Mini-Projet

Cooperative Path-Finding

3i025

Licence d'informatique

Sorbonne Université

Année universitaire 2018-2019

Réalisé par :

Merrouche Aymen

Sidhom Imad

Table des matières :

1 Introduction :

Étant donné plusieurs agents qui possèdent tous une cible à atteindre, il est facile de trouver le meilleur chemin à suivre pour chaque agent, cependant effectuer ces chemins parallèlement peut provoquer des collisions. Le problème est donc de trouver une stratégie qui permette aux agents d'atteindre leurs cibles le plus rapidement possible tout en évitant les collisions.

Nous allons-nous intéresser dans ce qui va suivre à trois différentes stratégies pour traiter ce problème : la stratégie opportuniste de portionnement de chemins « path slicing », une stratégie coopérative de base et une stratégie coopérative avancée, en les étudiant d'abord théoriquement et en comparant ensuite leurs performances.

2 Partie théorique :

2.1 Algorithme A* :

Principe :

L'algorithme A* permet de trouver un chemin de coût minimum pour aller d'un état initial à un état but dans un graphe d'états. La procédure consiste à parcourir le graphe en étendant à chaque étape un nœud, étendre un nœud consiste à générer tous ces fils. On distingue deux ensembles de nœuds :

- La **frontière** : les nœuds qui sont susceptibles d'être étendus.
- La **réserve** : les nœuds qui ont déjà été étendus.

Reste à déterminer le critère sur lequel on se base pour choisir quel nœud de la réserve on va étendre. Il s'agit du nœud qui minimise la valeur f(n) = g(n) + h(n), g(n) est le coût du chemin pour arriver à ce nœud, et h(n) est le coût estimé (en utilisant une heuristique) pour arriver à l'état but à partir de ce nœud. Les critères de choix d'une heuristique sont les suivants :

- Une heuristique est « **admissible** » si elle ne surestime jamais la distance pour arriver à l'état but.
- Une heuristique est « **consistante** » si pour tout état fils m de n : h(m) <= cost(m,n)+h(n), dans ce cas un nœud ne peut être étendu qu'une seule fois donc un nœud de la réserve ne peut pas être réétendu, il en découle que dès qu'un nœud entre dans la réserve la distance optimale pour y arriver est connue (c'est g(n)).

Implémentation :

On utilise comme heuristique la distance de Manhattan : nombre de cases à parcourir sur un monde de type grille, c'est une heuristique admissible et consistante.

Dans notre cas, les états correspondent aux positions d'un agents qui veut atteindre un but à une position cible. Un nœud de l'arbre contient : un pointeur vers son nœud parent, la position correspondante (x,y), les valeurs de g ,h et f.

La **frontière** est un tas (heapq) : pour retirer le nœud qui minimise la valeur de f. La **réserve** est dictionnaire indexé par la position (x,y).

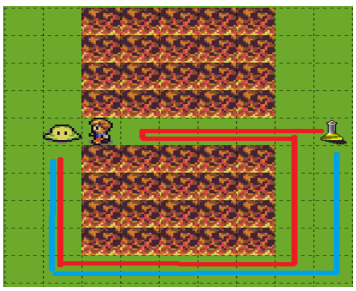
Fichier : Astar.py

2.2 Stratégie opportuniste de portionnement de chemins « path slicing » :

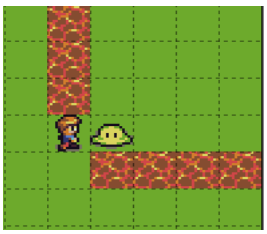
Principe :

Il s'agit de contourner l'obstacle rencontré en recalculant une portion du chemin initial. Un agent calcule au début son meilleur chemin (pour aller de 1 à N) en utilisant A* : [1,2,…,i,j,k….N], si en bougeant il se rend compte que la prochaine case qu'il va prendre contient un autre agent, par exemple il se rend compte à la case « i » que la case « j » contient un autre agent il va calculer en utilisant A* un autre chemin pour aller de i à k en considérant la case j comme illégale : [i,……,k] et contournera ainsi l'agent , son nouveau chemin devient : [1,2,…,i,…....,k….N].

