Dumoulin Zoe – 3702742

Team ***DisastrousGoblin***

Rapport de mini-projet
Cooperative Path-Finding

UE Intelligence Artificielle et Jeux (LU3IN025)

Introduction

Dans un problème de cooperative path-finding: on dispose d'un ensemble d'agents qui doivent chacun atteindre une destination qui leur est propre. Il s'agit de trouver un ensemble de chemins, sans collision, qui permette à chaque agent d'atteindre sa destination.

Quatre algorithmes sont mis en place et étudié : *Breadth First Search, Greedy Best First, Independent A** et *Cooperative A*,* ainsi que deux types de gestion des collisions.

Collisions

Gérées dans le *main* au moment d'un chemin bloqué par un obstacle inattendu (ici toujours un autre agent). Le chemin de l'agent en question est alors recalculé selon son algorithme de départ (par défaut A*, mais peut être *Breadth First Search* ou *Greedy Best First*). Nous avons ici deux approches : une recalculation **totale** du chemin, ou une recalculation **partielle** du chemin, appelée *local repair*.

Soit un agent P1 voulant se déplacer de la case C_{i-1} à la case C_i. C_i se trouve cependant occupée.

- **Première stratégie** : P1 recalcule son chemin partant de C_{i-1} jusqu'à son objectif C_o.
- **Seconde stratégie** : P1 recalcule son chemin partant de C_{i-1} jusqu'à une case C_{i+M}, avec M un entier arbitraire, ici M = 4, puis va insérer ce nouveau chemin dans le premier, contournant ainsi l'obstacle. L'agent P1 fait alors un détour jusqu'à retrouver son chemin originel.
  Si C_{i-1} : C_o est inférieur à M, P1 se rabat sur la première stratégie.

Il faut noter que cette seconde stratégie de *path-splicing* n'est pas vraiment utile dans notre cas et va généralement se ramener à un chemin beaucoup plus long que l'originel, ainsi qu'à un chemin moins rapide que lors d'une simple recalculation. Ce serait différent si de nouveaux obstacles apparaissaient en masse sur le chemin d'un agent.

Cependant cette stratégie est plus rapide à mettre en place, et plus efficace en ce qui concerne le temps de recherche, car elle conserve une majorité du chemin déjà calculé et n'en recalcule qu'une petite partie (qui dépend de M).

Calcul de chemins

Stratégie #1

Greedy Best First, avec collisions gérées par les stratégies vues avant.

Stratégie #2

Breadth First Search, même résultat que A* en termes de chemins trouvés.

Stratégie #3

A* indépendant au sein d'une équipe, avec collisions gérées par les stratégies vues avant.

Stratégie #4

A* coopératif au sein d'une équipe. Seules les collisions avec l'autre équipe sont gérées grâce à une recalculation du chemin.

Temps de recherche

Algo	Temps en s
Greedy BF	0.001826
Breadth FS	0.003501
A*	0.0034545
CA*	0.007666

Le plus rapide est donc *Greedy Best First.*

Breadth First Search donne les mêmes chemins que A* et est un peu plus long, nous n'allons donc pas le prendre en compte lors de nos tournois.

CA* prend le double du temps de A*,

Tournois

Nous avons testé nos algorithmes en les faisant s'affronter sur trois cartes différentes.

Une carte *« race »,* une carte *« exchange »,* et une carte *« mingle »,* comme décrit dans l'article *Adversarial Cooperative Path-finding: Complexity and Algorithms*, par Marika Ivanová and Pavel Surynek.

Ces cartes sont toutes de dimensions 20x20 et ne possèdent que six agents et objectifs.

Nous avons deux équipes de trois agents chacune.

Une partie est considérée gagnée par une équipe si ses agents ont atteint plus objectifs que ceux de l'autre équipe, ou, s'il y a égalité, si ses agents sont plus proches de leurs objectifs que l'équipe adverse.

Chaque « tournoi » est composé de 100 ou 50 parties de 50 tours chacune. Les positions initiales des agents sont affectées au hasard au début de chaque partie, selon la carte sur laquelle ils se trouvent.

Voici les résultats.

Carte *Mingle*			
team 1 : 2	Greedy BF	A*	CA*
Greedy BF	x	32 - 68	2 - 48
A*	65 - 35	x	15 - 34
CA*	40 - 10	31 - 18	x

Carte *Exchange*			
team 1 : 2	Greedy BF	A*	CA*
Greedy BF	x	24 - 75	5 - 45
A*	65 - 35	x	13 - 37
CA*	49 - 1	46 - 4	x

Carte *Race*			
team 1 : 2	Greedy BF	A*	CA*
Greedy BF	x	17 - 83	4 - 45
A*	78 - 28	x	14 - 36
CA*	45 - 5	34 - 14	x

