RAPPORT DE PROJET

Rasende Roboter

Réalisé par

Victor MIRA

Baptiste PERRAT DIT JANTON

UV : IA41

Année universitaire 2020 - 2021

Sommaire

1. Introduction

1.1 Principe du jeu

1.1.1 Déroulement

Le jeu de base est un jeu de plateau dans lequel chaque joueur joue en même temps pour trouver le chemin le plus court. Sur le plateau de jeu 4 robots de 4 couleurs différentes sont positionnés aléatoirement. Par la suite un joueur pioche une carte mission sur laquelle est représenté un symbole ex: Triangle Bleu. A partir de ce moment, les joueurs réfléchissent sur le chemin le plus court que le robot de la même couleur que la mission doit prendre. En sachant que les robots peuvent bouger uniquement en ligne doitre et ne peuvent s'arrêter que contre un mur.
Le joueur ayant le plus de missions gagne.

1.1.2 Interprétations

Nous avons interprété le problème avec la fonctionnalité de jouer contre un ordinateur de cette façon:
Nous avons choisis de réaliser ceci avec une programmation orienté objet dont le diagramme de classe est ci-dessous (Première version certaines choses ont été rajouté depuis) :

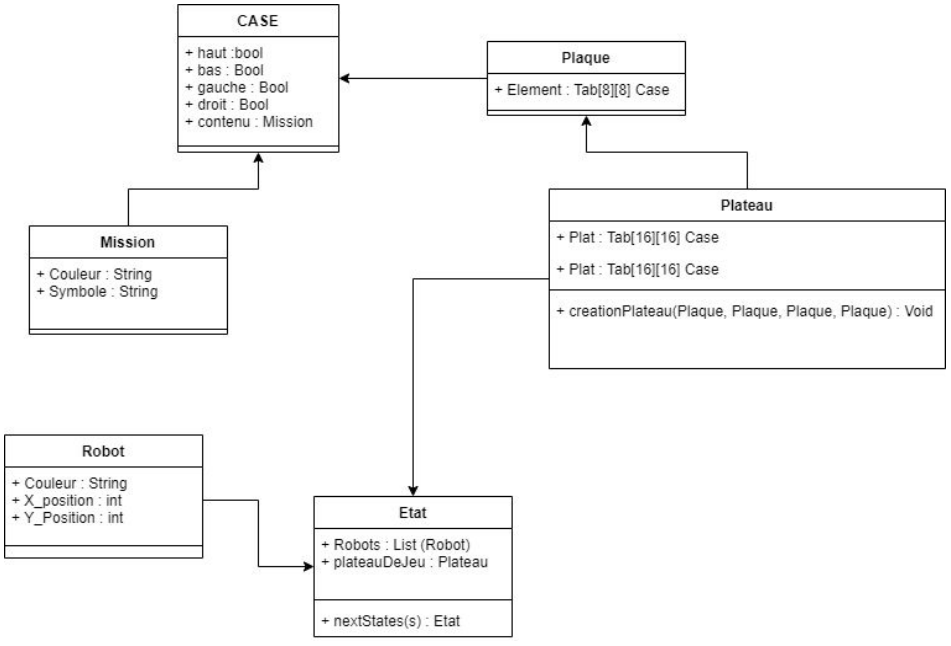

Figure 1 - Diagramme de Classe du Projet

Avec comme différentes classes:

- Robot
- Etat
- Mission
- Case
- Plaque
- Plateau

Une mission peut être affectée à une *Case*. Une *Plaque* est un tableau de 8 par 8 *Case* qui correspond au 4 plaques qui constituent le Plateau.

Un Etat est l'intelligence artificielle contre laquelle le ou les joueurs jouent, pour fonctionner il prend une liste de 4 robots et un plateau de jeu.

2. Implémentation

2.1 Classe Mission

2.1.1 Attributs

La Classe *Mission* est assez simple, elle ne possède que 2 attributs l'un définissant son symbole l'autre sa couleur.