Il y'a certain cas défavorables, en raison de la disposition des agents et des obstacles, ou la longueur du nouveau chemin [1,2,…,i,…....,k….N] est beaucoup plus grande que celle de l'ancien chemin, c'est-à-dire que le chemin [i,……,k] effectue un détour, ce cas peut être détecté en comparant la longueur du chemin [i,……,k] et la distance de Manhattan qui sépare i et k (si on recalcule à la case qui vient juste après c'est 2), si c'est sensiblement plus grand on préfèrera recalculer un tout autre chemin pour aller de « i » à « N ».



Il y a certains cas où cette méthode ne permet pas de trouver une solution, par exemple dans le cas suivant, si l'agent en vert souhaite sortir de l'enclos, il se rend compte que la prochaine case est occupée par un autre agent et donc il va essayer de le contourner, mais cela n'est pas possible dans ce cas.



Implémentation :

- Au début tous les agents calculent indépendamment le meilleur chemin pour atteindre leurs cibles.
- Dans la boucle principale de déplacement, avant de se déplacer vers une case, l'agent vérifie si elle n'est pas occupée par un autre agent. Si c'est le cas A* est utilisé pour trouver un chemin pour arriver à la case suivante, et l'agent met à jour son chemin.

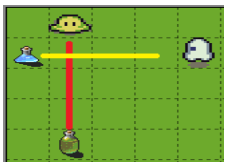
Fichier : path_slicing.py, version améliorée **: path_slicing_ameliored.py**

2.3 Stratégie coopérative de base :

Principe :

Le principe est simple, il s'agit de trouver les chemins des agents qui ne partagent aucune case, ceux-là peuvent être exécuté en parallèle, ces chemins sont organisés en groupes (les chemins qui ne se croisent pas appartiennent au même groupe), il reste à définir un ordre de passage entre les différents groupes de chemins. Une stratégie qui marche toujours est que les agents effectuent tour à tour leurs chemins (sauf dans le cas où un agent se trouve sur le chemin d'un autre, et aucun autre chemin n'est possible). Cependant il s'agit d'une stratégie très inefficace car si deux chemins se croisent cela ne veut pas dire pour autant qu'il y a un risque de collision.

Dans le cas suivant par exemple, les deux chemins se croisent donc en suivant cette stratégie les deux chemins vont être exécutées séquentiellement, il nous faudra donc 8 itérations, mais en réalité ces deux chemins peuvent être exécutés en parallèle sans générer de collisions en 4 itérations.



Implémentation :

- Au début tous les agents calculent indépendamment le meilleur chemin pour atteindre leurs cibles.
- A partir des chemins calculés en utilisant A*, on va construire des classes de chemins. Les chemins qui ne se croisent pas appartiennent à la même classe.
- Dans la boucle principale de déplacement, on définit un ordre de passage entre les classes et on exécute parallèlement les chemins de chaque classe.

Fichier : strategie_cooperative_base.py

2.4 Stratégie coopérative avancée :

Principe :

Carte spatio-temporelle :

L'idée est d'introduire une nouvelle dimension, le temps t. Il faut donc une nouvelle structure pour représenter les position des agents et des trésors à un instant t donné, on utilisera une carte spatio-temporelle (x,y,t) , qui indique la position de chaque agent à un instant donné, les positions des fioles et des obstacles ne changent pas en fonction du temps. On devra donc appliquer notre algorithme A* (qui devra être revisité en A* spatio-temporel) sur cette structure pour déterminer le plus court chemin de chaque agent. On ajoute une actions possible aux agents : c'est de rester en place ((x,y,t+1) est un état fils de (x,y,t)).

Table de réservation :

Pour résoudre le problème de collisions entre les agents on utilise une structure partagée, qui indique à tout moment les emplacements « légaux » (un emplacement (x,y,t) est légal s'il ne contient pas d'agent et pas d'obstacle),une table de réservation spatio-temporelle que chaque agent doit prendre en considération en calculant à l'aide du A* spatio-temporel son chemin et la mettre à jour après. Par exemple si un agent veut occuper une place (x,y) à un instant donnée t, il vérifiera d'abord que c'est une positon légale, il devra ensuite rendre illégales dans la table de réservation les cases (x,y,t) et (x,y,t+1), naïvement seule la case (x,y,t) devrait être rendu illégale, en suivant cette stratégie si un agent réserve la case (x,y,t) et (x+1,y,t+1), un autre agent peut tout à fait réserver la case (x+1,y,t) et (x,y,t+1) et faire une permutation avec collision.

