Compte Rendu Projet Algo1

BIHET Lucie

Professeur encadrant : M. GOASDOUE

Septembre - Novembre 2024

ENSSAT
LANNION

Table des matières

1 Introduction

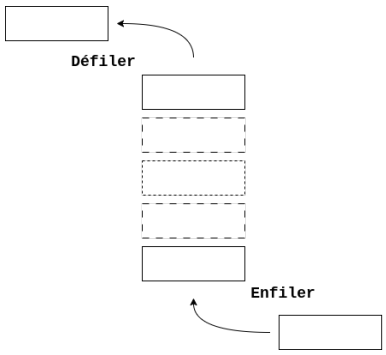
Le but de ce projet est de coder en langage C une IA (Intelligence Artificielle) d'un joueur de Lode Runner. Le moteur de jeu est fourni, il faut uniquement implémenter l'IA du joueur qui fonctionne au tour par tour en fonction de paramètres fournis : position du joueur, position des bonus, des ennemis et caractéristiques de la carte. Il faudra alors choisir entre les actions suivantes : aller à gauche, à droite, en haut, en bas, poser une bombe à droite ou à gauche. Le but du jeu est pour le joueur de récupérer les bonus du niveau tout en évitant les ennemis puis une fois tous les bonus récupérés, se rendre vers la sortie. Le joueur peut emprunter des échelles et des câbles mais ne peut pas traverser les murs sauf si c'est un trou causé par une de ses bombes. Il faut donc que l'IA arrive à finir le niveau sans mourir tout en respectant et en utilisant à son avantage les règles du lode runner.

Ainsi, à chaque tour, à l'aide des données transmises du jeu en cours l'IA doit décider de l'action à réaliser. Aucune donnée ne peut être récupérée des actions précédentes ni gardées pour une action futur, le joueur ne peut agir que pour un état T figé du jeu. On a donc : la carte du jeu représentée par une matrice de caractères, avec sa taille et les coordonnées de la sortie, une liste des bonus restants à trouver, une liste des entités (joueur et ennemis) du jeu (avec leur coordonées) et une liste de la position des bombes posées.

L'objectif est donc de coder une IA qui puisse à la fois trouver tous les bonus et se rendre vers la sortie tout en restant en sécurité mais de manière efficace. Il faut donc arriver à allier optimisation du chemin et assurer la survie du joueur. C'est pour cela que mon programme va d'abord chercher les bonus, puis prioritiser à la fois les plus proches, mais aussi les plus safes.

2 Préliminaires

File et file de priorité : Une file est une structure de données basée sur le principe de "Fisrt In First Out" (FIFO). Le premier élément qui a été entré dans la file sera le premier qui en sortira. La file de priorité ajoute le concept de trier les élément de la file selon un critère de priorité, plus ce critère est grand plus il est loin dans la file, c'est à dire qu'il sera sorti plus tard. Les files possèdent plusieurs fonctions et procédure qui permettent de travailler efficacement avec. La structure est composée initialement simplement d'un pointeur vers la tête de file et chaque élément pointe vers son voisin. Ici, les éléments enfilés sont des coordonnées qui ont un coût et un pointeur vers la coordonnée suivante pour la file.



creerFile : fonction qui permet d'initialiser et d'allouer la mémoire pour une file vide

estVideFile : fonction qui renvoie vrai si une liste est vide

enfilePrioritaire : procédure qui enfile un élément dans la file avec le critère de priorité

enfile : procédure qui enfile un élément sans le critère de priorité

defile : fonction qui renvoie le prochain élément de la file

dansFile : fonction qui vérifie si un élément est dans la vile, si oui il renvoie son coût, sinon il renvoie -1

retirerFile : procédure qui retire un élément de la file sans tenir compte de l'ordre de la file et libère la mémoire allouée

Algorithme A* : C'est un algorithme de recherche de chemin entre un point de départ et un point d'arrivée dans un graphe. Chaque noeud possède une heuristique qui permet d'estimer la distance de ce noeud au noeud d'objectif. On utilise alors cette heuristique pour privilégier quel noeud visiter.