2.1.2 Objets

La classe créé en tout 9 objets, chacun correspondant au 9 missions possibles, entre les 4 couleurs: Rouge, Bleu, Vert, Jaune et les 4 symboles: Losange, Cercle, Carré, Triangle un seul objet possède une petite spécificitée c'est la mission Multi-couleur, qui ne possèdent aucun symbole et en couleur Multicouleur

2.2 Classe Case

2.2.1 Attributs

La classe *Case* est définie par 5 attributs, tous les 4 différents murs pouvent être ou ne pas être présent (Haut, Bas, Droit, Gauche). Ainsi que l'attribut contenu qui prend soit une missions soit il reste nul.

2.2.2 Méthodes

Il y a deux méthodes dans cette classe, une *toDroite(self)* et *toGauche(self)*. Elles permettent de changer l'orientation des murs de la case, cela nous sera très utile pour la suite.

2.2.3 Objets

Il y a plusieurs types d'objets pour la classe *Case* tout d'abord il y a ceux qui serviront à toutes les *Plaques* ensuite ceux qui n'en serviront qu'une seule puis les *Cases* possèdants des missions.

Tout d'abords pour chaque plaques il était utile de créer chaque type de *Case* qui sont:

- Case Mur Droit
- Case Mur Gauche
- Case Mur Bas
- Case Mur Haut
- Case Mur Haut Bas
- Case Mur Haut Droit

Pour permettre le bon fonctionnement de certaines fonctionnalités car nous avons opté pour ne pas créer un objet pour chaque case mais d'en créer parfois des identiques.

Par la suite, j'ai créé une liste de toutes les cases "spéciales" que possèdent chaque Plaque. De plus, une liste d'autres objets cette fois neutre peuvent être utilisés dans des situations diverses.

2.3 Classe Plaque

2.3.1 Attribut

Une *Plaque* ne possède qu'un seul attribut, qui est element, un tableau de 8 par 8 Cases ,pour les tableaux nous utilisons la bibliothèque Numpy.

2.3.2 Méthode

La classe *Plaque* n'a qu'une seule méthode qui est *Switch90(self,t)*. Cette méthode permet comme son nom l'indique de tourner toute une plaque de 90°. Cela est nécessaire car dans l'élaboration du *Plateau* les *Plaques* doivent pouvoir être possitionnées dans n'importe quel sens. L'attribut "t" permet simplement d'indiquer de quelle *Plaque* s'agit-il, puis de tourner la liste des *Cases* "spéciales" de celle-ci grâce à la méthode *toDroite(self)* des *Cases*.Voici un exemple d'application de la fonction *Switch90* avec la Plaque1 :

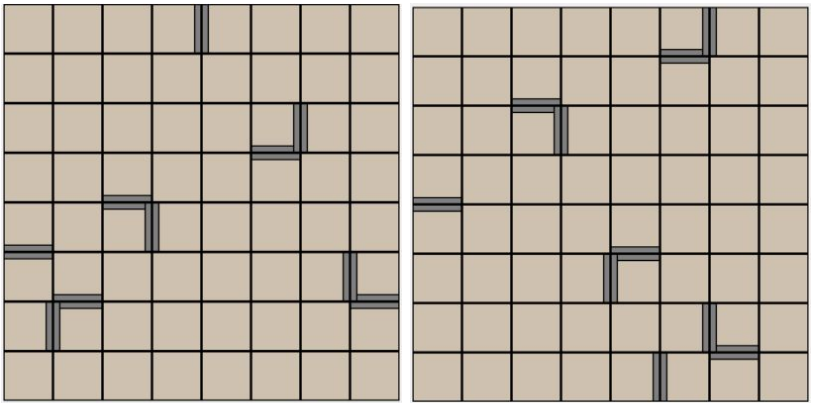

Figure 2 - A Gauche la plaque 1 avant et à Droite après Switch90

2.3.3 Objets

Les objets issus de la *Plaque* sont les 4 différentes *Plaque* du jeu, avec leurs missions respectives.

