Algorithmes de Planification pour le Jeu de Karuba

Projet tutoré par M. Bruno Zanuttini
Dylan Jubault
Romain Ambroise

Année 2018 – 2019 *M1 Informatique*

Sommaire

I - Cadre du Projet

Dans le cadre de la première année de Master Informatique de l'UFR de Sciences de Caen, nous avons été affilié au projet « Intelligence Artificielle pour le jeu du Karuba » sous la tutelle de Monsieur Bruno Zanuttini. L'objectif était dans un premier temps de réaliser une tâche technique : le développement du jeu. Puis, il était d'effectuer des recherches sur les algorithmes de planification tels que l'algorithme de Monte Carlo et la planification géométrique.

Notre groupe a évolué durant le projet, nous étions au début un trinôme. La technologie adoptée était alors le Python. Lorsque le groupe est devenu un binôme, il ne nous restait plus que quelques semaines c'est pourquoi nous avons décidé de traduire le projet en Java afin de gagner du temps et d'assurer la réalisation de nos objectifs.

Nous n'avons pas rencontré de réelles difficultés durant la réalisation des livrables. Nous nous sommes fixés des objectifs très précis ce qui nous a permis de bien gérer le projet.

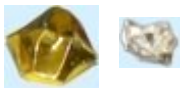
Nous tenons à remercier Monsieur Bruno Zanuttini pour sa disponibilité et son implication. Ses interventions nous ont assuré une parfaite compréhension du sujet et nous ont permis d'évoluer sereinement dans ce projet.

II - Le Jeu du Karuba

1 - Les Règles du Jeu

Karuba est un jeu de plateau pouvant se jouer de 2 à 4 joueurs. La grille du plateau est de 8 cases par 7.

Chaque joueur possède quatre aventuriers de couleurs différentes (bleu, violet, marron et orange). L'objectif est d'amener ces quatre aventuriers aux quatre temples de la couleur correspondante afin d'accumuler des trésors présents sur la route (pépites d'or et diamants) et dans les temples (trésors de temple).



Utilisation des Tuiles

Pour se faire, chaque joueur devra construire des routes à l'aide des 36 tuiles de la pioche. Ces tuiles possèdent des chemins compatibles entre elles.

Exemple de Tuiles

Toutefois, certaines règles sont à respecter quant à l'utilisation de ces tuiles :

- Une tuile doit être placée sur une case inoccupée.
- Le numéro de la tuile doit toujours être dans le coin supérieur gauche (pas de rotation des tuiles).

Au début de la partie, les joueurs choisissent à tour de rôle, l'emplacement de départ des aventuriers et l'emplacement des temples.

Déroulement d'un Tour de Jeu

Au début de chaque tour, le joueur pourra choisir :

- D'utiliser la tuile tirée au sort par le chef d'expédition.
- De la mettre de côté afin de pouvoir déplacer un aventurier au choix.

L'aventurier peut se déplacer d'un nombre de case correspondant au nombre de chemins qui partent de sa tuile de départ.

Les Trésors de Temple

Lorsqu'un aventurier arrive en premier au temple de sa couleur correspondante, il récupère le trésor valant le plus de points (5 points). L'aventurier suivant qui y arrivera récupérera la valeur du dessous (4 points) et ainsi de suite jusqu'à 2 points.

Attention, si les aventuriers arrivent durant le même tour au temple, ils récupèrent le trésor valant le même nombre de points.

Condition de Fin de Partie

La partie se termine lorsque :

- L'un des joueurs a mené ses aventuriers aux 4 temples.
- La dernière tuile a été piochée.

Comptabilisation des Points

- Un trésor vaut sa valeur imprimée sur son pion.
- Un pépite d'or vaut 2 points.
- Un diamant vaut 1 point.

2 - Adaptation du Jeu

Lorsque l'on participe à une partie de Karuba, l'objectif principal est d'amener ses aventuriers le plus vite aux temples. C'est pourquoi dans le but d'obtenir une IA ciblé sur cet objectif nous avons décidé de supprimer les règles suivantes :

- Les aventuriers et les temples seront placés aléatoirement sur la carte (pas de choix des joueurs).
- Les pépites d'or et diamands sont enlevés du jeu afin de se focaliser sur les trajectoires pour aller vers les temples.
- Il n'y a pas de chef d'expédition, les tuiles sont tirées aléatoirement à partir de la pioche.

Le jeu se déroulera uniquement avec 4 joueurs :

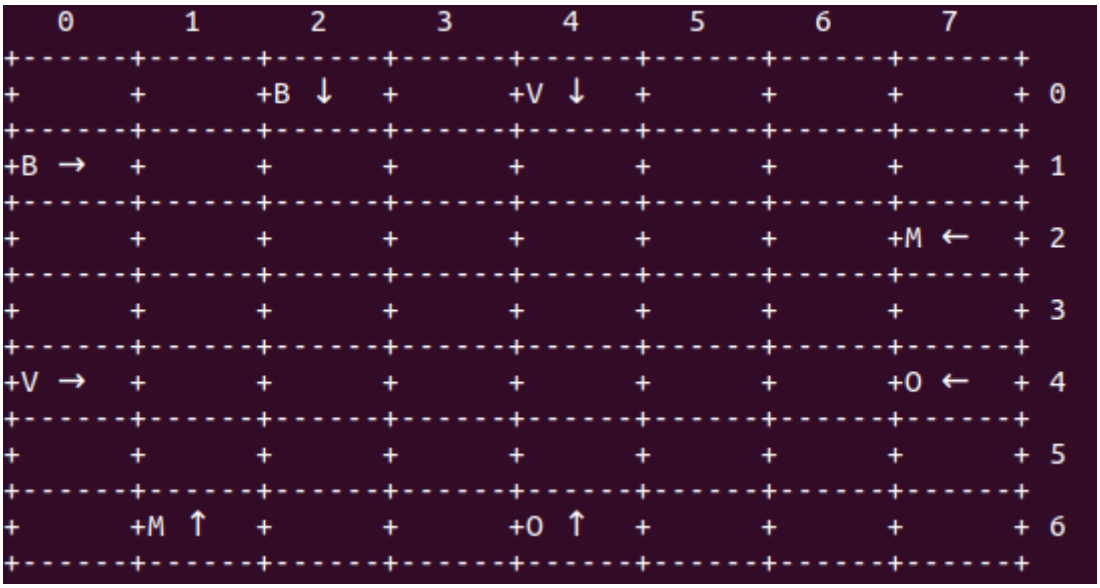
- Soit 1 humain contre 3 IA
- Soit 4 IA

III - Implémentation du Karuba

Pour réaliser cette tâche technique, nous avons décidé d'intégrer le patron de conception Modèle-Vue-Contrôleur. Nous avons anticipé le développement de l'algorithme de Monte Carlo en permettant à l'application de pouvoir cloner un état de jeu. Ainsi, à tout instant du jeu, nous pouvons dupliquer cet état, le modifier et accéder à ses états fils. Nous avons également permis à un état de transmettre une liste exhaustive de tous les coups qui sont réalisables à partir de cet état.