1. Initialisation : On démarre du noeud de départ qu'on ajoute à une file

d'attente prioritaire (open list).

2. Traitement : Le nœud à l'heuristique la plus basse est retiré de l'open list :
- Si c'est le but, on reconstruit le chemin.
- Sinon, ses voisins admissibles sont évalués : leur coût total est calculé, et ils sont ajoutés à l'open list, sauf si un meilleur chemin existe déjà.

3. Répétition : Le nœud traité est déplacé vers une liste vérifiée (closed list). Le processus continue jusqu'à trouver un chemin ou vider l'open list.

Parcours en largeur (BFS) : C'est un algorithme de parcours de graphe qui permet de visiter les noeuds d'un graphe. Pour chaque noeud visité, on le marque et on liste ses voisins pour les visiter. Puis on prend le premier noeud de la liste et on recommence jusqu'à avoir tout visité. Ici on arrête la recherche quand on trouve un bonus car le but du BFS ici est de trouver le plus court chemin vers un bonus.

Ces deux algorithmes sont utilisés avec des graphes, ici j'ai fait le choix de ne pas transformer la map en graphe mais j'ai "simulé" un graphe à l'aide de :

- chaque case accessible est un noeud et possède 4 voisins mais seulement certains seront légalement accessible, donc au lieu d'avoir les voisins associés au noeud initialement, je les vérifie quand j'en ai besoin
- un tableau qui trace les parents de noeuds visités pour pourvoir remonter le parcours
- chaque arête entre deux cases où on peut passer à une valeur de 1
- finalement, on parcours la map comme on parcourrait un graphe mais la structure à proprement parlé n'est pas implémentée

Structure Coord : Cette structure permet de stocker un duo de coordonnées x et y, mais également le pointeur vers l'élément suivant (utile pour les files) et le coût associé à cette coordonnée (utile pour le A*). Chacun de ces paramètres n'est pas utile pour chaque fonction mais il était plus pertinent de ne faire qu'une seule structure qui marche dans tous les cas souhaité.

3 Stratégie générale

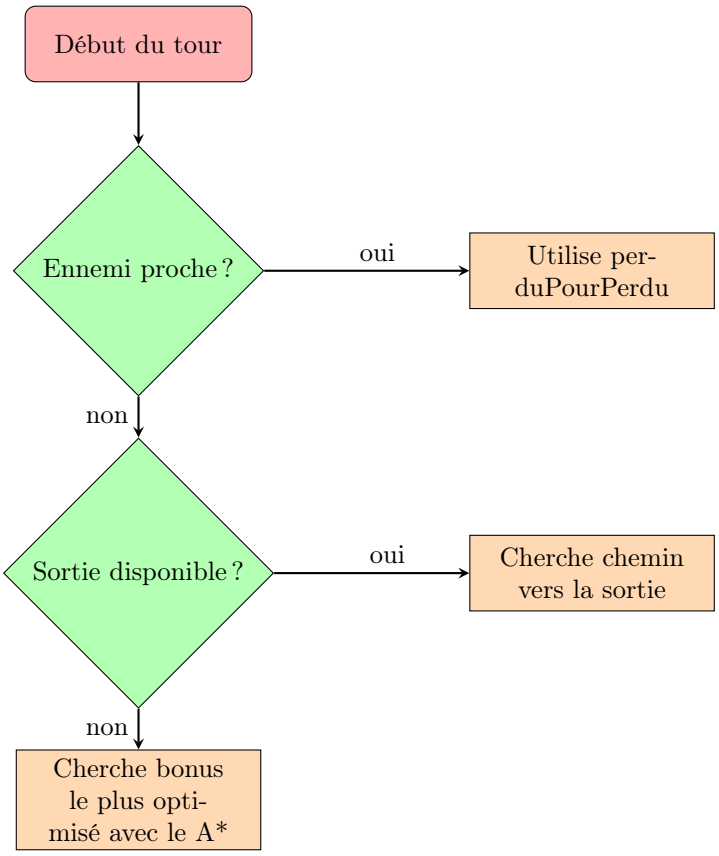


FIGURE 1 – Graphe de la stratégie générale

Explication de la stratégie en précision :