Déroulement :

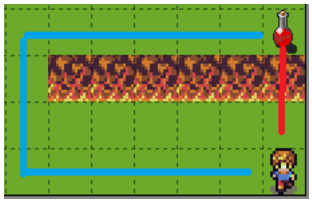
Chaque agent calcule à l'aide du A* spatio-temporelle et de la table de réservation son chemin, et met à jour la table de réservation. Les chemins obtenus à la fin ne se croiseront pas et pourrons être exécutés parallèlement.

A* spatio-temporel :

Choix d'une heuristique :

Dans la version spatial du A*, on utilisait la distance de Manhattan (nombre de cases à parcourir sur un monde de type grille) il s'agit d'une heuristique consistante (i.e. pour tout état m et tout fils n de m

h(m)≤cost(m,n)+h(n)) , c'est une heuristique qui fonctionne bien sur des grilles ouvertes, par contre dans le cas où le plus court chemin est détourné (la cible n'est pas accessible à cause d'obstacles et donc il faut prendre un détour), cette heuristique ne fonctionne pas très bien et on aura plein de nœuds avec des valeurs de f qui sont inférieurs à la distance réelle car cette heuristique ne prend pas en considération les obstacles, et donc le nombre d'états explorés explose. Ce phénomène va s'amplifier dans la version spatio-temporelle de A*, puisque chaque état va être caractérisée en plus de la position visitée (x,y), du temps t, et donc s'arrêter ou rebrousser chemin va paraitre plus optimal et on revisitera les nœuds qui sous-estiment la vrai distance à la cible.



L'heuristique « True Distance » :

Cette heuristique consiste à prendre en considération les obstacles dans l'estimation de la longueur du chemin mais ignore les agents, autrement dit elle renvoie exactement la longueur du chemin que renvoie le A* spatial, et donc si aucun agent n'interfère sur le chemin, le A* spatio-temporel procédera directement via le chemin le plus court, dans le cas contraire d'autres états vont être explorés, ce qui dépendra du nombre d'agents dans la grille. C'est une heuristique admissible et consistante. Mais comment calculer cette heuristique ?

Quand on utilise une heuristique consistante avec A*, dès que le nœud est mis dans la réserve, la distance optimale est connue, c'est la distance parcourue par A* (i.e. g). Supposons qu'on lance un A* spatial de la cible vers la destination en utilisant l'heuristique de Manhattan puisque c'est une heuristique consistante on connaîtra la réelle distance de la cible à chaque nœud de la réserve (c'est le g), et donc on a trouvé un moyen de calculer notre heuristique. L'algorithme A* s'arrête dès qu'il trouve la cible, et donc on aura peut-être des nœuds qui ne seront pas mis en réserve et donc l'heuristique ne sera pas calculée pour ces nœuds, il suffit donc de modifier A* pour qu'il s'arrête quand nécessaire (par exemple jusqu'à ce qu'une certaine position soit mise en réserve).

Déroulement du A* spatio-temporel :

On effectue la recherche de chemin de la position initiale à la cible, à chaque position explorée, l'heuristique « True distance » est requise, on utilise une recherche auxiliaire en utilisant le A* spatial de la cible vers la position initiale, si la position dont l'heuristique est nécessaire est dans la réserve la valeur g est renvoyée, sinon le A* spatial est relancé jusqu'à ce que cette position soit ajoutée à la réserve.

Implémentation :

La table de réservation est une liste qui contient toutes les positions (x,y,t) interdites (qui sont occupées par un agent).

A* spatio-temporel :

La structure du nœud est modifiée, on rajoute dans la position le temps t. En étendant un nœud les fils possibles sont (x+1,y,t+1), (x,y+1,t+1), (x-1,y,t+1), (x,y-1,t+1) et (x,y,t+1), on s'assurera en utilisant la table de réservation que toutes les position sont légales, à la construction du chemin on enrichira la table de réservation. Pour calculer l'heuristique « true distance » on crée une classe qui s'occupera de l'exécution d'un A* spatial de la cible à la destination avec une méthode getTrueDistance(x,y), cette méthode renvoie directement la valeur g si (x,y) est déjà dans la réserve , sinon elle reprend l'exploration jusqu'à ce qu'il soit ajouté à la réserve.