La classe Métier n'interagit que très peu avec le jeu, elle est utile pour initialiser l'état initial, gérer la pioche et gérer les conditions de fin de partie. Pour gérer les tours de jeu, nous avons décidé d'attribuer cette tâche à la classe Contrôleur qui interrogera les joueurs humains en récupérant les entrées clavier.

Nous avons décidé de réaliser une interface graphique simple, car l'objectif de ce projet était d'obtenir une solution simpliste afin de permettre à l'algorithme de Monte Carlo de s'épanouir. Ainsi, nous représentons le plateau à l'aide d'une chaîne de caractères comme le montre l'image suivante :



Les flèches directionnelles représentent les directions que peut prendre un aventurier pour se déplacer à partir d'une case.

Les lettres B, V, M, O représentent respectivement les couleurs bleu, violet, marron et orange d'un temple ou d'un aventurier. Les temples ne bougent pas et seront toujours placés sur la ligne 0 ou sur la colonne 7. Ainsi, le joueur peut distinguer ses aventuriers des temples mêmes si ils sont représentés par le même caractère.

Pour faciliter l'exécution de l'algorithme de Monte Carlo, le jeu peut se lancer de un joueur à 4 joueurs. Le joueur peut être une intelligence artificielle dans ce cas on ne demandera pas d'entrées clavier mais on interrogera l'IA ; ou il peut être joueur humain.

IV - L'Algorithme de Monte Carlo

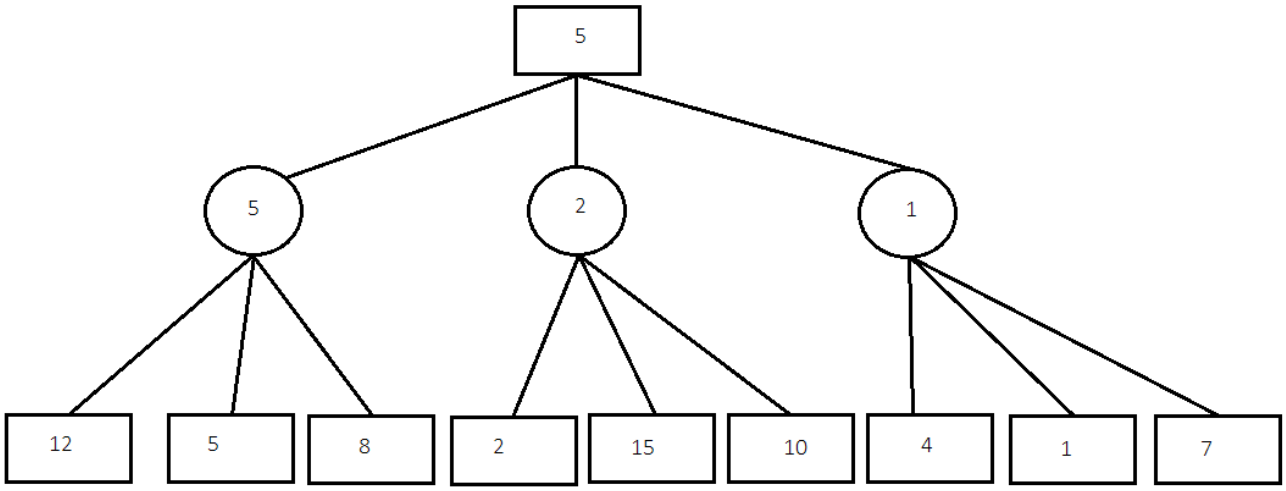
1 - L'Algorithme MinMax

Avant d'aborder l'algorithme de Monte Carlo, il est intéressant d'effectuer un bref rappel à propos de l'algorithme MinMax. En effet, on peut dire que l'algorithme de Monte Carlo en est une version dérivée.

L'algorithme MinMax est un algorithme qui s'applique à la théorie des jeux. Son objectif étant de lister toutes les possibilités de jeu à partir de l'état de jeu courant dans un arbre sous forme de nœuds. Chaque nœud listera tous les coups possibles pour l'adversaire. On progresse dans l'arbre jusqu'à une profondeur donnée.

Les nœuds qui sont à la profondeur souhaitée sont appelés « feuilles ». Elles se verront attribuer une valeur évaluée par une fonction heuristique. Une heuristique simple pour le jeu de Karuba serait de compter le nombre de temples atteints pour un état. L'objectif du MinMax étant de minimiser les pertes maximum. On va ainsi propager les valeurs dans l'arbre tels que :

- Les nœuds pairs se verront attribuer la valeur maximum de leurs fils.
- Les nœuds impairs auront pour valeur le minimum.



Sur ce graphique, les ronds représentent l'adversaire et les rectangles le joueur. On voit bien que l'on cherche à minimiser les pertes en prenant le pire coup que peut jouer le joueur pour le meilleur coup de l'adversaire. Ainsi, cet arbre nous renvoie le « meilleur » coup à jouer pour un état de jeu donné.