Calcul d'heuristique : J'ai choisi l'heuristique comme étant la somme de la distance de la case au bonus le plus proche auquel on additionne la dangerosité de la case avec heuristicCalculation.
La distance au bonus est calculée avec un BFS pour assurer la plus courte distance. Le BFS est accompagné de checkPaths qui pour chaque case visitée vérifie si ses voisins sont accessibles et de retraceRouteForBFS qui retrace la route du bonus vers la case de départ tout en marquant chaque case avec sa distance au bonus.
La dangerosité est calculée par rapport à la précense des ennemis : si un ennemi est repéré alors sa case et les cases à 3 de distance autour gagne en coût heuristique avec updateDangerHeuristic.

A* : C'est l'algorithme principal pour les déplacements du runner. Il utilise deux files, des tableaux pour le coût des cases et des déplacements, un tableau avec les heuristiques de chaques cases et un tableau pour marquer une case comme visitée et indiquer sa case parent.
La fonction aStarAlgorithm alloue en mémoire tous les éléments nécessaires puis vérifie si un ennemi est proche, puis vérifie si la sortie est disponible et sinon, l'algorithme cherche le bonus le plus optimal à atteindre en terme de coût du chemin. Pour chaque case, testNeighbors permet de vérifier si les cases voisines sont accessibles ou non et si oui, processNeighbor gère ce qu'on fait du voisin selon la situation. Si la case visitée par le A* est le bonus on retrace le chemin avec retracePathForAStar et on renvoie l'action qui est la première étape du chemin.
Le chemin vers le bonus le plus "proche" en terme de distance et de sécurité est donc recalculé à chaque instance du jeu selon les évolutions de la partie.

Main : Le main permet d'obtenir les coordonnées du runner et de mettre les bombes, bonus et ennemis sur la map afin d'avoir une vision globale de l'état de la map plus facilement.
C'est aussi la fonction qui permet de lancer le A* et de renvoyer l'action à faire par le runner.

Gestion des ennemis : Il y a principalement deux fonctions pour gérer les ennemis : isEnenmyNear qui vérifie si un ennemi est à une distance de Manhattan de 2 cases du joueur (comme le joueur est deux fois plus rapide que les ennemis cela lui laisse le temps de réagir) et perduPourPerdu qui est utilisée quand au moins un ennemi est proche du runner (elle est appelée aussi en cas de dernier recours dans le A* si rien n'est possible).
perduPourPerdu essaie de couvrir un maximum de situation dangereuse et d'agir en conséquence :
- si un ennemi est 2 cases à gauche ou à droite on s'en débarrasse immédiatement avec une bombe
- sinon on regarde si dans une des 4 directions (left, right, up et down) il y n'y a pas d'ennemi dans les 3 prochaines cases (cela permet de laisser le temps d'agir en cas de présence d'ennemis après avoir avancé) et on se dirige donc dans la direction où les 3 prochaines cases n'ont pas d'ennemis
- sinon on pose une bombe dans la direction où l'ennemi est le plus proche à gauche ou à droite
- sinon on teste chaque case adjacente et on va vers celle où il n'y a pas d'ennemis adjacent
- si rien de tout cela est possible, on ne fait rien et on a perdu

4 Trois modules centraux

Précisions :
- J'ai réalisé dans le pseudo code les "libère" en une seule ligne mais en réalite, chaque objet a son propre module pour libérer sa mémoire, j'ai fait ce choix par soucis de clarté pour le pseudo-code - J'ai remplacé les ↑ par des ? pour qu'ils soient affiché correctement - ongoing est en fait la structure coord pour les coordonnées x et y à chaque fois donc ongoing = (x, y) mais par soucis de simplicité j'utilise x et y ou ongoing selon les cas

Algorithme A* :

- C'est un fonction car elle doit renvoyer une "action", celle que le runner effectuera à ce tour.