Fichier : en utilisant l'heuristique de **Manhattan : space_time_Astar_Manhattan.py**, en utilisant l'heuristique **« true distance » : space_time_Astar_True_distance.py.**

Tout le travail est effectué dans l'algorithme A* spatio-temporel :

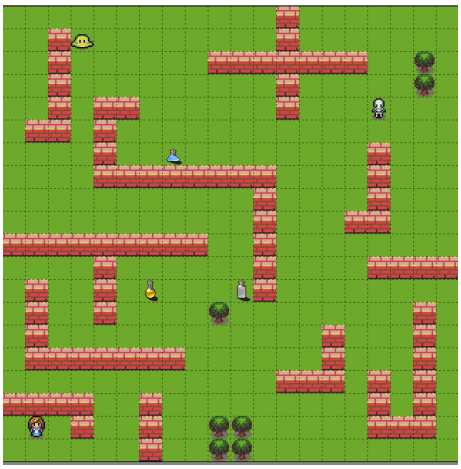
- Au début tous les agents calculent indépendamment le meilleur chemin pour atteindre leurs cibles en utilisant la A* spatio-temporel.
- Dans la boucle principale de déplacement, les agents effectuent leurs chemins parallèlement.

Fichier : en utilisant l'heuristique de **Manhattan : strategie_cooperative_avancee_manhattan.py**, en utilisant l'heuristique **« true distance » : strategie_cooperative_avancee_ true_distance.py**.

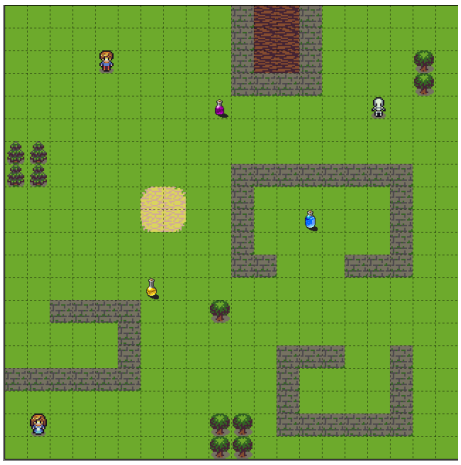
3 Comparaison entres les stratégies :

3.1 Cartes de test :

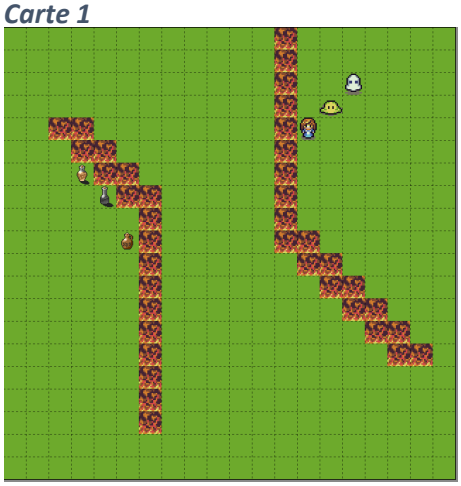
Fichiers: pathfindingWorld_MultiPlayer « i ».json