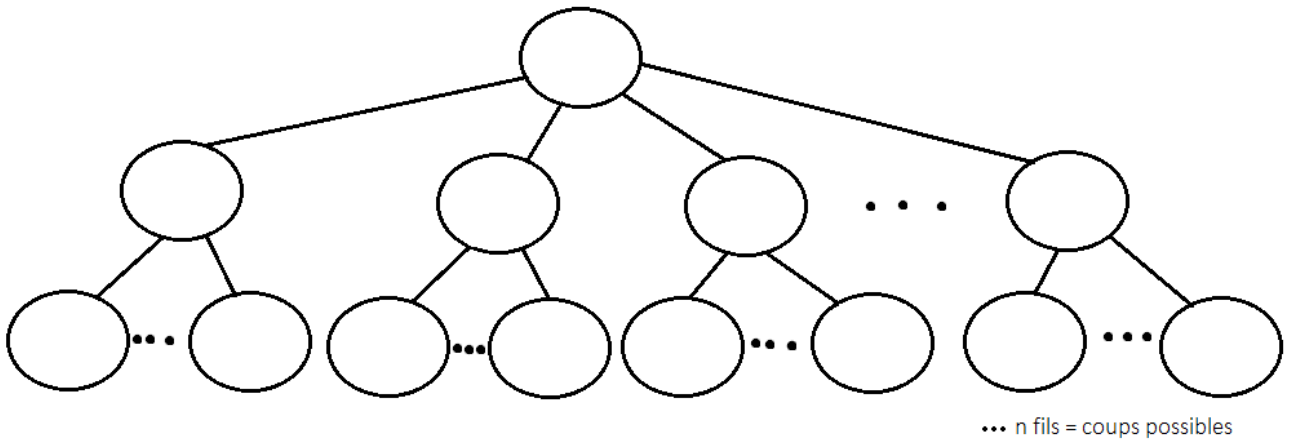
2 - L'Algorithme de Monte Carlo

L'algorithme de Monte Carlo est lui aussi un algorithme s'appliquant à la théorie des jeux. Il veut son nom à la ville de Monte Carlo qui possède un célèbre casino où se pratiquent des jeux de chances. A l'inverse de l'algorithme MinMax, l'algorithme de Monte Carlo ne va pas raisonner en fonction de l'adversaire. Il a un objectif à atteindre et sa stratégie est d'y parvenir le plus vite possible. Pour y arriver, l'algorithme prendra deux arguments en paramètres :

- Une profondeur d'exploration maximum de l'arbre.
- Un budget.

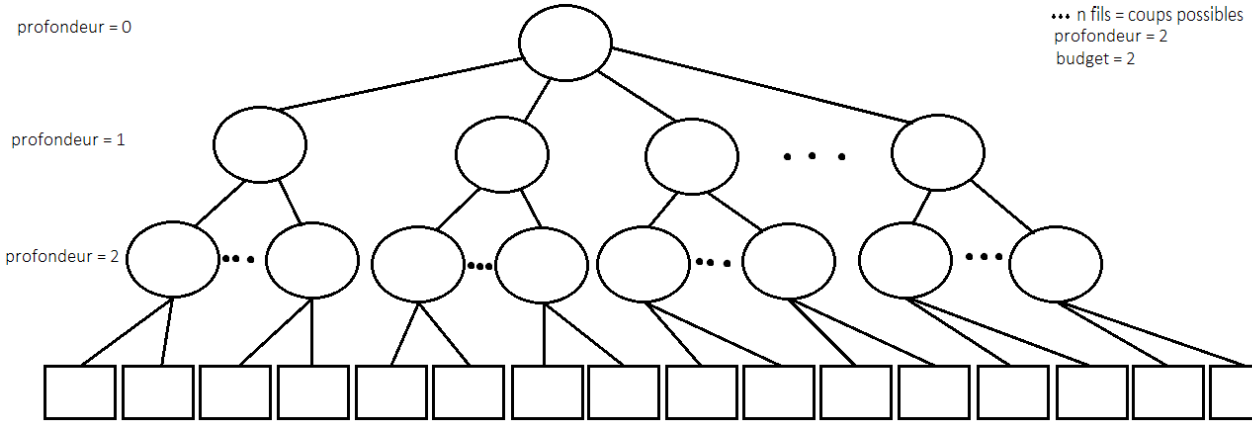
Génération de l'Arbre

Cet algorithme va générer un arbre à partir de l'état courant. Chacun des nœuds de l'arbre représentera un coup à jouer. Ils auront autant de nœuds fils que de possibilité de coups à jouer. On explore ainsi l'arbre jusqu'à la profondeur souhaitée.



Les Nœuds Simulations

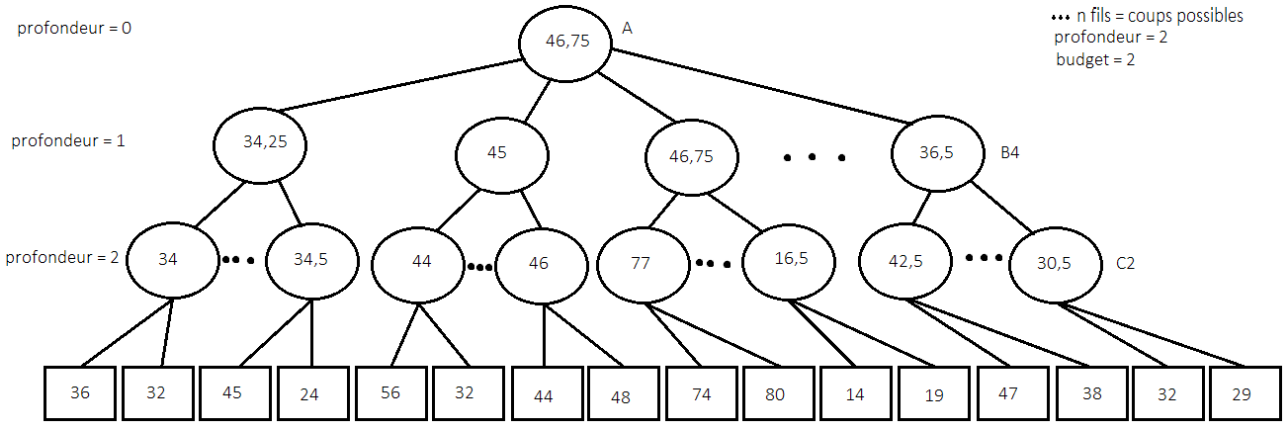
Les feuilles de l'arbre (nœuds à la profondeur souhaitée) auront des fils. Des nœuds que l'on appellera « nœuds simulation ». Pour ces simulations, nous joueront rapidement des coups choisit aléatoirement parmi la liste des coups possibles de façon à arriver le plus vite à un objectif donné. Pour Karuba, l'objectif fixé est d'amener les quatre aventuriers aux quatre temples. Ces nœuds simulations prendront comme valeur le nombre de coups réalisés avant d'arriver à l'objectif.



Propagation des valeurs

Voici une autre étape où se démarque l'algorithme de Monte Carlo par rapport aux algorithmes MinMax. On attribuera la moyenne des valeurs des nœuds simulation à leurs nœuds parents. Puis nous remontrons dans l'arbre tel que :

- Les nœuds pairs, comme pour MinMax, prendront le maximum de leurs fils.
- es nœuds impairs, quant à eux, auront comme valeur l'espérance de leurs fils. Autrement dit, pour chaque nœud on multipliera la probabilité du coup par la valeur du nœud. Puis, on additionnera ce produit de chaque nœuds fils.