- En entrée on a besoin de la plupart des informations sur le niveau : la map qui comprend les bonus, ennemis et bombes, les coordonnées actuelles du joueur pour démarrer le A*, le liste des bonus pour vérifier si la sortie est disponible (bonusl = NULL) et les infos sur le level.

- Les boucles servent à initialiser les matrices, la boucle principales est un TantQue qui tourne tant que la file n'est pas vide, cela permet d'assurer soit de vider la file, soit de trouver un bonus/la sortie avant de la vider.

C'est l'algorithme principal du runner déjà présentée dans les parties précédentes du compte-rendu. Avec le pseudo code on peut voir plus en détail comment l'algorithme a été implémenté non pas avec un graphe mais juste avec un suivi de la map avec des tableaux de tableaux (matrice).

Cet algorithme va donc parcourir la map depuis la position du joueur et tant qu'il n'y a as d'ennemi ou de sortie accessible, il va se contenter d'enfiler les cases selon la priorité de leur heuristique. Si un chemin vaut moins cher qu'un autre pour une même case, on gardera le chemin le moins coûteux.
L'algorithme a été découpé en sous fonction/procédure pour une meilleure compréhension du code mais la fonction aStarAlgorithm reste très longue au vu des nombreuses matrices à initialiser et des différents cas à gérer.

Pseudo-code : A* Algorithm

```
FONCTION aStarAlgorithm en action
Declarations :
    Parametres :
        map : tableau de tableau de caracteres
        x : entier
        y : entier
```

```
        level : levelinfo
        bonusl : liste chainee de bonus
    Variables :
        openFile : pointeur vers une file
        closedFile : pointeur vers une file
        tab_g : pointeur sur un pointeur d'un entier
        tab_f : pointeur sur un pointeur d'un entier
        heuristic : pointeur sur un pointeur d'un entier
        ongoing : coord
        list_parents : pointeur sur un pointeur d'un coord
        i : entier
        j : entier

DEBUT

    heuristic <- allouer(taille(entier) * level.ysize)
    calculHeurheuristicCalculationistic // Alloue et remplit
    heuristic

    openFile <- creerFile()
    closedFile <- creerFile()

    tab_g <- allouer(taille(entier) * level.ysize) //initialistion
    des matrices g et f pour le A*
    tab_f <- allouer(taille(entier) * level.ysize)
    Pour i allant de 0 a level.ysize - 1 Faire
        ?(tab_g + i) <- allouer(level.xsize * taille(entier))
        ?(tab_f + i) <- allouer(level.xsize * taille(entier))
    FinPour

    Pour i allant de 0 a level.ysize - 1 Faire //Remplit f et g par
     des 0
        Pour j allant de 0 a level.xsize - 1 Faire
            ?((tab_g + i)?(j)) <- 0
            ?((tab_f + i)?(j)) <- 0
        FinPour
    FinPour

    ongoing <- (x, y)
    enfile(ongoing dans openFile) //Met le sommet initial dans la
    file

    list_parents <- allouer(level.xsize * taille(coord)) //
    initialise la matrice qui retrace les parents lors des visites
    Pour i allant de 0 a level.ysize - 1 Faire
        ?(list_parents + i) <- allouer(level.xsize * taille(entier)
    )
        Pour j allant de 0 a level.xsize - 1 Faire
            ?((list_parents + i)?(j)) <- (-1, -1) //on le remplit
    par une valeur par defaut
        FinPour
    FinPour

    ?((list_parents + y)?(x)) <- (x, y) //marque la case initiale
    comme parent d'elle meme

    Si isEnemyNear(map, x, y, level) Alors //Action si un ennemi
```

```
    est proche
56      libere(openFile, closedFile, tab_g, tab_f, heuristic,
    list_parents)
57      retourner perduPourPerdu(map, x, y, level)
58  FinSi
59
60  Tant que non estVide(openFile) Faire //Tant que la file n'est
    pas vide
61      ongoing <- defile(openFile)
62      Si (ongoing = (level.xexit, level.yexit) ET bonusl != NULL)
     Alors // Si la sortie est accessible et trouvee, y aller
63          libere(openFile, closedFile, tab_g, tab_f, heuristic)
64          retourner retracePathForAStar(x, y, ongoing,
    list_parents, level)
65      FinSi
66      Si (map[y][x] = BONUS) Alors //Si le bonus est trouve, y
    aller
67          libere(openFile, closedFile, tab_g, tab_f, heuristic)
68          retourner retracePathForAStar(x, y, ongoing,
    list_parents, level)
69      FinSi
70      Sinon
71          enfile(ongoing dans closedFile)
72          testNeighbors(map, ongoing, tab_g, tab_f, list_parents,
     openFile, closedFile, heuristic) //teste les voisins
    accessibles et on les enfile
73      FinSinon
74  FinTantQue
75
76  libdre(openFile, closedFile, tab_g, tab_f, heuristic,
    list_parents)
77  retourner perduPourPerdu(map, x, y, level) //si rien n'est
    possible
78
79  FIN
```