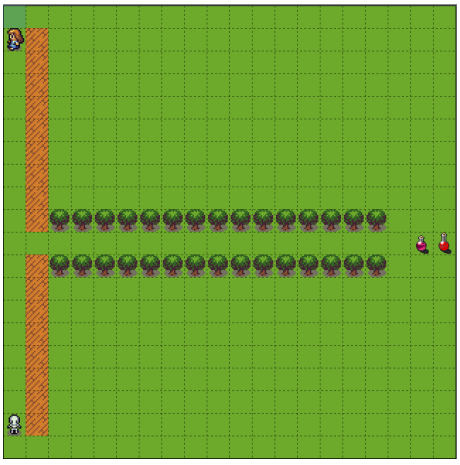
Carte 1



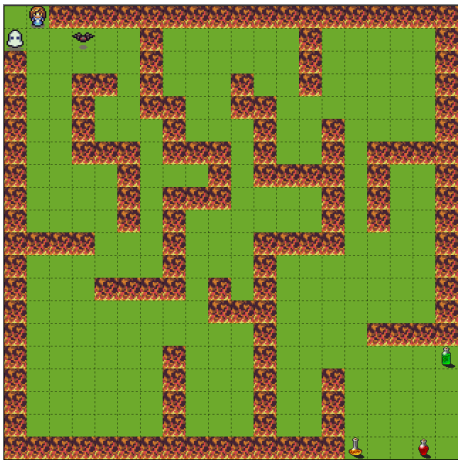
Carte 2



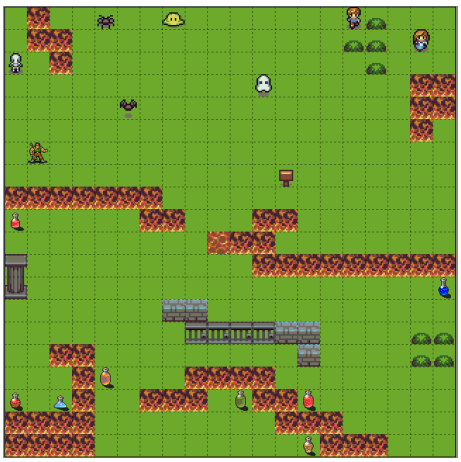
Carte 3



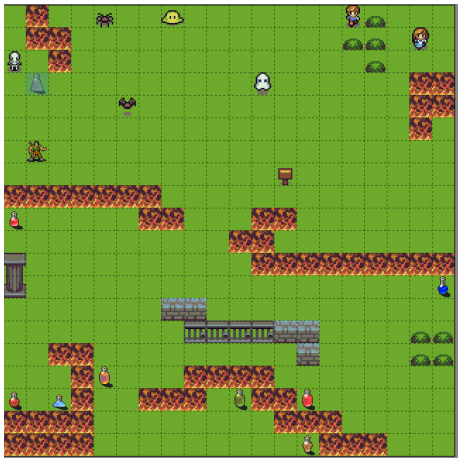
Carte 4



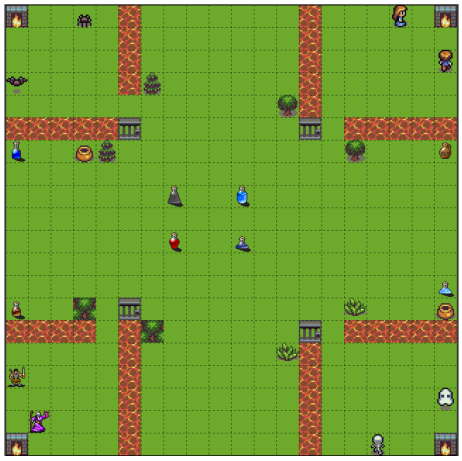
Carte 5



Carte 6



Carte 7



Carte 8

3.2 Résultats :

Borne inf : donne une borne inf sur le nombre d'itérations, correspond au coût max des chemins optimaux calculés de manière individuelle (i.e. sans se préoccuper des autres agents). **Fichier : allAstar.py**

Borne sup : donne une borne sup sur le nombre d'itérations, correspond à la somme des coûts max des chemins calculés de manière individuelle (i.e. séquentiellement). **Fichier : allSequentiel.py**

La carte	Critère	Borne inf	Path slicing	Stratégie coopérative de base	Stratégie coopérative avancée - Manhattan	Stratégie coopérative avancée – True distance	Borne sup
Carte1	Possible	Oui	Oui	Oui	Oui	Oui	Oui
	Itérations	21	21	39	21	21	49
	Temps de routage	0.0409	0.021	0.062	0.197	0.044	0.0838
Carte2	Possible	Oui	Oui	Oui	Oui	Oui	Oui
	Itérations	19	19	19	19	19	39
	Temps de routage	0.0324	0.0141	0.068	0.153	0.043	0.060
Carte3	Possible	**Non**	Oui	Oui	Oui	Oui	Oui
	Itérations	49	52	140	49	49	140
	Temps de routage	0.1781	0.295	0.301	3.75	0.535	0.272
Carte4	Possible	**Non**	Oui	Oui	Oui	Oui	Oui
	Itérations	28	78	56	30	30	56
	Temps de routage	0.0322	0.081	0.0056	0.052	0.09	0.016
Carte5	Possible	**Non**	**Non**	Oui	Oui	Oui	Oui
	Itérations	62	xx	182	68	68	182
	Temps de routage	0.226	xx	0.149	5.05	0.249	0.149
Carte6	Possible	**Non**	**Non**	Oui	Oui	Oui	Oui
	Itérations	35	xx	228	48	46	228
	Temps de routage	0.284	xx	0.392	6.14	2.52	0.294
Carte7	Possible	**Non**	Oui	Oui	Oui	Oui	Oui
	Itérations	35	37	176	38	42	228
	Temps de routage	0.380	0.331	0.320	2.89	1.064	0.287
Carte8	Possible	**Non**	**Non**	Oui	Oui	Oui	Oui
	Itérations	25	xx	68	29	28	151
	Temps de routage	0.157	xx	0.171	0.612	0.305	0.154

