (2 cases) pour aller explorer la première zone, et revenir par ce chemin pour ensuite explorer la seconde. Voir dessin encadré en rouge. Le parcours est parfait, au sens où il ne reste plus qu'une case inexplorée. 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.

Partie 3 – Geek Time !

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, c'est-à-dire un langage assez évolué, souple, simple qui permet d'appeler des routines externes exécutant des tâches complexes de manière optimisée.

Avec le temps, il s'avère que ce choix de développement s'avère judicieux. Le langage Python est simple, flexible et intuitif, il permet de coder vite et bien, mais il est lent. Ce n'est pas important, si en parallèle, on dispose de packages contenant les fonctions requises optimisées avec grands soin. Cela est même avantageux, car comme l'environnement Python permet un import facile des packages externes, on peut en 5 secondes avoir accès à des librairies haut de gamme alors que cet import aurait nécessité 2/3/4 jours de configuration / installation / compilation en C++ par exemple.

Le seul problème survient lorsque l'on doit développer en Python du code faisant beaucoup de calculs et qu'aucune librairie ne répond à notre besoin. Dans ce cas, on se retrouve dans la zone d'ombre du langage, devant faire tourner du code à une vitesse au moins 10 à 50 fois plus lente comparée à du langage C/C++/Java.

Cependant, la communauté Python, très active, a pensé à tout. Nous allons utiliser une facilité fournie par la librairie NUMBA. En mettant juste le tag @jit devant les fonctions cruciales, NUMBA va les intercepter, les convertir en C, les compiler de manière optimisée et les réintégrer dans votre code Python, et sans rien vous demander de plus !!! Formidable !!!

Cependant, il y a certaines précautions à prendre. Pour fonctionner, NUMBA doit comprendre les données externes (la grille par exemple) qu'il utilise ainsi que les fonctions fournies par les packages externes :

- Pour la compatibilité avec les packages, c'est très simple, soit le package est supporté par NUMBA (comme dans le cas de random) soit il ne l'est pas (comme dans le cas de deepcopy) et il faudra recoder la fonction en python.
- Pour les données, il faut que NUMBA arrive à typer les données qu'il reçoit. Si vous ne l'aidez pas, il risque de ne pas prendre le type le plus avantageux pour optimiser les performances.

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 5 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 et liste 2D

Numba ne sait pas gérer les listes 2D du langage Python. Par contre, il existe une librairie appelée Numpy que Numba sait manipuler parfaitement. Ajoutez : import numpy en début de programme et ensuite changer le type de la grille en tableau 2D numpy :

Grille = numpy.zeros((LARGEUR,HAUTEUR))

La fonction deepcopy n'est pas comptabile avec Numba, nous allons donc la changer pour son équivalent numpy :

Grille2 = numpy.copy(Grille)

Lancez maintenant votre nouvelle version. Pour 10 000 simulations, vous devriez obtenir un temps d'attente de l'ordre de 3 secondes. Le bénéfice de Numpy s'est fait ressentir !

Etape 3 : Numba !!

En début de programme, ajoutez le code suivant :

```
Import numba
from numba import jit
```

Ajouter le tag @jit sur la ligne précédent les fonctions à optimiser :

```
64
65  @jit
66  def toto(...)
```

On parle ici de :

- La fonction qui sélectionne les directions possibles
- La fonction qui simule au hasard une partie
- La fonction de Monte Carlo qui lance et moyenne les résultats de plusieurs simulations

A ce niveau, vous devriez, si tout se passe bien, avoir un gain de temps d'un facteur 10. Lancez maintenant votre nouvelle version. Pour 10 000 simulations, vous devriez obtenir un temps d'attente de l'ordre de 250 millisecondes !!! Hip hip hip !!! C'est l'occasion d'augmenter le nombre d'itérations.

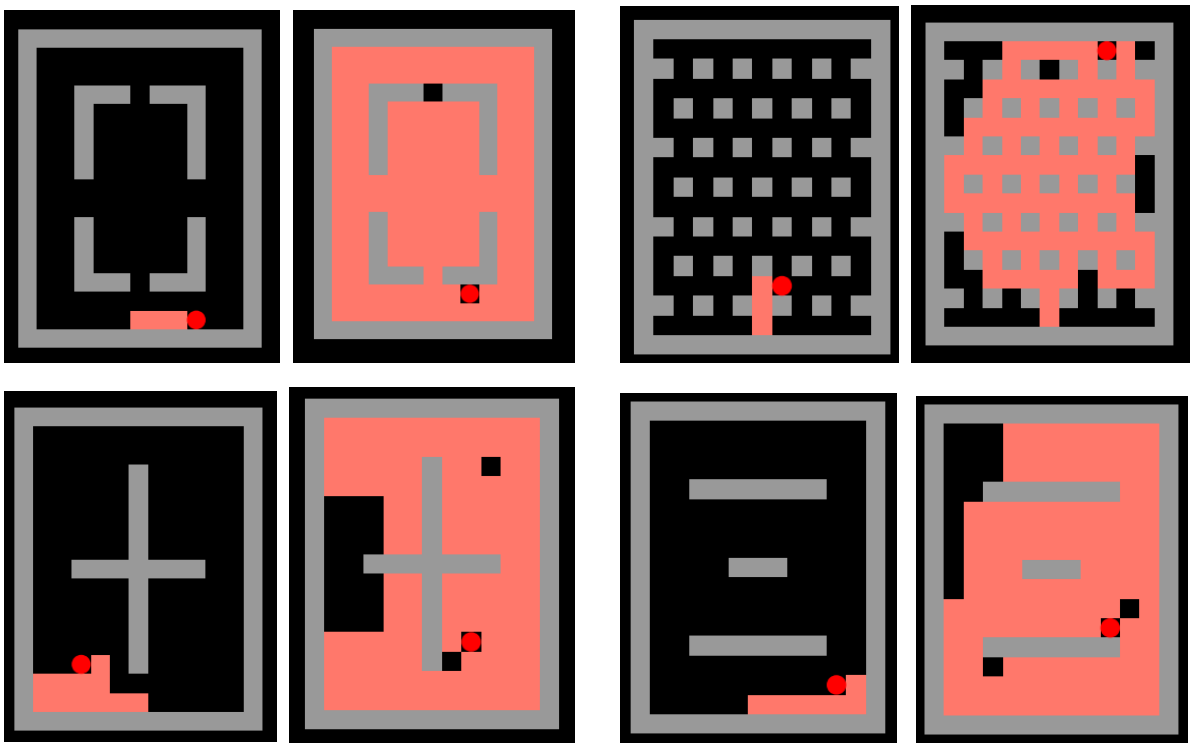
CONSEILS :

- Pour que l'accélération Numba fonctionne, il faut que la fonction accélérée n'utilise pas de variables globales à l'intérieur de la fonction. Toutes les données doivent donc passer en arguments.
- Les listes [] peuvent poser problème. En effet, les listes Python peuvent contenir des types hétérogènes comme un nombre puis une chaîne de caractères. Numba lui préfère des numpy array qui sont de type connu et uniforme. Pour la liste des coups possibles, vous pouvez donc

utiliser un numpy array de 9 ints, le premier indiquant le nombre de coups possibles et les cases suivantes servant à stocker la liste des 4 directions, par exemple.

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 ☺