Listing 1 – A* Algorithm Function

Parcours en largeur (BFS) :

- C'est une procédure car elle ne renvoie rien et ne fait qu'agir sur heuristic grâce aux pointeurs.

- En entrée on a besoin de la plupart des informations sur le niveau : la map qui comprend les bonus, ennemis et bombes, les coordonnées d'où on démarre le BFS pour calculer les heuristiques, et les infos sur le level.

- Les boucles servent à initialiser les matrices, la boucle principales est un TantQue qui tourne tant que la file n'est pas vide, cela permet d'assurer soit de vider la file, soit de trouver un bonus avant de la vider.

C'est l'algorithme qui permet de calculer les heuristique de distance pour le tableau d'heuristique utile au A*. Il permet de calculer à coup sûr le chemin le plus court en terme de nombre de "pas" vers un bonus.
Pour chaque voisins de la case visitée, on enfile les accessibles, ce qui permet de chercher sur la map les bonus de tel sorte que toutes les cases à 1 pas sont visitées, puis celles à 2 pas, 3 pas etc...

Pseudo-code : Breadth-First Search (BFS)

```
PROCEDURE BFS
    Declarations :
        Parametres :
            map : tableau de tableau de caracteres
            x_start : entier
            y_start : entier
            level : levelinfo
            heuristic : pointeur sur un pointeur sur un entier
        Variables :
            f : pionteur vers une file
            ongoing : coord
            list_parents : pointeur sur un pointeur d'un coord

DEBUT

    list_parents <- allouer(level.xsize * taille(coord)) //
    initialise le tableau qui trace les parents et le remplit avec
    une valeur par defaut
    Pour i allant de 0 a level.ysize - 1 Faire
        ?(list_parents + i) <- allouer(level.xsize * taille(entier)
    )
        Pour j allant de 0 a level.xsize - 1 Faire
            ?((list_parents + i)?(j)) <- (-1, -1)
        FinPour
    FinPour

    f <- creerFile()
```

```
25
26      ongoing <- (x_start, y_start) //coordonnee en cours
27      enfile(ongoing dans f)
28      ?((list_parents + y_start)?(x_start)) <- ongoing //marque la
        case de depart comme parent d'elle meme
29
30      Tant que non estVide(f) Faire //tant que la file n'est pas vide
31          ongoing <- defile(f)
32          Si (map[y][x] = BONUS) Alors //si la case visitee est un
        bonus
33              retraceRouteForBFS(list_parents, x_start, y_start) //
        Retrouve le parent initial
34              break
35          FinSi
36          checkPaths(map, ongoing, f, list_parents) //verifie si les
        voisins sont accessibles
37      FinTantQue
38
39      libere(f, list_parents)
40
41  FIN
```

