ENSSAT

LANNION

Compte rendu de projet : Algorithme 1

Lode Runner

Julien BOURDET

ENSSAT

1ère année - Informatique

Lannion, December 2024

Table des matières

1

Introduction

Dans le cadre du module Algorithme 1, nous avons pu appliquer les concepts étudiés en cours en réalisant un projet. Ce projet consiste à réaliser une IA capable de jouer au jeu Lode Runner. C'est un jeu d'arcade qui se déroule sur une carte en deux dimensions où le joueur doit récupérer des bonus tout en évitant des ennemis.

L'objectif du projet est de développer une IA capable de pouvoir terminer n'importe quel niveau du jeu. Cependant, des limitations ont été imposées : l'IA ne connaît pas les actions futures des ennemis et ne peut pas mémoriser des informations d'un tour à l'autre. Ce cadre impose une stratégie approfondie sur la position actuelle afin de ne pas faire un mouvement qui serait mortel plus tard.

Au cours du développement, j'ai rencontré plusieurs défis. L'absence de mémorisation empêche d'utiliser une stratégie qui prend trop de temps de calcul ou de pouvoir créer une stratégie sur plusieurs tours, tandis que ne pas connaître les prochains coups des ennemis empêche d'explorer un "arbre des positions". Par ailleurs, l'implémentation en C, avec ses contraintes de gestion manuelle de la mémoire, a ajouté une dimension technique non négligeable.

Malgré ces contraintes, j'ai pu élaborer une IA fonctionnelle et performante (au moins sur les niveaux disponibles).

Ce compte rendu présente ma démarche de développement, la stratégie utilisée et les résultats obtenus.

2

Préliminaires

Avant de commencer à détailler la stratégie utilisée, il est nécessaire de présenter certaines notions sur lesquelles elle repose.

2.1 Tas-Min

Un Tas-min est une structure de données qui permet de stocker un ensemble d'éléments et de les récupérer dans un ordre particulier. C'est un cas particulier d'une file de priorité, où l'élément avec la plus petite priorité est toujours en tête de file.

Notre Tas-min est implémenté sous forme d'un tableau d'entiers, où chaque élément est un nœud de l'arbre binaire représentant le tas. Les indices des éléments sont choisis de manière à ce que le fils gauche de l'élément à l'indice i soit à l'indice $2i + 1$ et le fils droit à l'indice $2i + 2$. La racine de l'arbre est à l'indice 0.

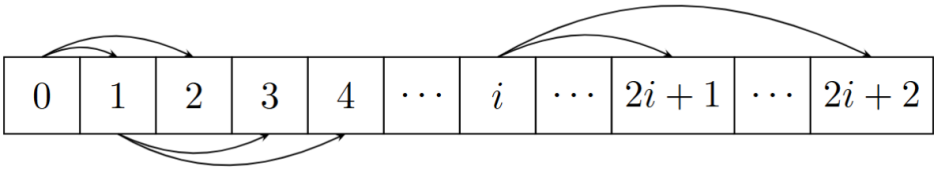

Figure 2.1: *Représentation d'un Tas-min sous forme de tableau.*

Pour gérer les Tas-min, nous avons besoin de trois procédures : `insert`, `modify_priority` et `extract_min`. La première permet d'ajouter un élément au tas, la deuxième de modifier la priorité d'un élément et la troisième de récupérer l'élément de priorité minimale.
On sait que l'élément de priorité minimale est toujours à la racine de l'arbre, on peut donc le récupérer en temps constant. En cas d'insertion, de suppression ou de modification de priorité, on doit s'assurer que le tas reste bien un Tas-min. Pour cela, on utilise une des deux procédures suivantes : `percolate_up` et `percolate_down`.

2.1.1 Procédure `insert`

On ajoute l'élément à la fin du tableau et on appelle `percolate_up` pour s'assurer que le tas reste un Tas-min. La procédure `percolate_up` remonte l'élément ajouté tant que son parent a une priorité plus grande que la sienne.