3.4 Analyse :

Critères de comparaison :

Possible : si la stratégie permet aux agents de récupérer leurs fioles sans collisions.

Itérations : le nombre d'itérations de la boucle principale de déplacement pour que chaque agent atteigne sa fiole.

Temps de routage : Le temps cpu nécessaire pour le calcul de chemins dans chaque stratégie.

Remarque : Toutes les cartes qui ne sont pas possible en utilisant la stratégie « borne inf », provoquent des collisions entre les agents.

Path slicing :

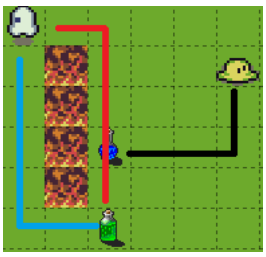
- Cette stratégie ne permet pas de résoudre les cartes 5,6 et 8 à cause de problèmes d'interblocages.
- Pour la carte numéro 4 le nombre d'itérations nécessaire est de 78, ce qui est même plus mauvais que la « borne sup » car dans ce cas un des agents va réaliser un grand détour pour éviter l'autre, il va donc suivre un chemin très couteux, (on obtient 39 avec la version améliorée).
- Lorsqu'elle permet de trouver une solution, cette stratégie a un bon temps de routage.

Stratégie coopérative de base :

- Cette stratégie est très inefficace en termes de nombres d'itérations, par exemple pour la carte 1 qui ne provoque pas de collisions ,39 itérations sont nécessaires contre 21 itérations pour la stratégie « borne inf », car tous les chemins qui s'intersectent ne provoquent pas pour autant des collisions. Pour les cartes 3,4,5 et 6 elle renvoie le même nombre d'itérations que la « borne sup ».
- Cette stratégie a un bon temps de routage, car la construction de classes ne prend pas beaucoup de temps avec une grille de petite taille.

Stratégie coopérative avancée :

- Cette stratégie permet de résoudre toutes les cartes avec un nombre d'itérations optimal, proche de la « borne inf ».
- **Avec distance de Manhattan :** Le temps de routage explose avec les cartes (3,5,6,7) ou les agents doivent prendre des chemins détournés, car l'heuristique de Manhattan donne de très mauvaises estimations dans ces cas-là.
- **Avec « True distance » :** Utiliser cette heuristique permet d'améliorer grandement le temps de routage pour cette stratégie.
- **Pourquoi Manhattan et « True distance » donnent des nombres d'itérations non-identiques pour les cartes 6,7 et 8 ?**
  L'utilisation de A* spatio-temporel avec ces deux heuristiques donnent toujours un chemin de coût minimum, mais ce chemin peut ne pas être unique et le choix entre plusieurs possibilités influence sur les autres agents. Par exemple les deux chemins en rouge et en bleu sont de coût minimum pour l'agent en blanc, mais le choix de l'un ou de l'autre influencera sur le chemin de l'agent en vert, l'heuristique influe sur ce choix.



4 Conclusion

Au terme de ce mini-projet nous avons pu comparer l'efficacité de trois stratégies différentes résolvant un même problème « la recherche de chemin coopérative ». Malgré toutes les hypothèses que nous avons posées, résoudre ce problème reste assez complexe. Dans d'autres situations où l'information n'est pas partagée, que les agents n'ont pas les mêmes priorités ni la même vitesse et avec des cibles qui bougent, d'autres mécanismes doivent être mis en place pour régler ce problème.