2.4 Classe Plateau

2.4.1 Attribut

Tout comme la classe *Plaque*, *Plateau* possède un tableau de 16 par 16 *Cases*. Mais en plus il a d'autres attributs utiles, comme une liste allant de 0 jusqu'à 3, une liste de Plaque ainsi qu'un compteur de nombre de *switch90* affecter à la *Plaque* n°3.

2.4.2 Méthodes

Deux méthodes sont à relever dans la classe *Plateau*. Là où précédamment nous ne les avons point évoqués, ici le constructeur de la classe est assez complexe. Mais d'abord la méthode *élaborationPlateau(self)* qui permet de dire dans quel ordre les *Plaques* seront dans le *Plateau*. Pour cela j'utilise la bibliothèque *Random* avec la fonctionnalitée "choice" qui choisit aléatoirement un élément d'une liste. A la fin de la méthode la liste [0,1,2,3] sera dans un ordre aléatoire.

Ensuite le constructeur, qui prend comme attributs 4 *Plaques*. Avec la bibliothèque Randon.randint, j'utilise *Switch90* entre 0 et 3 fois ce qui nous permet d'obtenir des *Plaques* dans un sens aléatoire. On appelle la fonction *elaborationPlateau* puis on remplit *Plateau*.

Il nous reste à gérer le centre du *Plateau* qui doit être inaccessible, pour cela on change simplement les *Cases* qui nous intéressent dans celui-ci.

Pour finir, il existe une *Case* qui reste problématique, car de la façon dont nous avons programmé, la *Case* adjacente à celle Multi-couleur doit posseder un mur, le problème c'est que le mur doit être défini des deux côtés. Or en changeant de place aléatoirement les plaques cela pose un problème c'est pour cela, qu'en fonction du nombre d'appels à la fonction *Switch90* pour la *plaque* n°3, nous devons, si nécessaire, ajouter un mur à la *case* adjacente à Multi-couleur.

Pour finir, il faut simplement définir les *Cases* qui autour du centre et voilà le *Plateau* est créé.

2.4.3 Objet

Un seul *Plateau* est créé pour le besoin du jeu.

2.5 Classe Robot

2.5.1 Attributs

La classe *Robot* possède 4 attributs, une position en x, une position en y et un plateau qui permet l'utilisation de certaines méthodes

2.5.2 Méthodes

En plus de son constructeur, la classe robot possède 4 méthodes, gauche, droite, haut et bas qui permettent le déplacement du robot sur son plateau. Pour chaque méthode, on vérifie : si la case sur laquelle est le robot possède un mur dans la direction où l'on souhaite se déplacer, si la case où l'on souhaite aller fait partie du plateau, et si un autre robot est déjà présent sur la case. Les méthodes prennent en argument un état car autrement on ne pourrait pas vérifier la présence des robots sur les différentes cases.

2.5.3 Objets

On définit 4 *robots* initiaux avec des positions différentes choisies aléatoirement parmi toutes les cases du plateau excepté les cases centrales. Les autres robots seront instancié à partir des premier dans les états suivants

2.6 Classe DrawCase

2.6.1 Attributs

La classe est héritée d'un canvas et possède donc ses attributs. Elle possède également des attributs de classe : la couleur des murs et des cases sous forme de chaîne de caractère représentant le code hexadécimal de la couleur, la taille des murs et des cases (afin de simplifier les modifications d'affichage). La classe possède également comme attribut une case

2.6.2 Méthodes

Le constructeur permet de créer sur le canvas tout ce qui est commun à chaque case, un rectangle pour le fond de la case et des murs en fonction des murs ; il appelle ensuite la méthode draw puis si la case est une case du milieu, la méthode *drawMission*. La méthode drax ne prend pas d'argument, elle est chargée, si une mission est présente, de dessiner sur la case le symbole correspondant en respectant la bonne couleur. La méthode *drawMisson* permet de dessiner sur le canvas une mission passée en argument. La dernière méthode de cette classe est *drawRobot*, elle prend en argument un robot et dessine sur la case le robot de la couleur correspondante.