Pour calculer les nœuds :

- A (profondeur paire) on prend le maximum des fils
- B4 (profondeur impaire) on calcule l'espérance soit :42,5*0,5 + 30,5 * 0,5
- C2 (père des nœuds simulations) on calcule la moyenne de ses fils :(32+29)/2

Le point fort de Monte Carlo est qu'il introduit une notion de probabilité dans sa prise de décision.

3 - Implémentation de l'Algorithme de Monte Carlo

L'algorithme de Monte Carlo utilise l'arbre binaire comme structure de données, nous avons donc décidé de créer une classe Nœud qui sera chaînée avec ses nœuds fils. Ainsi, nous créons un nœud racine à partir de l'état de jeu courant. Il possédera un nombre de nœuds fils correspondant à tous les coups réalisables à partir de cet état. Ces nœuds fils feront de même et ce jusqu'à la profondeur demandée. Nous créons un nombre de nœuds simulation égale au budget demandée. Ces nœuds joueront des coups sélectionnés aléatoirement jusqu'à arriver à l'état final qui est la fin de partie.

Nous n'avons pas rencontré de réelles difficultés pour cette partie si ce n'est que le code n'est pas optimale en terme de temps d'exécution.

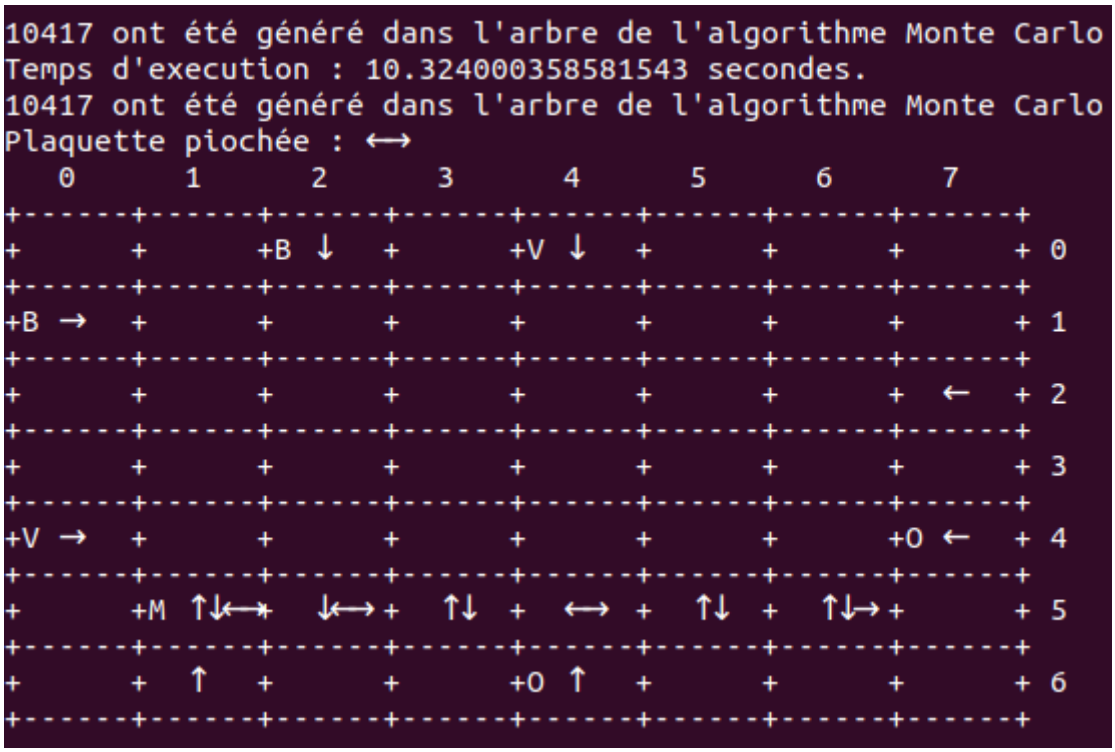
4 – Résultats

Ces données représentent les résultats de Monte Carlo en fonction de la profondeur et du budget :

	Noeuds dans l'arbre	Temps d'exécution d'un choix (en secondes)
Prof 1 / budget 1	46	0.136
Prof 1 / budget 2	61	0.277
Prof 2 / budget 1	684	0.824
Prof 2 / budget 2	902	1.343
Prof 3 / budget 1	9622	5.690
Prof 3 / budget 2	12674	12.7489
Prof 4 / budget 1	130176	70.956
Prof 4 / budget 3	190395	206.391

On peut voir que l'algorithme peut vite être chronophage. Nous avons donc choisi d'étudier les prises de décisions à profondeur 3.

Lorsque les paramètres sont bas, Monte Carlo place quelques plaquettes et choisit de déplacer les aventuriers indéfiniment.
Comme le montre la capture suivante, l'algorithme a placé des plaquettes incohérentes entre elles. Il bloque même les aventuriers.

