Compte-rendu mini-projet 3I025

© Rexha Sabina, Franceschetti Maël.

Stratégie 1 : Opportuniste

La première stratégie repose sur un principe de portionnement du trajet, si un autre joueur gêne le passage, notre joueur va recalculer une partie de son trajet pour contourner ce joueur gênant.

Entrée [ ]:
```
# /  Extrait du main() effectuant l'esquive des joueurs sur le trajet /
[...]
x_inc,y_inc = trajetPlayers[j][etapeTrajet[j]] #le déplacement prévu par le joueur
etapeTrajet[j] = etapeTrajet[j]+1
next_row = row+x_inc
next_col = col+y_inc
notMoved = True
while(notMoved):#on actualise le chemin si besoin jusqu'à pouvoir faire un mouvement valide
    if ((next_row,next_col) not in wallStates) and ((next_row,next_col) not in posPlayers)
    and next_row>=0 and next_row<=19 and next_col>=0 and next_col<=19:
        #... S'il n'y a pas de joueur sur la case ciblée, on se déplace... #
        [...]
        notMoved = False
    else:
        #Si il y a un joueur sur la case, on cherche un autre itinéraire...

        #on considère temporairement la case ciblée comme un obstacle :
        tempWallStates = wallStates[:]
        tempWallStates.append((next_row,next_col))

        #on recalcule la fin du trajet entre le joueur et le but pour contourner l'obstacle :
        prob = Problem(posPlayers[j],goalsPlayer[j], tempWallStates, hauteur=20, largeur=20)
        trajet, way = astar(prob)

        #on écrase l'ancien trajet:
        trajetPlayers[j] = trajet

        #on prépare le 1er déplacement qui sera effectué à la prochaine itération du while:
        etapeTrajet[j] = 0
        x_inc,y_inc = trajetPlayers[j][etapeTrajet[j]]
        etapeTrajet[j] = etapeTrajet[j]+1
        next_row = row+x_inc
        next_col = col+y_inc
[...]
```

Nous avons fait le choix de recalculer toute la fin du trajet jusqu'au but à partir de la position actuelle pour éviter de rallonger le trajet total dans le cas où on aurait recalculé seulement une portion de trajet.

En effet, A* retourne le trajet le plus court jusqu'au but, alors qu'en recalculant une portion du trajet pour contourner les autres joueurs et se rattacher à la fin du trajet initialement prévu, on serait parfois ammenés à faire des détours plus conséquents.

Stratégie 2 : Coopérative de base

La seconde stratégie repose sur une exécution en parallèle des trajets totalement compatibles (aucune case en commun), et une gestion de l'ordre de passage des joueurs.

Entrée [ ]:
```
# /  Extrait de la méthode getGroupesCompat(wallStates, posPlayers, goalsPlayer)
#    effectuant la répartition des joueurs pas groupes compatibles,
#    c'est-à-dire dont les trajets peuvent s'exécuter en parallèle                /
[...]
groupes = []
trajets = []
actions = []
dureeGroupes = [] #longueur du plus long trajet du groupe
dureeMaxGroupes = []
#longueur des plus longs trajets des groupes temporaire (avec le nouveau joueur)
for player in range(len(posPlayers)):#pour chaque joueur
    addedToGroup = False
    createNewGroup = True
    if len(groupes)>0:#si on a au moins un groupe
        indiceMeilleur = -1
        [...]
        for groupe in range(len(groupes)):
            wallsGroupe = wallStates[:]#on prend en compte les murs
            for playersInGroup in range(len(groupes[groupe])):
                #on ajoute temporairement le trajet des joueurs du groupe comme obstacles
                wallsGroupe += trajets[playersInGroup]
            prob = Problem(posPlayers[player],goalsPlayer[player], wallsGroupe,
            hauteur=20, largeur=20)
            # on cherche s'il existe un trajet compatible pour le joueur :
            actionsPlayer, wayPlayer = astar(prob)
            if actionsPlayer != False and wayPlayer != False:
                #si le trajet du nouveau joueur est compatible avec ce groupe
                #on calcule la durée totale d'exécution des trajets du groupe :
                dureeMaxGroupes[groupe] = max(dureeGroupes[groupe], len(actionsPlayer))
                indiceMin = groupe
                #on calcule le temps d'exécution qui serait ajouté par ce joueur:
                ecartActuel = dureeMaxGroupes[groupe] - dureeGroupes[groupe]
                ecartMin = ecartActuel
                for grp in range(len(groupes)) : #on compare avec tous les groupes
                    ecartGroupe =  dureeMaxGroupes[grp] - dureeGroupes[grp]
                    if ecartGroupe < ecartMin :
                        indiceMin = grp
                        ecartMin = ecartGroupe
                if ecartActuel == ecartMin :
                    #si on a trouvé le meilleur groupe actuel, on le choisit
                    indiceMeilleur = groupe
                    [...]
                    addedToGroup = True
        if addedToGroup :
            prob = Problem(posPlayers[player],goalsPlayer[player], wallStates, hauteur=20,
            largeur=20)# on calcule un trajet du joueur seul
            actionsPlayer, wayPlayer = astar(prob)
            ecartGroupe = dureeMaxGroupes[indiceMeilleur] - dureeGroupes[indiceMeilleur]
            #si le joueur ralonge la durée totale d'exécution du programme en étant
            #dans le meilleur groupe plutôt qu'en faisant son trajet tout seul
            #après les autres, on le met dans un groupe à part :
            ecartSeul = len(actionsPlayer)
            if ecartSeul < ecartGroupe :
                #on va créer un nouveau groupe
                createNewGroup = True
            else:
                #sinon, on associe le joueur au meilleur groupe
                createNewGroup = False
                groupes[indiceMeilleur].append(player)
                #on mémorise le trajet du nouveau joueur
                [...]
                if dureeGroupes[groupe] < len(actionsPlayer):
                    dureeGroupes[groupe] = len(actionsPlayer)
                    dureeMaxGroupes[groupe] = len(actionsPlayer)

    if len(groupes)==0 or createNewGroup == True:
        #on créé un nouveau groupe avec ce joueur
        [...]
```