2.6.3 Objets

Pour chaque case, un objet *drawCase* sera créé et stocké dans un tableau à deux dimensions *drawnCase*, chaque case sera alors ajouter à la grille en suivant sa position dans le plateau. Ensuite les robots seront dessinés sur leur case initiale puis le tout sera pack sur une fenêtre tkinter.

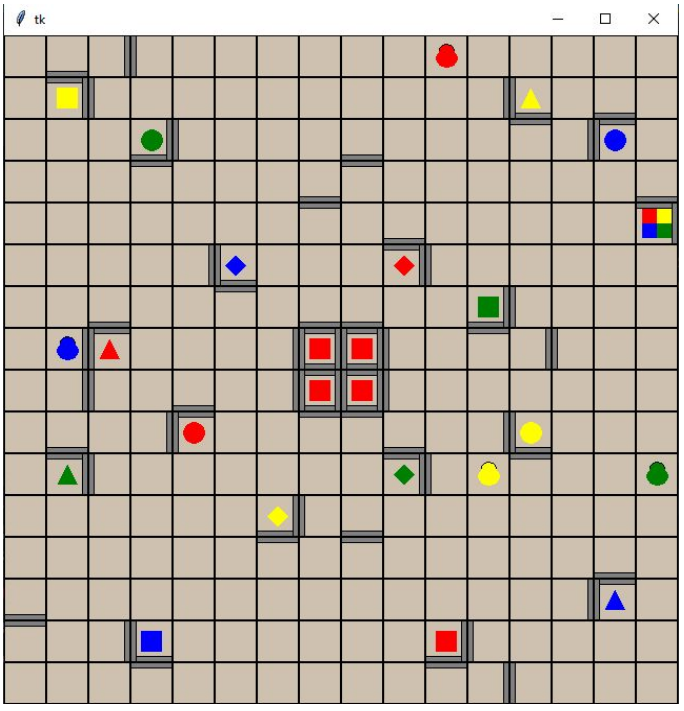

Figure 3 - Un exemple d'affichage d'un plateau

2.7 Classe Etat

2.2.1 Attributs

La classe *Etat* a comme attribut un plateau, 4 robots (un de chaque couleur), une mission une distance euclidienne, une distance de manhattan ainsi qu'un coût.
Chacun de ces attributs est, ou peut être, essentiel à la recherche de solution selon la configuration de celle-ci.

2.2.2 Méthodes

Le constructeur initialise tous les attributs avec les arguments qui lui sont envoyés, sauf les distances de manhattan et d'euclide qui sont calculées directement par les méthodes correspondantes.

La méthode *isRobotHere* qui prends en arguments deux entier représentants une position dans plateau retourne un booléen valant vrai si un des robots de l'état se trouve sur la position indiqué du plateau

La méthode *nextState* renvoie une liste de toutes les configurations possibles des prochains états, elle va donc pour chaque robot, s'il peut se déplacer dans une direction, le faire se déplacer puis ajouter cet état à la liste.

La méthode *isFinalState* permet de vérifier si l'état est un état final elle renvoie True lorsque le robot correspondant à la couleur de la mission (n'importe lequel pour la case multicolore) a les mêmes coordonnées que celle ci

La méthode *euclideanDistance*, renvoie la distance euclidienne entre la mission et le robot de couleur correspondante, et dans le cas de la mission multicolore, la plus petite distance euclidienne entre les robots et la case de la mission

La méthode *manhattanDistance* fait la même chose mais en calculant la distance de Manhattan

La méthode *toString* permet de renvoyer les information concernant, le coût de l'état, son heuristique, et la position de ses robots

2.2.3 Objets

Un état initial contenant le plateau, les robots, la mission ainsi qu'un coût de 0 est créé, il sera envoyé en paramètres à la fonction choisie pour la recherche.