Son pseudo code est le suivant :

```
1   Procédure percolate_up
2   Parametres :
3       @heap en tas-min
4       i en entier
5   Debut
6       Tant que heap.priority[i] < heap.priority[(i - 1) / 2] et i > 0
7           Echanger(heap.array, i, (i - 1) / 2)
8           Echanger(heap->priority, i, (i - 1) / 2)
9           i <- (i - 1) / 2
10      Fin Tant que
11  Fin
```

Listing 2.1: *Procédure `percolate_up`.*

2.1.2 Procédure `modify_priority`

On modifie la priorité de l'élément et on appelle `percolate_up` pour s'assurer que le tas reste un Tas-min.

2.1.3 Procédure `extract_min`

On récupère l'élément de priorité minimale, on le remplace par le dernier élément du tableau et on appelle `percolate_down`. La procédure `percolate_down` descend l'élément remplacé jusqu'à ce que ses fils aient une priorité plus grande que la sienne.

Son pseudo code est le suivant :

```
1   Procédure percolate_down
2   Parametres :
3       @heap en tas-min
4       i en entier
5   Declarations :
6       current, left, right en entier
7   Debut
8       current <- i
9       left <- 2 * i + 1
10      right <- 2 * i + 2
11
12      Si left < heap.size et heap.priority[left] < heap.priority[current]
13          current <- left
14
15      Si right < heap.size et heap.priority[right] < heap.priority[current]
16          current <- right
17
18      Si current != i
19          Echanger(heap.array, i, current)
20          Echanger(heap.priority, i, current)
21          percolate_down(heap, current)
22  Fin
```

Listing 2.2: *Procédure `percolate_down`.*

2.1.4 En C

En C, on utilise une structure `min_heap` pour représenter le Tas-min.

```
1       typedef struct min_heap{
2           int size; // Nombre d'éléments dans le tas
3           int capacity; // Taille max du tas
4           int* array; // Tableau des éléments
5           float* priority; // Prioritée des éléments
6         } min_heap;
```

Listing 2.3: *Structure `min_heap` en C.*

On dispose en plus des fonctions suivantes pour manipuler les Tas-min :

```c
min_heap* create_min_heap(int);
void free_min_heap(min_heap*);
void percolate_up(min_heap*, int);
void percolate_down(min_heap*, int);
void insert(min_heap*, int, float);
void modify_priority(min_heap*, int, float);
int extract_min(min_heap*);
bool is_member(min_heap*, int);
```

Listing 2.4: *Fonctions sur les Tas-min en C.*

2.2 Algorithme A*

L'algorithme A* est un algorithme de recherche de chemin dans un graphe pondéré. Il est basé sur l'algorithme de Dijkstra, mais utilise une heuristique pour guider la recherche. L'algorithme A* est utilisé pour trouver le chemin le plus court entre un nœud de départ et un nœud d'arrivée dans un graphe.
Il utilise une file de priorité pour stocker les nœuds à explorer, où la priorité d'un nœud est la somme du coût du chemin parcouru pour atteindre ce nœud et de l'estimation du coût restant pour atteindre le nœud d'arrivée (l'heuristique). On utilisera ici un Tas-min (Section 2.1) pour implémenter cette file de priorité.