Nous avons fait le choix de regrouper de manière optimale les joueurs pouvant exécuter leurs trajets simultanément.

Les joueurs sont regroupés de façon à ne pas pénaliser la durée totale d'exécution du programme (nombre d'itérations).

Stratégie 3 : Coopérative avancée

La troisième stratégie repose sur la gestion du temps : on considère dorénavant une collision quand deux joueurs sont sur une même case en même temps.

Il faut dont tenir une table de réservation des cases pour savoir quand elles sont disponibles.

Entrée [ ]:
```
# / Classe permettant de réserver les cases et vérifier leur disponibilité, prototypes des méthodes: /
class ReservationTable():

    def __init__(self, largeur, hauteur, time)

    def is_Blocked(self,l ,h, t)

    def checkAvailableCase(self,l, h, t)

    def block_the_Case(self,l,h,t)

    def block_forever(self, l, h, t)

    def checkAvailableCaseForever(self,l, h, t)

    def searchNearestStayPlace(self,l, h, t, wallStates)


# / Extrait de la méthode getTrajetsCompat(wallStates, posPlayers, goalsPlayer) qui calcule
#    des trajets sans collision temporelle                                              /
[...]
trajetTrouve = False
casesCollisions = []
actionsPlayer = []
wayPlayer = []
while trajetTrouve==False : #on itère tant qu'on a pas trouvé de trajet
    wallsGroupe = wallStates[:]#on prend en compte les murs
    wallsGroupe += casesCollisions
    prob = Problem(posPlayers[player],goalsPlayer[player], wallsGroupe,
    hauteur=20, largeur=20)
    actionsPlayer, wayPlayer = astar(prob)
    if actionsPlayer != False and wayPlayer !=False:
        #on a trouvé un trajet
        trajetTrouve = True
        for i in range(len(wayPlayer)):
            case = wayPlayer[i]
            #on vérifie qu'il n'y a pas de collision temporelle :
            if tableResa.checkAvailableCase(case[0],case[1], i)==False :
                #case occupée...
                [...]
                trajetTrouve = False
                casesCollisions.append(case)#on on considère qu'on doit éviter cette
                #case pour trouver un trajet sans collision temporelle
    else:
        #pas de solution
        [...]
lasti = 0
for i in range(len(wayPlayer)-1):
    #on réserve les cases utilisées par notre trajet
    case = wayPlayer[i]
    tableResa.block_the_Case(case[0],case[1],i)
    lasti = i
#on vérouille la case sur laquelle le joueur s'arrête à la fin de son trajet:
derniereCase = wayPlayer[len(wayPlayer)-1]
tableResa.block_forever(derniereCase[0],derniereCase[1], lasti+1)
[...]
```

Nous avons choisi de calculer un nouveau trajet tant qu'il en existe jusqu'à trouver un trajet sans collision.

Pour ce faire, nous réservons les cases utilisées par les joueurs à un instant 't' dans une table des réservations, et nous ne validons que les trajets qui n'utilisent pas de cases réservées à un même instant 't'.

Quand un trajet utilise une case indisponible à un moment 't' , on la note comme un obstacle et on calcule un nouveau trajet. Nous avons choisi cette implémentation simple plutôt que de modifier A* car les collisions temporelles sont rares.

Quand un joueur a fini son trajet et ne bouge plus, on considère qu'il occupe la case jusqu'à la fin du programme, on la vérrouille donc définitivement. Si ce vérouillage définitif n'est pas possible car un autre joueur a déjà prévu de passer sur cette case plus tard, on décale notre joueur pour qu'il ne gène pas.