2.8 Stratégies de Recherche

Une classe node contenant un état et son état père est utilisé pour créer des graphes.

La fonction *getPathFrom* permet de récupérer la liste des etats en partant d'un état initial

la fonction *getStringPath* qui affiche une liste d'état en utilisant la méthode toString d'un état

la fonction *Astar* implémentant l'algorithme *A** adapté à notre problème, l'algorithme possède un argument "depth" permettant de choisir la profondeur des états à chercher avant de procéder au tri.

la fonction *BFS* implémentant l'algorithme *BFS* adapté à notre problème.

3. ANALYSE

3.1 Jeux d'essais

Nous avons tout d'abord commencé les recherches avec les algorithmes DFS et BFS, le DFS ne donne aucun résultat, cela n'est pas étonnant vu la profondeur du graphe créé. Le BFS quant à lui ne trouve que très rarement la solution qui est alors une solution à un problème simple, avec un maximum de 3 coups de résolution.

Passons maintenant à A*, avec une "depth" de 1 et une distance de Manhattan comme heuristique : sur 20 essais, une solution est trouvée 6 fois, les autres essais durant plus d'une minutes ils ont été arrêtés. parmi les réussite on note : une difficulté de problème nécessitant 7 coups de résolution sans aide d'autre robots (d'une autre couleur que celle de la mission), des solution de 2 et 1 coup sans aide d'autre robots, des solution de 4, 5 et 6 coups impliquant d'autre robots que le robot de la couleur concernée.

En changeant la "depth" à 2, les résultats semblent biens moins intéressants, en effet seulement 3 solutions sont trouvées, dont deux nécessitant seulement 2 coups pour être résolues sans autres robots et une en 6 coups avec d'autres robots alors que le problème était résolvable en 4.

Changement de l'heuristique : distance euclidienne. Sur 20 essais la solution est trouvée 8 fois avec un difficulté maximale de 9 coups, d'autres robots sont utilisés 5 fois pour résoudre des problèmes d'une difficulté de 94, 5, 5, 6 et 9 coups alors que sans autre robot la difficulté est à 3, 4 et 5.

3.1 Exploitation

L'heuristique de la distance euclidienne semble marcher mieux que la distance de Manhattan, mais le nombre de jeux d'essais ne permet pas de l'affirmer. Un phénomène intéressant à lieux lorsqu'on analyse la manière dont sont trouver les solution : avec la distance de Manhattan, il m'est arrivé plusieur fois ce problème :