2.2.1 Pseudo code

```
Fonction a_star en @path
Parametres :
    origin en entier
    destination en entier
    level en niveau
Déclarations :
    pat en @path
    u, v en entier
    h_v en flottant
Début
    pat <- create_path // On initialise le chemin
    // pat.d est le tableau des distances, pat.p est le tableau des parents
    pat.d[origin] <- origin
    insert(pat.heap, origin, 0) // On ajoute le point d'origine a la file

    Tant que pat.heap n'est pas vide
        // Sommet courant, celui avec la priorite minimale
        u <- extract_min(pat.heap)

        Si u = destination
            // On a trouve le bonus, on pourra remonter le chemin grace a pat.p
```

```
                pat.found <- VRAI
                Sortir de la boucle
            Fin Si

            Pour chaque action possible
                Si l'action est valide // Dépend de level
                    v <- position apres l'action
                    h_v <- distance entre v et le bonus (heuristique)
                    Si pat.d[u] + poids de l'action < pat.d[v]
                        // Si on a trouve un chemin plus court
                        pat.d[v] <- pat.d[u] + poids de l'action
                        pat.p[v] <- u
                        Si v n'est pas dans la file
                            insert(pat.heap, v, pat.d[v] + h_v) // On l'ajoute
                        Sinon
                            // On modifie sa priorite
                            modify_priority(pat.heap, v, pat.d[v] + h_v)
                        Fin Si
                    Fin Si
                Fin Si
            Fin Pour
        Fin Tant que

        Retourner pat
    Fin
```

Listing 2.5: *Fonction `a_star`.*

2.2.2 En C

En C, la fonction renvoie un chemin, qui est une structure contenant les informations nécessaires pour retrouver le chemin trouvé.

On conserve le tableau des distances `d`, le tableau des parents `p`, un booléen `found` indiquant si le chemin a été trouvé et le Tas-min `heap`. On garde le Tas-min pour les cas où il n'est pas vide (on n'a pas trouvé de chemin) et pour libérer la mémoire.

```c
typedef struct path{
    min_heap* heap; // File de priorité
    int* p; // Tableau des parents
    int* d; // Tableau des distances
    bool found; // Chemin trouvé ou non
    } path;
```

Listing 2.6: *Structure `path` en C.*

3

Stratégie

La stratégie utilisée pour terminer le jeu est construite autour de l'algorithme A* (Section 2.2).
On commence par récupérer la liste des bonus par ordre croissant de distance et on essaye, en utilisant A*, de trouver un chemin pour atteindre le bonus le plus proche.
Si on trouve un chemin vers un des bonus, le runner le prendra : A* a une priorité sur toutes les autres décisions.
Si l'algorithme ne trouve aucun chemin vers aucun bonus (au plus il y a d'ennemis, au plus ça arrive), le runner utilise des fonctions de mouvements dites "spéciales" pour se déplacer.

Ceci veut dire que A* doit prendre en compte les ennemis, mais il y a plusieurs avantages à utiliser cette stratégie :

1. On fait une confiance totale à A* : s'il sait qu'un ennemi n'est pas dangereux, il n'y a pas besoin de faire de mouvements pour l'éviter.
2. On ne fait pas de mouvements inutiles : si A* trouve un chemin, on peut prouver que c'est le chemin le plus court.
3. Si A* ne trouve pas de chemin, on peut être sûr qu'il n'y en a pas, et on peut donc se permettre de faire des mouvements "spéciaux".

Néanmoins, il y a un inconvénient : on ne peut pas prédire l'évolution de la position des ennemis, les choix de A* se basent sur la situation à l'instant t. C'est pour cela qu'on l'appelle à chaque tour, pour prendre en compte les nouvelles positions des ennemis.

Heureusement, l'imprécision de la position des ennemis augmente avec la distance, et leur dangerosité diminue. L'algorithme peut être à peu près sûr de la position des ennemis proches (les ennemis dangereux), et donc de les éviter.