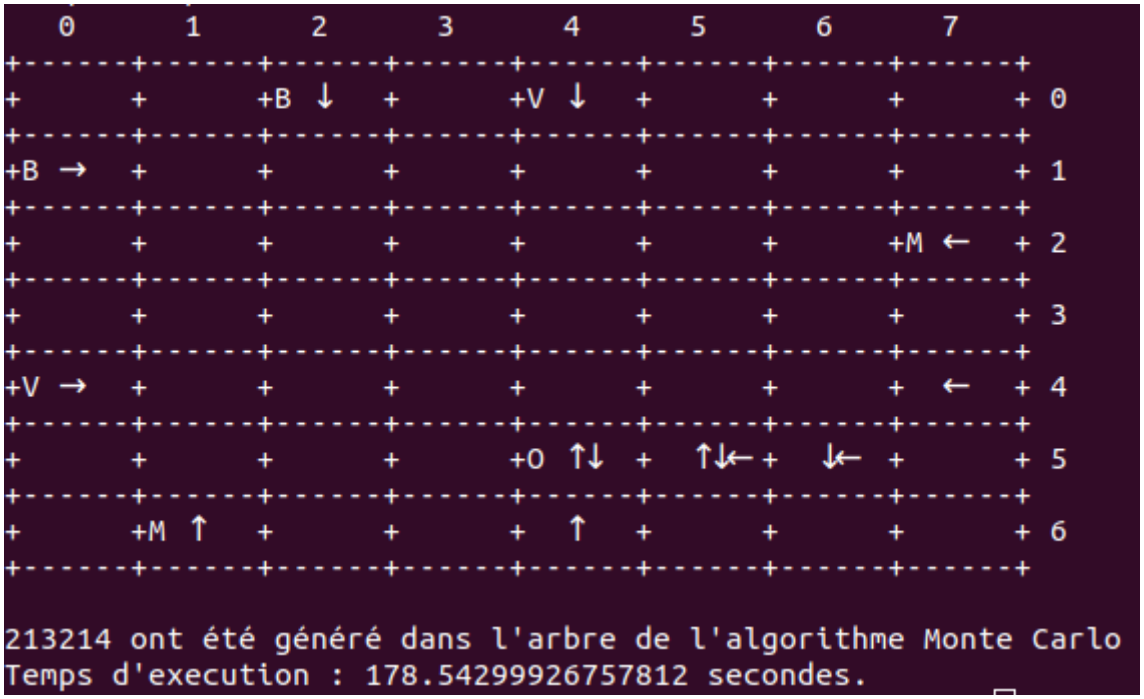


Résultats Monte Carlo (3, 2)

Nous avons testé l'algorithme avec une profondeur de 3 et un budget de 5. L'algorithme décide de poser 4 plaquettes et fait bouger indéfiniment l'aventurier sans poser d'autres plaquettes.

Résultats Monte Carlo (3, 5)

La configuration avec une profondeur de 4 et un budget de 3 est très longue à l'exécution : 3 minutes et 30 secondes pour une prise de décision. Malgré le parcourt d'un plus grand nombre de nœuds, on peut voir sur les résultats suivants que l'algorithme n'est encore pas cohérent dans ses choix.



Résultats Monte Carlo (3, 5)

L'algorithme pose quelques plaquettes et déplace indéfiniment l'aventurier sur ce court chemin.

Nous pensons que ces résultats sont dûs à une mauvaise heuristique des nœuds simulations. En effet, actuellement l'heuristique est de compter le nombre de coups avant la fin de partie qui se caractérise par l'arrivée aux temples des aventuriers mais aussi au fait par le fait de tirer la dernière plaquette dans la pioche. Nous pensons que bien souvent les nœuds simulations créent des chemins incompatibles et s'arrête lorsque la pioche est vide. Cette heuristique n'est donc pas vraiment pertinente. Nous aurions pu imaginer une solution calculant la distance euclidienne entre les aventuriers et leurs temples respectifs. Il doit être également possible de valoriser la pose d'une

plaquette à côté d'autres qui sont compatibles afin de créer un chemin.

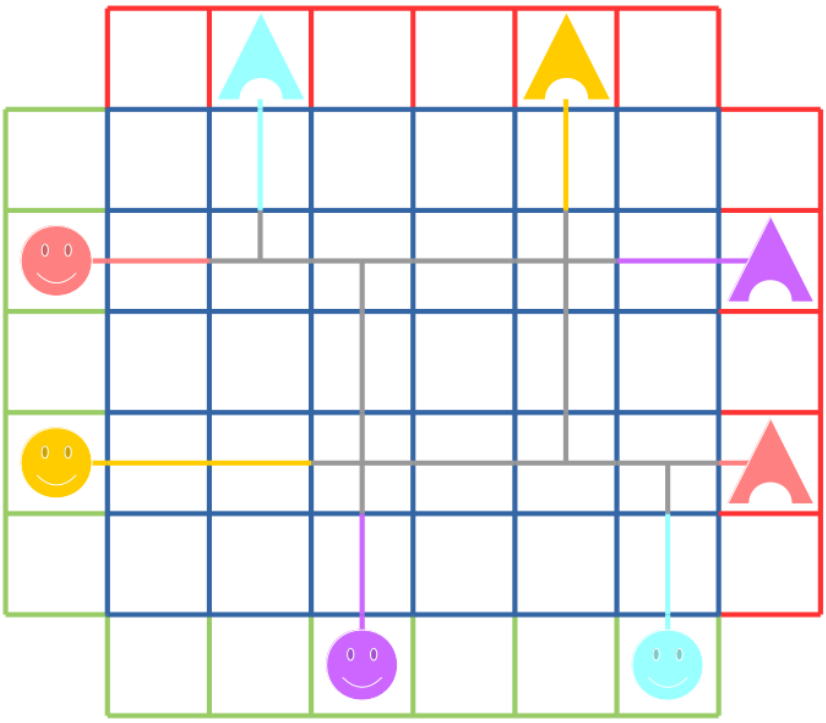
Il aurait été intéressant d'adapter la profondeur et le budget en fonction de l'avancement de la partie. On peut imaginer qu'en début de partie l'algorithme effectue ses recherches avec une profondeur et un budget faible (exemple profondeur 3, budget 1). Ensuite, lorsque le jeu propose moins de possibilités de coups à jouer, augmenter la profondeur à 4 et le budget à 3. Cette solution permettrait d'optimiser les performances de l'algorithme.
Pour tester l'algorithme il aurait été fructueux de pouvoir moduler la taille du plateau afin de tester les performance de Monte Carlo sur une surface plus petite.

V – La Planification Géométrique

1 - Fonctionnement de la Fonction Géométrique

Le jeu de Karuba est un jeu de stratégie ayant pour but principal de faire avancer tous ses propres aventuriers dans leurs temples respectifs avant nos adversaires. Pour cela chaque joueur doit à tour de rôle se frayer un chemin pour chacun de ses aventuriers vers son temple qui lui est attribué. La manière la plus simple pour finir rapidement la partie et ainsi arriver à ce résultat est d'utiliser le moins de tuile possible lors de la création des chemins. De ce fait, nous avons eu l'idée de créer un algorithme permettant de générer un modèle graphique qui répond à ce besoin. À partir de quatre points de départ, l'algorithme nous génère les chemins à créer et à emprunter pour les relier aux quatre points d'arrivée, bien sûr en utilisant le moins de tuile possible. Le but est de représenter des chemins comme dans cet exemple :

- Les aventuriers sont représentés par des Têtes
- Les temples sont représentés par des Triangles



2 - Implémentation Générale de l'Algorithme

Le code pour cet outil a été codé en python pour la simplicité du codage et de l'abstraction du typage que nous offre ce langage. L'algorithme étant non fini, nous avons pas réussi à atteindre notre objectif final. Cependant l'exécution de cet algorithme reste intéressante et permet de retourner une visualisation graphique avec des chemins de crées. Chacun des chemins sont donc crées indépendamment des uns des autres. Ainsi lors du traçage de chacun de ces chemins, nous ne prenons pas en compte de l'emplacement des autres déjà tracés. Cela permet de trouver le(s) point(s) de convergence(s) entre tous ces chemins. Et ainsi pour chacun des joueurs du jeu de Karuba de savoir ou placer la tuile représentant un carrefour avec les quatre directions dans la grille de manière à savoir où la jouer quand ces joueurs la tireront dans la pioche.