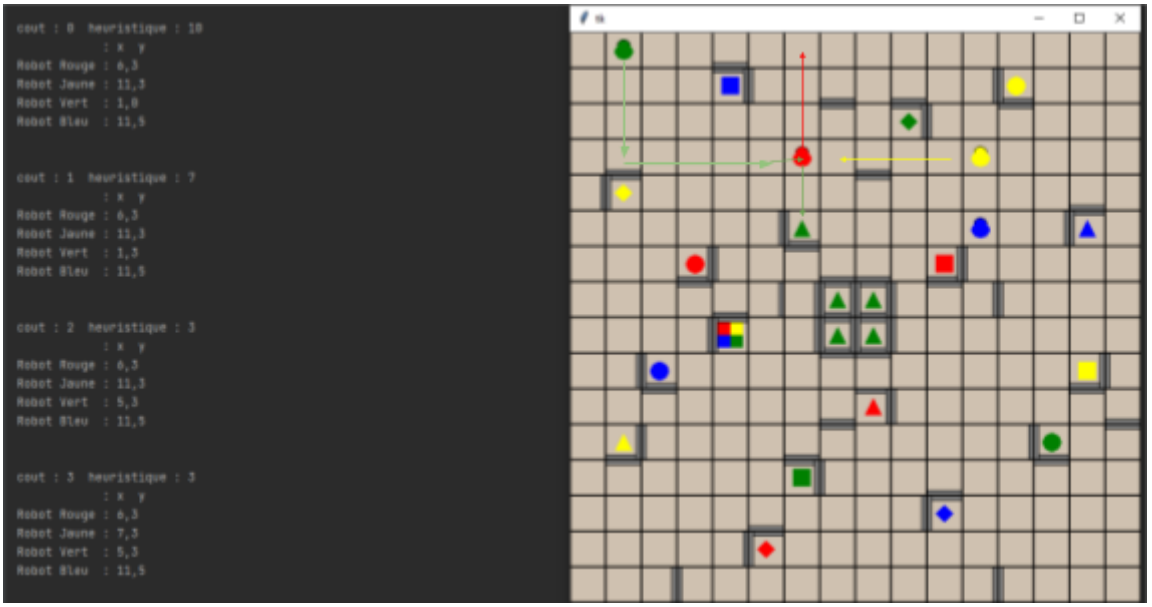

Figure 4 - Une résolution possible avec Manhattan

Les robots jaune et rouge aurait pu bouger avcant le vert ce qui aurait permis de faire gagner 1 coups, dans la même situation, avec une heuristique de distance euclidienne, les autres robots bougent en premier, ce qui correle avec les meilleurs résultats de la distance euclidienne.

La deuxième chose qui m'a interpellé est un encore problème de priorisation

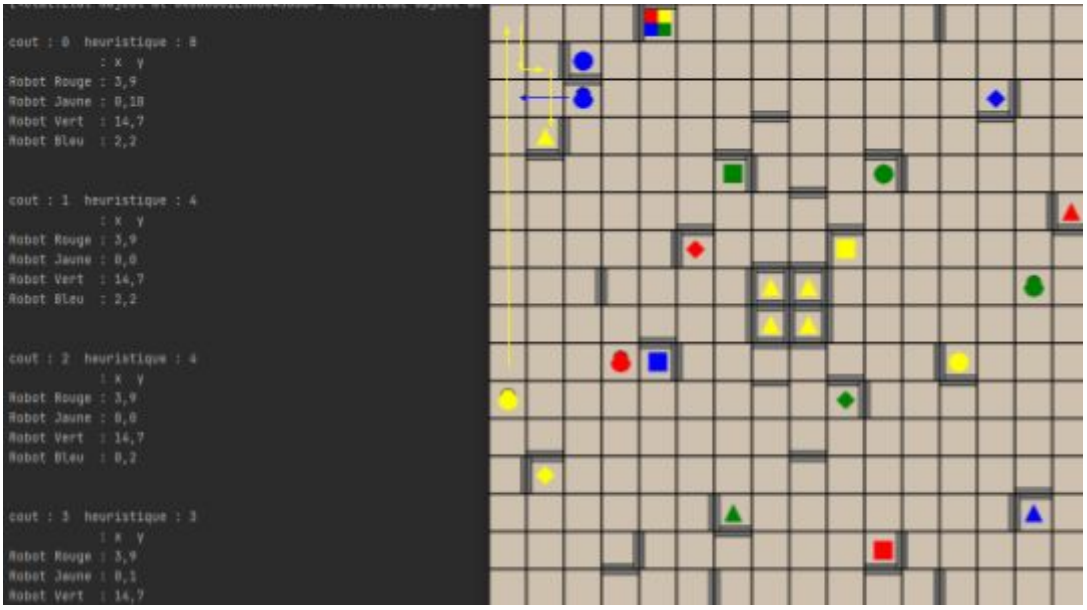

Figure 5 - Une résolution possible avec distance Euclidienne

Voilà la manière don ce problème est résolu, au lieu de déplace le robot bleu en premier et de le résoudre en deux coups, le robot jaune est bougé en premier pour surement à cause de l'heuristique, celà nous indique qu'il y a un problème d'équilibrage entre le couts, faible pour un long déplacement et l'heuristique très élevée.