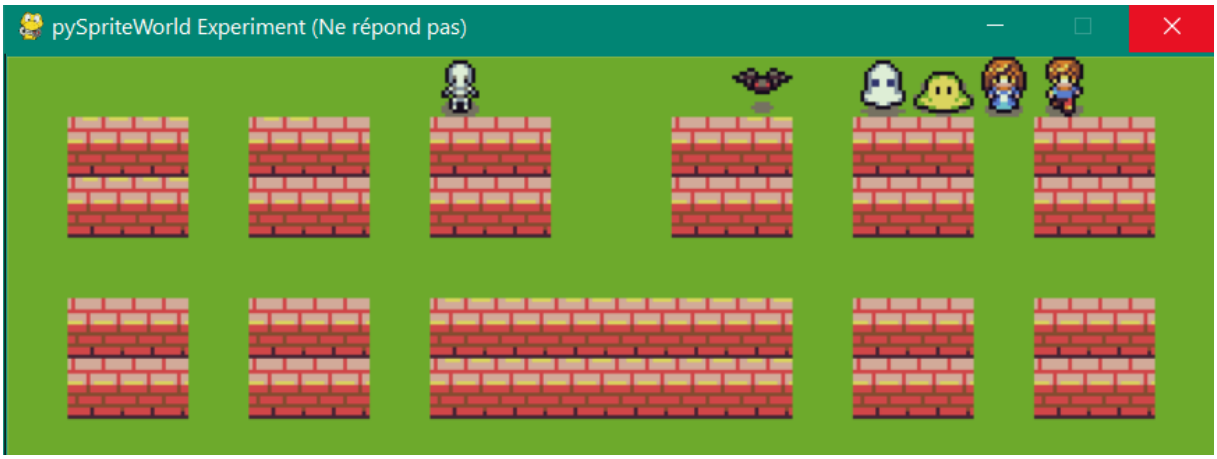
On voit bien que *Greedy Best First* ne gagne jamais contre A* et CA*.

CA* gagne tout le temps.

Dans les parties opposant *Greedy Best First* à A*, on peut remarquer que passer en deuxième apporte un avantage à l'équipe.

Situations intéressantes

Dû à ma réservation dans CA*, l'agent 1 pense que toutes les cases adjacentes à celle où il se trouve sont occupées. J'ai essayé de créer une table de réservation temporaire lors de chaque calcul de CA*, mais le jeu se freeze.



La même situation, mais cette fois-ci les deux équipes utilisent soit Greedy Best First soit A*.

Tour 1 : player 1 cannot move because player 4 at (0, 14), so -> (0, 15) -> (0,16)

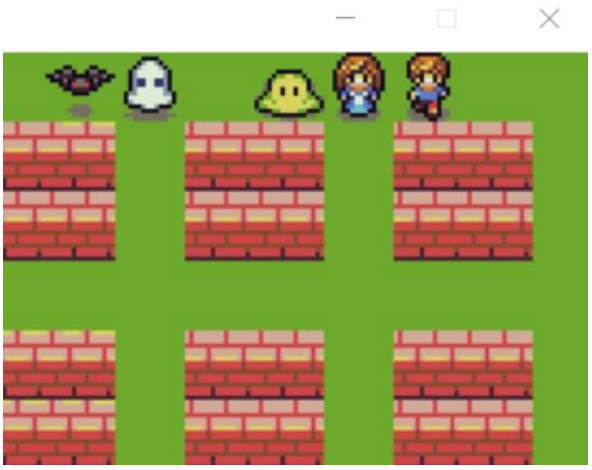
Ce problème est dû au fait que, dans mon main, au moment de gérer les collision, je mets à jour la grille en mettant des *False* à tous les emplacements occupés par des joueurs.

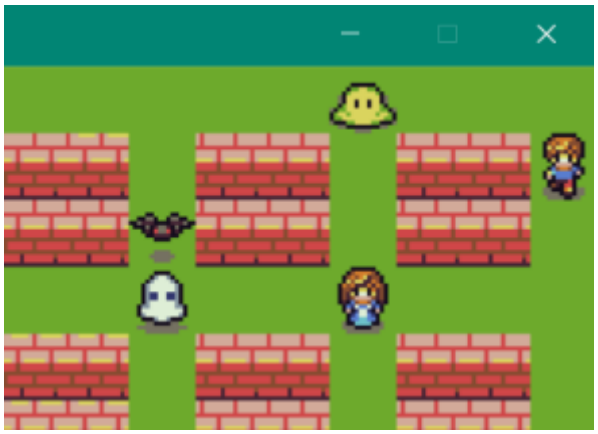
```
else:
    for c in collisions:
        g[c] = False

    probleme.collision_checking(chosen_algo1, 0, j, team1, i, g)

    for c in collisions:
        g[c] = True
```

Cela permet d'habitude de sauver du temps en collision, car si l'agent 1 en (4, 5) se trouve bloquer à sa gauche en (4, 4) et que son nouveau chemin le fait passer à sa droite en (4, 6), si cet emplacement est aussi occupé, il va devoir à nouveau rechercher un chemin. Alors que s'il sait que (4, 6) est occupé, il n'essaierait pas d'y aller.

Le problème ici est que les deux autres emplacements possibles sont soit hors-zone, soit pas légales. Aussi, lorsque l'agent 1 est à l'arrêt, il ne recalcule pas son chemin.

Une solution serait de recalculer le chemin après x itérations, pour voir si les blocages sont toujours là.





Conclusion

En conclusion, la stratégie la plus efficace en termes de chemins trouvés est CA*, mais mon implémentation de cette stratégie est source de bugs et n'est probablement pas optimale.

Aussi, dû à sa matrice des positions pour chaque temps t, cet algorithme est beaucoup plus lent dans sa recherche que A* ou encore *Greedy Best First.*

Plus la carte sur laquelle on l'applique est grande, plus cet algorithme prendra du temps et de la mémoire.