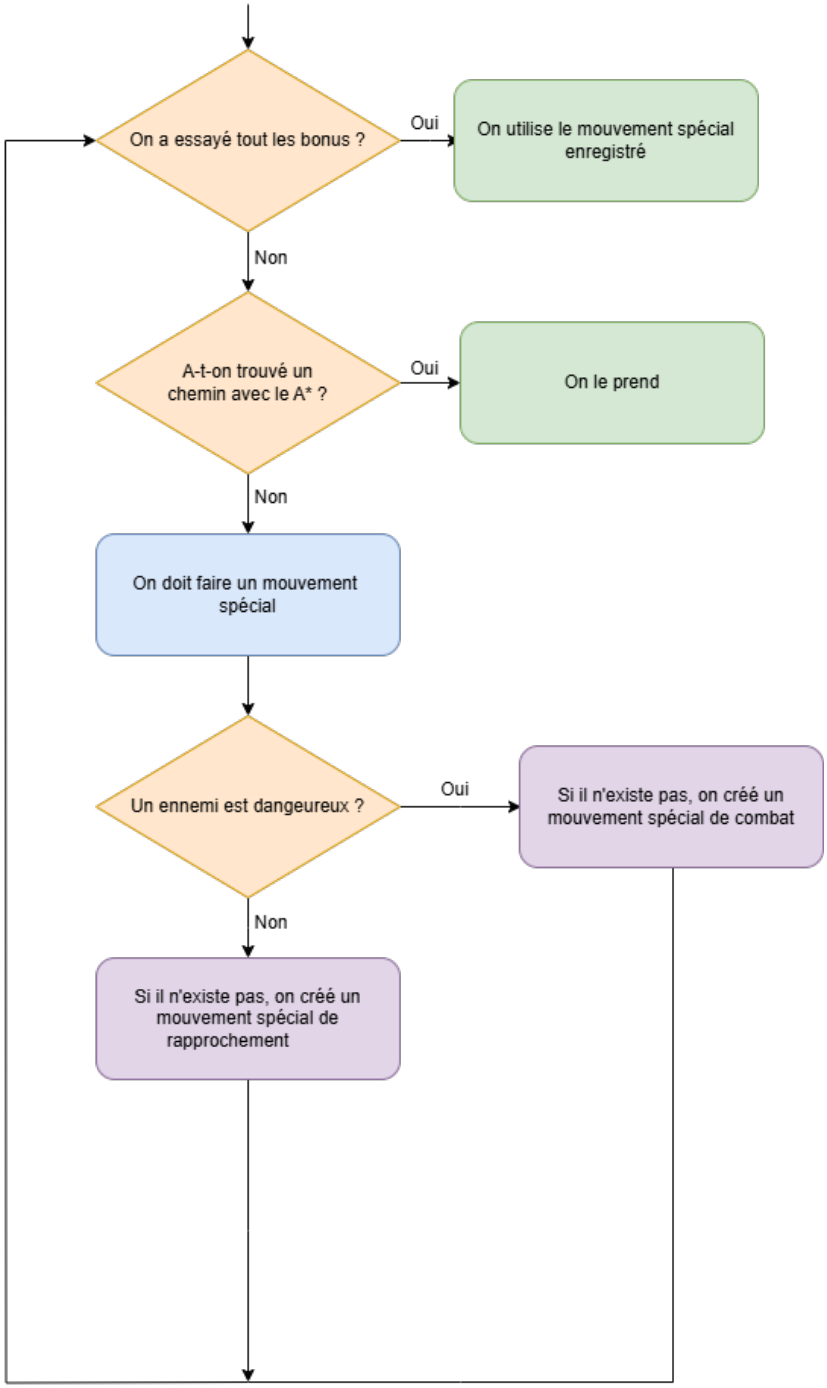

Figure 3.1: *Logigramme de la stratégie.*

3.1 Mouvemements spéciaux

Les mouvements spéciaux sont les mouvements effectués quand A* ne trouve aucun chemin, ils tentent, sans garantie de succès, d'améliorer la situation, pour que A* puisse trouver un chemin dans un prochain tour.
Ils sont, contrairement à A*, beaucoup plus spécifiques.

3.1.1 Mouvement de combat

Comme vu dans le logigramme (Figure 3.1), un mouvement de combat est enregistré si A* ne trouve aucun chemin, qu'un ennemi est considéré comme dangereux (**??**) et qu'il n'existe pas déjà un mouvement de combat. Alors, on va utiliser une suite de conditions (Section 6.4) pour déterminer l'action à réaliser (se déplacer, poser une bombe, attendre).