3.3 Cas particulier de la Mission multicolore

Sur 20 essais, on trouve une solution 12 fois, ce qui est le mieux que l'on ai fait pour l'instant mais 4 de ces solutions ne sont pas optimales. L'explication se trouve dans la définition de l'heuristique pour la case multicolore, dans les deux cas on cherche la distance minimale pour chaque robot, ce qui revient souvent à coincer un robot non loins de la mission avant de trouver la solution avec un autre robot cf ci dessous :

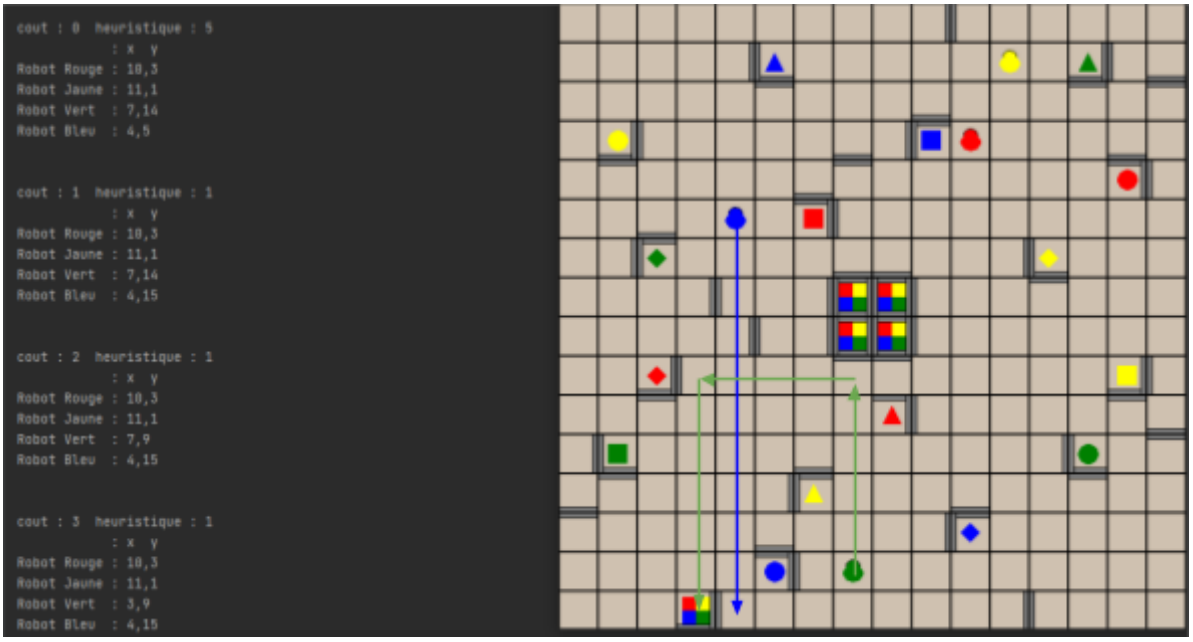

Figure 5 - Illustration du cas Multicolore

4. CONCLUSION

Dans certains cas, la répartition des murs transforme le plateau en un labyrinthe, et le robot est capable d'atteindre son but sans aide, mais dans les cas plus compliqué ou cela est impossible, la solution est moins souvent trouvée.

L'analyse a montré que nos heuristiques n'étaient pas optimales dans tous les cas. Lorsqu'aucune solution n'est trouvée, c'est souvent que le robot de la couleur de la mission est " coincé" car il ne peut pas se rapprocher de son but sans les autres robots, cela résulte en un déplacement aléatoire des autres robots. Il faudrait donc que notre heuristique prenne en compte la position des autres robots.

Pour conclure, notre rendu actuel mérite encore des améliorations pour la soutenance, comme par exemple, définir un menu de jeu pour lancer une partie avec différents joueurs, ou améliorer l'heuristique.

Enfin, nous dirons que ce projet fut pour nous une excellente façon de découvrir l'intelligence artificielle, et de travailler le langage Python, ces deux derniers seront sans aucun doute, utiles dans nos futurs apprentissages.