Listing 2 – BFS Algorithm

Test et traitement des voisins :

- Ce sont des procédures car elle ne renvoie rien et ne font qu'agir sur les matrices g, f et sur les files grâce aux pointeurs.

- En entrée on a besoin de la map qui comprend les bonus, ennemis et bombes, les coordonnées en cours, mais aussi de g et f créés dans A*, des files ouvertes et fermées, de la matrice des parents et de la matrice d'heuristique. processNeighbors a aussi besoin des coordonées du voisin à traiter.

- Il n'y a pas de boucle, ces procédures n'ont besoin que des conditions if pour gérer les différents cas (soit tester si un voisin est accessible, soit vérifier si une case est déjà dans une file, etc...).

Ce sont les procédures qui ont été créer pour soulager la fonction A* et "sous-traiter" les cases adjacentes à celle visitée. En effet, ici on teste si les 4 cases adjacentes sont accessibles ou non, si oui, la case est traitée : on vérifie si elle n'est pas déjà dans une des files avec un meilleur coût, si ce n'est pas le cas, on marque les parents (traitement de la case) et on ajoute le nouveau coût puis on la retire des files où elle était possiblement. On finit la l'enfiler de manière prioritaire dans openFile.

Pseudo-code : TestNeighbors et ProcessNeighbor

```
1  PROCEDURE testNeighbors
2      Declarations :
3          Parametres :
4              map : tableau de tableau de caracteres
5              ongoing : coord
6              tab_g : pointeur sur un pointeur d'un entier
7              tab_f : pointeur sur un pointeur d'un entier
8              list_parents : pointeur sur un pointeur d'un coord
9              openFile : pointeur vers une file
10             closedFile : pointeur vers une file
11             heuristic : tableau d'entiers
12
13 DEBUT
14
15     Si checkRight(ongoing, map) ET list_parents[y][x + 1].x == -1
       Alors //verifie si la case de droite est accessible et pas deja
        visitee
16         processNeighbor(ongoing, x + 1, y, g, f, list_parents,
       openFile, closedFile, heuristic) //si c'est ok, on la traite
17     FinSi
18     //on fait pareil dans toutes les directions
19     Si checkLeft(ongoing, map) ET list_parents[y][x - 1].x == -1
       Alors
20         processNeighbor(ongoing, x - 1, y, g, f, list_parents,
       openFile, closedFile, heuristic)
```

```
    FinSi
    Si checkDown(ongoing, map) ET list_parents[y + 1][x].x == -1
    Alors
        processNeighbor(ongoing, x, y + 1, g, f, list_parents,
    openFile, closedFile, heuristic)
    FinSi
    Si checkUp(ongoing, map) ET list_parents[y - 1][x].x == -1
    Alors
        processNeighbor(ongoing, x, y - 1, g, f, list_parents,
    openFile, closedFile, heuristic)
    FinSi

FIN
```