L'algorithme se présente sous la forme de trois classes de la manière suivante :

- D'une classe Direction qui représente une des quatre directions d'une tuile du jeu de Karuba. C'est à dire les directions HAUT, DROITE, BAS, et GAUCHE.
- D'une classe Case qui représente une tuile du jeu de Karuba.
- D'une classe Grille qui représente la grille de jeu de Karuba.

2-1 - La Classe Direction

La classe Direction étant la classe principalement utilisé pour nos tests de directions lors de la création de nos chemins, il fallait utiliser une architecture simple et logique pour la représenter. Nous avons choisis une énumération avec la bibliothèque *enum* de Python. Il en existe donc quatre, une pour chacune des directions existante. De plus une méthode pour trouver la direction opposée a été mise en place, elle sera très utile par la suite.

```
class Direction(Enum): ### Objet représentant une direction d'une tuile du jeu de Karuba
        HAUT   = ( 0,-1)
        DROITE = ( 1, 0)
        BAS    = ( 0, 1)
        GAUCHE = (-1, 0)

        def opposee(self): ### Retourne l'opposée de la direction courante
                if self == Direction.HAUT:
                        return Direction.BAS
                if self == Direction.DROITE:
                        return Direction.GAUCHE
                if self == Direction.BAS:
                        return Direction.HAUT
                if self == Direction.GAUCHE:
                        return Direction.DROITE
```

2-2 - La Classe Case

La classe Case est le deuxième objet qu'on va la plus répéter derrière la classe Direction. À chaque fois que l'on va placer une tuile dans la grille du jeu de Karuba, un nouvel objet Case sera crée. Cet objet possède dès sa création d'une coordonnée (par rapport où elle sera placé dans la grille) et d'une liste de directions (la liste des directions possibles quand un aventurier passe par cette tuile). De plus elle possède des méthodes de classe pour lire chacune de ces informations.

```
class Case(object): ### Objet représentant une tuile du jeu de Karuba
        def __init__(self, x, y, directions):
                self.__x = x
                self.__y = y
                self.__directions = directions

        def lireX(self): ### Retourne l'abscisse de la tuile courante
                return self.__x

        def lireY(self): ### Retourne l'ordonnée de la tuile courante
                return self.__y

        def lireDirections(self): ### Retourne les directions de la tuile courante
                return self.__directions
```

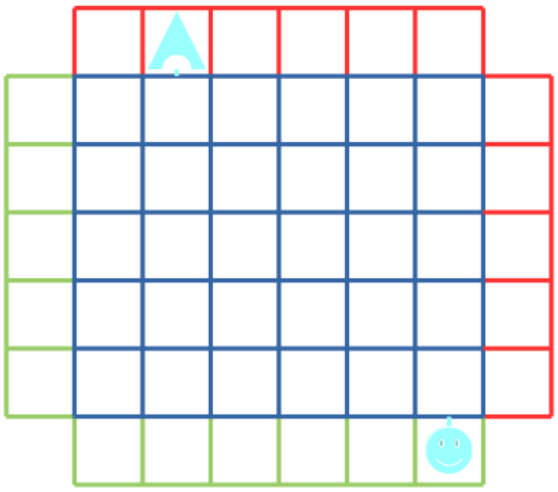
2-3 - La Classe Grille

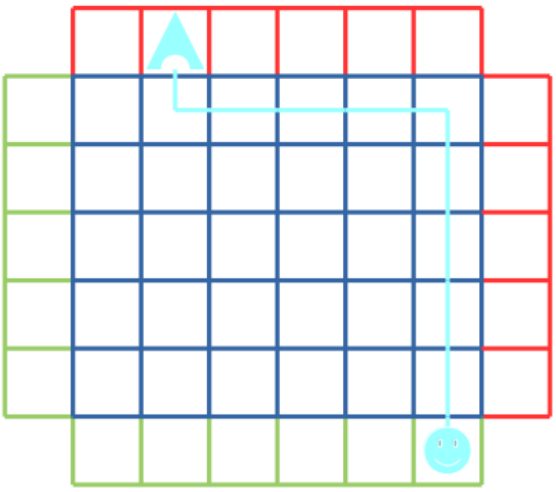
La classe Grille est la classe la plus importante de notre programme, étant le moteur de l'algorithme, elle s'occupe de toutes les interactions pour obtenir le résultat tant attendu à la fin du programme. Elle représente la grille de simulation des chemins où l'on placera toutes nos tuiles pour créer les chemins « parfaits » pour gagner le plus rapidement au jeu de Karuba.

Initialement l'objet une fois crée, crée simplement une grille vide d'une taille donnée en entrée. Elle possède une méthode pour vérifier si une case de la grille est vide à partir d'une coordonnée donnée. Une autre pour créer et placer une tuile de jeu de Karuba dans la grille à une coordonnée donnée, dans le cas où la case n'était pas vide les listes de directions de l'ancienne et de la nouvelle sont fusionnées. C'est ainsi que l'on peut obtenir des tuiles de carrefour avec quatre directions. Et encore une qui permet de retourner la liste des directions d'une tuile se trouvant à une coordonnée donnée dans la grille. Ceci étant dit, la création des meilleurs des chemins est dirigée par la méthode de classe creerChemin(). Cette méthode prend en entrée un point de départ et un point d'arrivée et place dans la grille les tuiles nécessaires pour lier ces deux points. Il existe ainsi quatre cas de chemins.