5. ANNEXES

```python
def Astar(initial_state,depth,occurence_test=True):
    OL=[Node(initial_state,None)]   #on ajoute l'état initial à l'OpenList
    CL=[]                           #on créé la CloseList
    n=0
    while OL:           #tant que la liste n'est pas vide
        e=OL.pop(0)     #on récupère la tete de liste
        #print(getStringPath(getPathFrom(e)))   permet d'afficher chaque noeud cherché
        if e.State.isFinalState():  #Si l'état est final
            print(getPathFrom(e), 'extending', n, 'nodes\n')    #on affiche le chemin
            print(getStringPath(getPathFrom(e)))        #on le traduit pour l'utilisateur
            break       #on sort de la boucle
        else:
            for i in range(depth):  #on effectue le nombre de fois de depth (profondeur)
                next_states=e.State.nextState()      #on récupère les états suivants
                for s in next_states:   #pour chaque état
                    if not occurence_test or s not in CL:   #si l'état n'est pas dans la CloseList
                        n+=1
                        node=Node(s,e)      #on créé un nouveau noeud
                        OL.append(node)     #On ajoute le noeud à l'OpenList
                        if occurence_test:
                            CL.append(s)    #on ajoute le Noeud à la Close List
            # on trie l'OpenList en fonction de l'Heuristique (ici la distance de Manhattan
            OL.sort(key=lambda state_: state.State.manatDist + state.State.cost*1)
```

Figure 6 - Fonction Astar()

```python
def nextState(self):
    list = []
    # toute les position que peux prendre robotRouge
    #Si le robot peut bouger vers la gauche
    if not self.plateau.plat[self.robotRouge.x][self.robotRouge.y].gauche and self.robotRouge.x > 0 and not self.isRobotHere(
            self.robotRouge.x-1, self.robotRouge.y):
        #on ajoute l'état ou le robot bouge vers la gauche à la listeet on augmente le cout de 1
        list.append(Etat(self.plateau, copy(self.robotRouge).gauche(self), self.robotJaune, self.robotVert,
                         self.robotBleu, self.mission, self.cost + 1))
    # Si le robot peut bouger vers la droite
    if not self.plateau.plat[self.robotRouge.x][self.robotRouge.y].droit and self.robotRouge.x < 15 and not self.isRobotHere(
            self.robotRouge.x+1, self.robotRouge.y):
        # on ajoute l'état ou le robot bouge vers la droite à la liste et on augmente le cout de 1
        list.append(Etat(self.plateau, copy(self.robotRouge).droite(self), self.robotJaune, self.robotVert,
                         self.robotBleu, self.mission, self.cost + 1))
    # Si le robot peut bouger vers le haut
    if not self.plateau.plat[self.robotRouge.x][self.robotRouge.y].haut and self.robotRouge.y > 0 and not self.isRobotHere(
            self.robotRouge.x, self.robotRouge.y-1):
        # on ajoute l'état ou le robot bouge vers le haut à la liste et on augmente le cout de 1
        list.append(Etat(self.plateau, copy(self.robotRouge).haut(self), self.robotJaune, self.robotVert,
                         self.robotBleu, self.mission, self.cost + 1))
    # Si le robot peut bouger vers le bas
    if not self.plateau.plat[self.robotRouge.x][self.robotRouge.y].bas and self.robotRouge.y < 15 and not self.isRobotHere(
            self.robotRouge.x, self.robotRouge.y+1):
        # on ajoute l'état ou le robot bouge vers le bas à la liste et on augmente le cout de 1
        list.append(Etat(self.plateau, copy(self.robotRouge).bas(self), self.robotJaune, self.robotVert,
                         self.robotBleu, self.mission, self.cost + 1))
    #on fait de meme pour chaque robots
    return list  # on renvoie la liste des nouveaux états
```

Figure 7 -Méthode nextState()