Listing 3 – TestNeighbors Procedure

```
PROCEDURE processNeighbor
    Declarations :
        Parametres :
            ongoing : coord
            x_neighbor : entier
            y_neighbor : entier
            tab_g : tableau de tableau d'entiers
            tab_f : tableau de tableau d'entiers
            list_parents : tableau de tableau de coord
            openFile : pointeur vers une file
            closedFile : pointeur vers une file
            heuristic : pointeur sur un pointeur d'un entier
        Variables :
            g_tmp : entier
            h : entier
            f_tmp : entier
            f_cost_open : entier
            g_cost_open : entier

DEBUT

    g_tmp <- ?((tab_g + y)?(x)) + 1 //varibale pour le cout g
    temporaire
    h <- ?((heuristic + y_neighbor)?(x_neighbor)) //valeur de l'
    heuristique pour cette case
    f_tmp <- g_tmp + h //f temporaire
    f_cost_open <- dansFile(openFile, x_neighbor, y_neighbor) //
    recupere le cout de la case si elle est dans openFile, -1 sinon
    f_cost_closed <- dansFile(openFile, x_neighbor, y_neighbor) //
    de meme avec closedFile

    Si (f_cost_open != -1 ET f_tmp > f_cost_open) OU (f_cost_closed
     != -1 && f_tmp > f_cost_closed) Alors //si dans openFile ou
    closedFile mais temporaire a un meilleur cout
        // Fin de la procedure
    FinSi

    list_parents[y_neighbor][x_neighbor] <- ongoing //marque le
    parent
    g[y_neighbor][x_neighbor] <- g_tmp //marque le nouveau cout
    f[y_neighbor][x_neighbor] <- f_tmp

```

```
36      Si f_cost_open != -1 Alors //retire de openFile si elle y etait
         deja
37          retirerFile(closedFile, x_neighbor, y_neighbor)
38      FinSi
39
40      Si f_cost_closed != -1 Alors //de meme pour closedFile
41          retirerFile(closedFile, x_neighbor, y_neighbor)
42      FinSi
43
44      enfilePrioritaire(openFile, {x = x_neighbor, y = y_neighbor,
        cost = f_tmp})//enfile prioritairement dans openFile
45
46  FIN
```

Listing 4 – ProcessNeighbor Procedure

5 Limites et comment aller plus loin

La stratégie permet de rechercher le plus proche bonus tout en priorisant un chemin loin des ennemis grâce à une heuristique pour chaque case.

- Tout d'abord, la gestion d'attaque des ennemis est assez limitée, on ne l'utilise qu'en cas de danger direct. Il pourrait être judicieux d'utiliser les bombes si un ennemi est sur une ligne droite que l'on souhaite parcourir ou s'il fait obstacle direct au joueur. De même, la fuite par un trou de bombe n'est pas vraiment exploité. Finalement, la gestion d'urgence est assez limité et il faudrait rajouter plus de vérification pour être sûr de la sécurité du runner mais cela devient vite lourd au niveau de l'écriture du code.

- Les principales sources de défaites sont : lorsque le joueur saute d'un câble ou d'une plateforme et que l'endroit où il tombe est immédiatement collé à un ennemi donc se fait attraper directement. J'ai tenté de gérer cela en prédisant l'endroit de la chute mais la vérification n'est valable que lorsque le joueur tombe sur un câble. Il y a aussi quand le joueur est vraiment coincé sur des échelles entre deux ennemis, il faudrait une stratégie d'évasion vraiment fine ici pour s'en sortir.

- Faire un BFS sur chaque case non visitée pour le calcul des heuristiques n'était pas vraiment optimisé, on aurait pu plutôt utiliser une distance de Manhattan, moins lourde en calculs.

- Il pourrait aussi être intéressant de gérer le fait que les ennemis se dirige vers le joueur en utilisant le chemin le plus cours, donc les déplacements deviennent prévisible. De plus, leur vitesse est deux fois plus faible que celle du joueur donc on pourrait imposer la condition que chaque action amène à une case qui a une distance d'au moins 2 cases à l'ennemi le plus proche.

- De plus, une approche intéressante peut être ma première idée qui est de remplacer le A* par seulement un Parcours en Largeur mais elle a été testée et est moins efficace.

- Enfin, une approche par l'utilisation de probabilités est aussi envisageable, en calculant la probabilité de quelle case vaut plus le coup d'y accéder ou non par rapport à une estimation de la prochaine position des ennemis par exemple.