2-3-1 - Cas 1 : le Départ en Bas et l'Arrivée en Haut

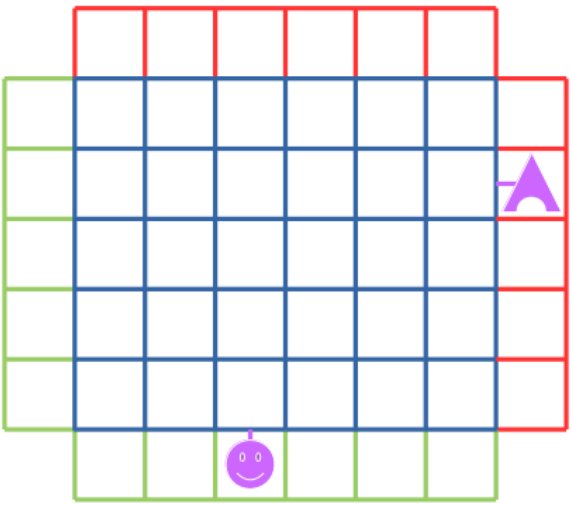
Dans ce cas, le point de départ affiche une direction HAUT (car la tuile est en bas de la grille) et le point d'arrivée une direction BAS (car la tuile est en haut de la grille). Le principe est simple, ici il faudra donc placer une multitude de tuiles de ligne droite de HAUT en BAS de la case de départ qui se trouve en bas vers le haut. Une fois que l'avant dernière ligne de la grille est atteinte, on change de direction si les abscisses de ces deux points ne sont pas les même en plaçant des tuiles de ligne droite de GAUCHE vers la DROITE de sorte à se rapprocher du point d'arrivée. Puis on rechange de direction pour atteindre le point d'arrivée. Si initialement les deux points avaient la même abscisse, seule une successions de tuiles de HAUT en BAS est placée dans la grille, reliant plus simplement ces deux points.

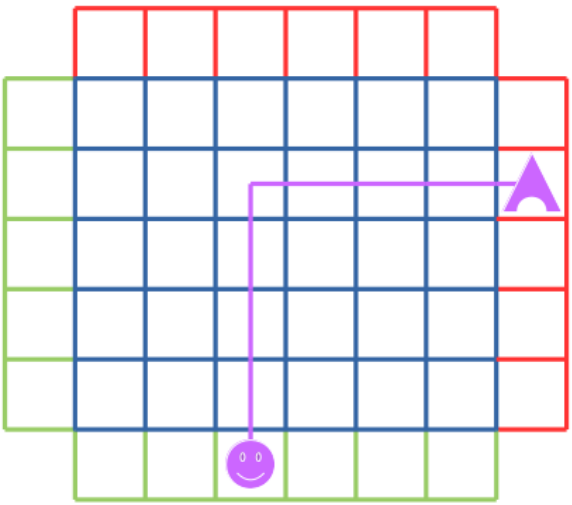




2-3-2 - Cas 2 : le Départ en Bas et l'Arrivée à Droite

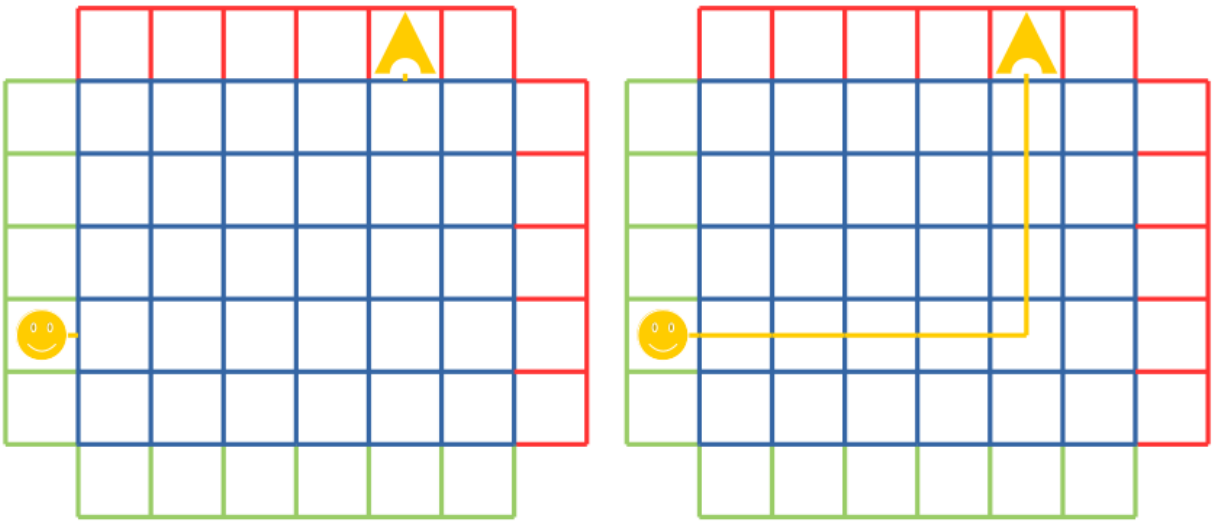
Dans ce cas, le point de départ affiche une direction HAUT (car la tuile est en bas de la grille) et le point d'arrivée une direction GAUCHE (car la tuile est à droite de la grille). Le principe est simple, ici il faudra donc placer une multitude de tuiles de ligne droite de HAUT en BAS de la case de départ qui se trouve en bas vers le haut. Une fois que la ligne du point d'arrivée est atteint, on change de direction et on se dirige vers la DROITE. Et puis on place à nouveau une multitude de ligne droite mais de GAUCHE vers la DROITE jusqu'à que la liaison des deux points soient faite.