3.1.2 Mouvement de rapprochement

Si A* ne trouve aucun chemin et qu'aucun ennemi n'est considéré comme dangereux, c'est qu'un ou plusieurs ennemis bloquent le chemin. On va alors essayer de se rapprocher du bonus, jusqu'à ce que :

- A* trouve un chemin.
- On se rapproche assez d'un ennemi pour qu'il soit considéré comme dangereux.

3.1.3 Remarques

Il faut faire attention à bien conserver le premier mouvement de combat (et de rapprochement), car on va itérer sur tous les bonus, mais on veut se rapprocher du plus proche (le premier).

4

Paramètres et Valeur de retour

Afin de permettre à notre code d'intéragir avec le moteur de jeu, il faut utiliser une fonction :

```
1        action lode_runner(levelinfo, character_list, bonus_list, bomb_list);
```

Listing 4.1: *Prototype de `lode_runner` en C.*

4.1 Paramètres

Paramètre	Type	Description
`level`	`levelinfo`	Structure contenant les informations du niveau.
`characterl`	`character_list`	Liste chaînée contenant le runner et les ennemis.
`bonusl`	`bonus_list`	Liste chaînée contenant les bonus.
`bombl`	`bomb_list`	Liste chaînée contenant les bombes.

Table 4.1: *Paramètres de la fonction `lode_runner`.*

4.2 Valeur de retour

Cette fonction est appellée à chaque tour du jeu. Elle retourne une action à effectuer par le runner. Cette action est un élément de l'énumération suivante :

Entier	Valeur	Description
0	`NONE`	Ne rien faire.
1	`UP`	Aller en haut.
2	`DOWN`	Aller en bas.
3	`LEFT`	Aller à gauche.
4	`RIGHT`	Aller à droite.
5	`BOMB_LEFT`	Poser une bombe à gauche.
6	`BOMB_RIGHT`	Poser une bombe à droite.

Table 4.2: *Énumération des actions possibles.*

4.3 Struture `levelinfo`

Parmi les paramètres de la fonction `lode_runner`, il y a la structure `levelinfo`. Cette structure contient les informations du niveau actuel, sous la forme suivante :

```
1  typedef struct{
2      char **map;
3      int xsize;
4      int ysize;
5      int xexit;
6      int yexit;
7  } levelinfo;
```

Listing 4.2: *Structure `levelinfo` en C.*

Le tableau `map` contient les éléments de la carte, sous la forme d'un tableau de caractères, un exemple est donné sur la Figure 4.1.

	0	1	2	3	4	5	6	7	8	9
0	W	W	W	W	W	W	W	W	W	W
1	W				X					W
2	W				L		B			W
3	W				L	F	F	F	L	W
4	W		E		L				L	W
5	W	L	F	F	F				L	W
6	W	L							L	W
7	W	L		B			R		L	W
8	W	F	F	F	F	F	F	F	F	W
9	W	W	W	W	W	W	W	W	W	W

Figure 4.1: *Exemple de carte.*

On peut remarquer que les `x` et les `y` sont inversés par rapport à une matrice classique, c'est à dire que l'axe des ordonnées est l'axe des abscisses et vice-versa. De plus, les coordonnées sont relatives à la carte, c'est à dire que le coin supérieur gauche est en (0, 0) et le coin inférieur droit est en (`xsize`, `ysize`). Enfin, cela mène à ce que contrintuitivement, se déplacer vers le haut signifie diminuer les `y`.

Afin de faciliter le développement, et de laisser une liberté de choix dans l'implémentation, chaque caractère de la carte est défini par une constante, comme suit :

Caractère	**Constante**	**Description**
'H'	`BOMB`	Bombe.
'B'	`BONUS`	Bonus.
'C'	`CABLE`	Câble.
'E'	`ENEMY`	Ennemi.
'X'	`EXIT`	Sortie.
'F'	`FLOOR`	Sol.
'L'	`LADDER`	Échelle.
'.'	`PATH`	Chemin.
'R'	`RUNNER`	Runner.
'W'	`WALL`	Mur.