2-3-3 - Cas 3 : le Départ à Gauche et l'Arrivée en Haut

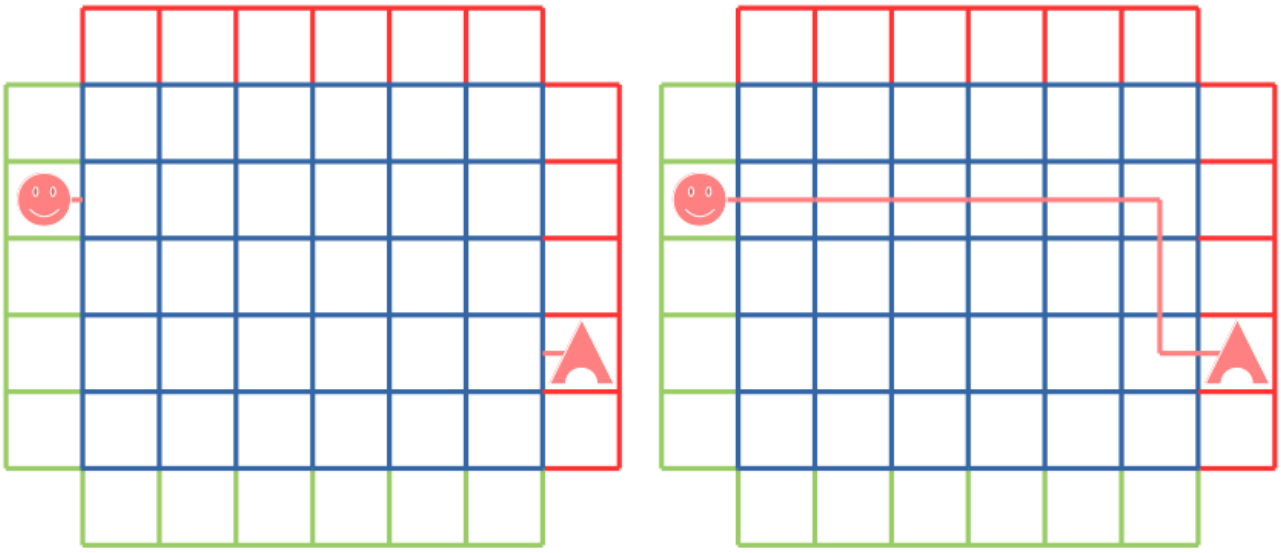
Dans ce cas, le point de départ affiche une direction DROITE (car la tuile est à gauche de la grille) et le point d'arrivée une direction BAS (car la tuile est en haut de la grille). Le principe est simple, ici il faudra donc placer une multitude de tuiles de ligne droite de GAUCHE vers la DROITE de la case de départ qui se trouve à gauche vers la droite. Une fois que la colonne du point d'arrivée est atteint, on change de direction et on se dirige vers le HAUT. Et puis on place à nouveau une multitude de ligne droite mais de HAUT en BAS jusqu'à que la liaison des deux points soient faite.



2-3-4 - Cas 4 : le Départ à Gauche et l'Arrivée à Droite

Dans ce cas, le point de départ affiche une direction DROITE (car la tuile est à gauche de la grille) et le point d'arrivée une direction GAUCHE (car la tuile est à droite de la grille). Le principe est simple, ici il faudra donc placer une multitude de tuiles de ligne droite de GAUCHE vers la DROITE de la case de départ qui se trouve à gauche vers la droite. Une fois que l'avant dernière colonne de la grille est atteinte, on change de direction si les ordonnées de ces deux points ne sont pas les même en plaçant des tuiles de ligne droite de HAUT en BAS de sorte à se rapprocher du point d'arrivée. Puis on rechange de direction pour atteindre le point d'arrivée. Si initialement les deux

points avaient la même ordonnée, seule une successions de tuiles de GAUCHE vers la DROITE est placée dans la grille, reliant plus simplement ces deux points.

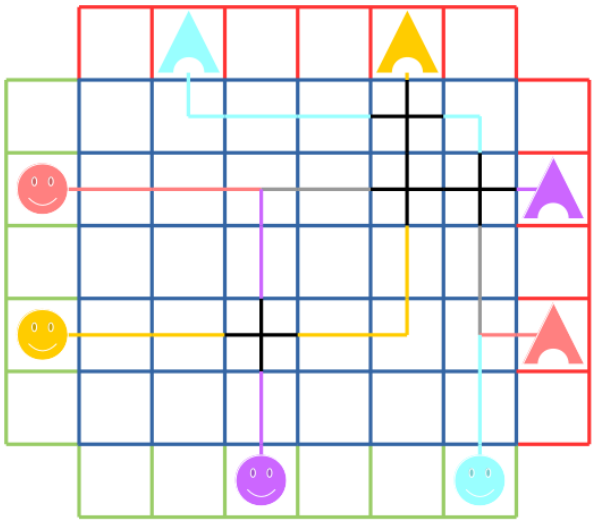


Dans chacun de ces cas, un test est effectué avant chaque placement de nouvelle tuile. Si une tuile est déjà placé alors la nouvelle est fusionnée avec l'ancienne, permettant d'obtenir des tuiles avec un nombre de directions supérieures à deux et donc d'obtenir des tuiles sous forme de carrefour à quatre direction ou en T à trois directions.

3 - Test

En lançant le programme avec les coordonnées suivantes nous obtenons cette génération de chemins suivante dans le terminal :

- Aventurier n°1 → (0, 4)
- Temple n°1 → (5, 0)
- Aventurier n°2 → (3, 6)
- Temple n°2 → (7, 2)
- Aventurier n°3 → (0, 2)
- Temple n°3 → (7, 4)
- Aventurier n°4 → (6, 6)
- Temple n°4 → (2, 0)



```
---- ---- --B- ---- ---- --B- ---- ----
---- ---- HD-- -D-G -D-G HDBG --BG ----
-D-- -D-G -D-G -DBG -D-G HDBG HDBG ---G
---- ---- ---- H-B- ---- H-B- H-B- ----
-D-- -D-G -D-G HDBG -D-G H--G HDB- ---G
---- ---- ---- H-B- ---- ---- H-B- ----
---- ---- ---- H--- ---- ---- H--- ----

Tuile à 4 directions sur la case (5, 1)
Tuile à 4 directions sur la case (5, 2)
Tuile à 4 directions sur la case (6, 2)
Tuile à 4 directions sur la case (3, 4)
```

Le programme retourne que des tuiles à quatre directions sont nécessaires sur les cases en (5, 1), en (5, 2), en (6, 2), et en (3, 4). Cela se voit aussi visuellement sur notre schéma au dessus avec les tuiles à quatre directions coloriées en noir.

VI - Conclusion

Le domaine de la théorie des jeux est intéressant. Il permet de se focaliser sur les éléments clés du jeux et d'essayer de modéliser une stratégie qui pourrait le résoudre. Ce projet nous a permis d'évaluer nos compétences techniques. En effet, la réalisation du jeu étant secondaire, nous devions impérativement réussir à obtenir un jeu fonctionnel. Il a été très formateur de travailler sur deux langages différents (Python, Java).

Bien qu'à l'étude l'algorithme de Monte Carlo semblait pouvoir mieux fonctionner pour le jeu de Karuba, il a été décevant de voir ses maigres performances. Cependant, il a été intéressant de voir une alternative aux algorithmes MinMax qui intègre une part de probabilité dans la prise de décision.

L'algorithme de fonction géométrique est parfaitement adapté à Karuba mais pas seulement. On peut très bien le voir évoluer dans d'autres types d'application et pas uniquement les jeux de plateau. Il nous montre une alternative à des algorithmes comme A* afin de générer des chemins.

Il aurait été pertinent de comparer nos algorithmes à d'autres algorithmes de planification comme l'algorithme d'UTC. L'autre point à améliorer est de revoir notre fonction heuristique quant à l'évaluation des nœuds de simulations.