Table 4.3: *Énumération des caractères de la carte.*

5

Modules

La stratégie se décompose en plusieurs modules, chacun ayant un rôle bien défini. Ces modules sont indépendants les uns des autres et peuvent être utilisés de manière isolée.

5.1 Liste des modules

5.1.1 Tas-Min

Prototype	Description
`min_heap* create_min_heap`	Crée un tas-min.
`void free_min_heap`	Libère la mémoire allouée pour le tas-min.
`void swap`	Échange les valeurs des deux entiers.
`void swapf`	Échange les valeurs des deux flottants.
`void percolate_up`	Conserve l'invariant du tas-min.
`void percolate_down`	Conserve l'invariant du tas-min.
`void insert`	Insère un élément dans le tas-min.
`void modify_priority`	Modifie la priorité d'un élément du tas-min.
`int extract_min`	Extrait l'élément de priorité minimale du tas-min.
`bool is_member`	Vérifie si un élément est présent dans le tas-min.

Table 5.1: *Modules relatifs au tas-min.*

5.1.2 A*

Prototype	Description
`path* create_path`	Crée un chemin.
`void free_path`	Libère la mémoire allouée pour le chemin.
`bool is_valid`	Vérifie si une action est valide.
`int weight`	Valeur d'une action, utilisée pour le calcul de la priorité.
`int get_new_pos`	Renvoie la position après avoir effectué une action.
`action get_action`	Renvoie l'action à effectuer pour aller de u à v, si ce n'est pas possible, renvoie `NONE`.
`path* a_star`	Renvoie le chemin le plus court entre le runner et le bonus.
`child* find_closest_child`	Si A* ne trouve pas de chemin, on cherche le chemin qui nous rapproche le plus du bonus.
`action lode_runner`	Action du lode-runner.

Table 5.2: *Modules relatifs à l'algorithme A*.*

5.1.3 Outils

Prototype	Description
`float dist`	Calcule la distance euclidienne entre deux points (avec leurs coordonnées).
`float vdist`	Calcule la distance euclidienne entre deux points (avec leurs positions dans le niveau).
`character_list get_runner`	Renvoie le runner parmi les personnages.
`bool is_in_bonus_list`	Vérifie si un bonus est dans une liste de bonus, on utilise ses coordonnées.
`levelinfo add_enemies`	Ajoute les ennemis et les bombes à la carte.
`bonus_list get_closest_bonus`	Renvoie le bonus le plus proche du runner, en évitant ceux déjà vus.
`character_list get_closest_enemy`	Renvoie l'ennemi dangereux le plus proche du runner.
`bool is_valid_closest`	Vérifie si une action est dangereuse, utilisée pour le mode `closest`.
`void combat_moves`	Gère les mouvements de combat.

Table 5.3: *Modules relatifs aux outils.*

6

Présentation des modules

6.1 Module `is_valid`

```
1        bool is_valid(int pos, action a, levelinfo level, levelinfo air_level);
```

Listing 6.1: *Prototype de `is_valid` en C.*

6.1.1 Description

Dans l'algorithme A*, on a besoin de récupérer touts les voisins d'une case donnée. On appelle voisin d'une case, une case atteignable en un seul mouvement possible.

La fonction `is_valid` prend donc en paramètre une action, et retourne un booléen indiquant si cette action est valide.

6.1.2 Paramètres

Paramètre	Type	Description
`pos`	`int`	Position de la case à tester.
`a`	`action`	Action à tester.
`level`	`levelinfo`	Structure contenant les informations du niveau.
`air_level`	`levelinfo`	Structure contenant les informations du niveau sans les ennemis.

Table 6.1: *Paramètres de la fonction `is_valid`.*

6.1.3 Choix d'implementation

On choisit d'utiliser un switch sur l'action à tester puisque chaque action a des conditions différentes.

6.1.4 Pseudo-code

```
1   Fonction is_valid en booléen
2   Parametres :
3       pos en entier
4       a en action
5       level en levelinfo
6       air_level en levelinfo
7   Declarations :
8       x en entier
9       y en entier
10      map en tableau de caractères
11      air_map en tableau de caractères
12      not_in_air en booléen
13  Debut
14      x <- pos % level.xsize // On récupère la coordonnée x de la case
15      y <- pos / level.xsize // On récupère la coordonnée y de la case
16      map <- level.map // On récupère la carte du niveau
17      air_map <- air_level.map // On récupère la carte du niveau sans les ennemis
18
19      // Variable indiquant si le joueur n'est pas en l'air
20      not_in_air <- (air_map[y + 1][x] != PATH && air_map[y + 1][x] != CABLE &&
        ↪  air_map[y + 1][x] != BOMB) || air_map[y - 1][x] == CABLE
21      // Si la case en dessous du joueur n'est pas un chemin, un cable ou une
        ↪  bombe, ou si la case au dessus du joueur est un cable, alors le joueur
        ↪  n'est pas en l'air
22
23      Selon a faire
24          Cas NONE :
25              Retourner VRAI
26          Cas UP :
27              // On ne peut monter que si on est sur une echelle et qu'il n'y a
                ↪  pas de mur au dessus
28              Si map[y][x] == LADDER et map[y - 1][x] != WALL et map[y - 1][x] !=
                ↪  FLOOR et map[y - 1][x] != ENEMY alors
29                  Retourner VRAI
30          Cas DOWN :
31              // On ne peut descendre que si il y a une echelle, un chemin ou un
                ↪  cable en dessous
32              // On laisse la possibilite de descendre si le joueur est en l'air
33              Si (map[y + 1][x] == LADDER ou map[y + 1][x] == PATH ou map[y +
                ↪  1][x] == CABLE) et map[y + 1][x] != ENEMY alors
34                  Retourner VRAI
35          Cas LEFT :
36              // On ne peut aller a gauche que si il n'y a pas de mur a gauche et
                ↪  que le joueur n'est pas en l'air
37              // C'est un petit hack car le moteur avance de plusieurs tour de
                ↪  jeu sans utiliser le code du joueur tant qu'il tombe, mais le
                ↪  A* ne le sait pas,
38              // alors on se debrouille pour que la seule action possible soit
                ↪  DOWN
```

```
39                  Si map[y][x - 1] != WALL et map[y][x - 1] != FLOOR et map[y][x - 1]
                    ↪  != ENEMY et map[y][x - 1] != DEAD et map[y + 1][x - 1] != ENEMY
                    ↪  et not_in_air alors
40                      Retourner VRAI
41              Cas RIGHT :
42                  // De meme pour la droite
43                  Si map[y][x + 1] != WALL et map[y][x + 1] != FLOOR et map[y][x + 1]
                    ↪  != ENEMY et map[y][x + 1] != DEAD et map[y + 1][x + 1] != ENEMY
                    ↪  et not_in_air alors
44                      Retourner VRAI
45              Defaut :
46                  Afficher "ERROR: Invalid action"
47                  Sortir du programme
48          Fin Selon
49
50          Retourner FAUX
51      Fin
```

Listing 6.2: *Pseudo-code de la fonction* `is_valid`.

6.2 Module get_closest_bonus

```
1       bonus_list get_closest_bonus(bonus_list bonusl, character_list runner,
        ↪  bonus_list already_seen);
```

Listing 6.3: *Prototype de* get_closest_bonus *en C.*

6.2.1 Description

Pour A*, on a besoin de destinations : les bonus. Mais, afin de finir le niveau plus rapidement (et pour éviter des cas de boucles infinies), on veut aller sur le bonus le plus proche du runner.
De plus, si le bonus le plus proche est inaccessible pour des raisons quelconques, on veut aller sur le second bonus le plus proche, et ainsi de suite. C'est pourquoi on a besoin de la liste des bonus déjà vus, pour ne pas les revoir.

6.2.2 Paramètres

Paramètre	Type	Description
bonusl	bonus_list	Liste des bonus.
runner	character_list	Liste des personnages.
already_seen	bonus_list	Liste des bonus déjà vus.

Table 6.2: *Paramètres de la fonction* get_closest_bonus.

6.2.3 Choix d'implementation

Les listes de bonus sont des listes chaînées, on choisit donc une boucle tant que pour parcourir la liste des bonus. De même, la fonction is_in_bonus_list fonctionne similairement, on ne la détaille donc pas.

6.2.4 Pseudo-code

```
1   Fonction get_closest_bonus en bonus_list
2   Parametres :
3       bonusl en bonus_list
4       runner en character_list
5       already_seen en bonus_list
6   Declarations :
7       closest_bonus en bonus_list
8       best_dist en reel
9       current en bonus_list
10  Debut
11      closest_bonus <- NULL
12
13      Si bonusl == NULL alors
14          // Si la liste des bonus est vide, il n'y a pas de bonus le plus proche
15          Retourner NULL
16      Fin Si
17
18      best_dist <- 100000 // On initialise la meilleure distance à une valeur
        ↪ très grande
19      current <- bonusl // On initialise le bonus courant à la tête de la liste
        ↪ des bonus
20
21      Tant que current != NULL faire
22          // On parcourt la liste des bonus
23          Si dist(current->b.x, current->b.y, runner->c.x, runner->c.y) <
            ↪ best_dist et non is_in_bonus_list(current, already_seen) alors
24              closest_bonus <- current
25              best_dist <- dist(current->b.x, current->b.y, runner->c.x,
                ↪ runner->c.y)
26          Fin Si
27          current <- current->next // On passe au bonus suivant
28      Fin Tant que
29
30      Retourner closest_bonus // On retourne le bonus le plus proche
31  Fin
```

Listing 6.4: *Pseudo-code de la fonction* `get_closest_bonus`.

6.3 Module add_ennemis

```
1       levelinfo add_ennemis(levelinfo level, character_list characterl, bomb_list
        ↪  bombl);
```

Listing 6.5: *Prototype de add_ennemis en C.*

6.3.1 Description

Pour que A* prenne en compte les ennemis, et ne tombe pas dans les trous causés par les bombes, on ajoute les ennemis et les bombes à la carte. C'est à dire qu'on créé une nouvelle carte, où les ennemis et les bombes sont des obstacles. De plus, si un ennemi est sur une bombe, on considère que l'ennemi est mort, (on peut marcher dessus).

6.3.2 Paramètres

Paramètre	Type	Description
level	levelinfo	Structure contenant les informations du niveau.
characterl	character_list	Liste des personnages.
bombl	bomb_list	Liste des bombes.

Table 6.3: *Paramètres de la fonction add_ennemis.*

6.3.3 Choix d'implementation

Les listes d'ennemis et de bombes sont des listes chaînées, on choisit donc une boucle tant que pour parcourir ces listes. Afin de s'assurer qu'on a les bons type de personnages, on utilise un Si / Sinon.

6.3.4 Pseudo-code

```
1       Fonction add_ennemis en levelinfo
2       Parametres :
3           level en levelinfo
4           characterl en character_list
5           bombl en bomb_list
6       Declarations :
7           currentb en bomb_list
8           current en character_list
9       Debut
10          currentb <- bombl // On initialise currentb à la tête de la liste des
            ↪  bombes
11
12          Tant que currentb != NULL faire
```

```
13              // On itère sur les bombes
14              level.map[currentb->y][currentb->x] <- BOMB
15              currentb <- currentb->next // On passe à la bombe suivante
16          Fin Tant que
17
18          current <- characterl // On initialise current à la tête de la liste des
            ↪  personnages
19
20          Tant que current != NULL faire
21              // On itère sur les personnages
22              Si current->c.item == ENEMY alors
23                  // Si le personnage est un ennemi (on pourrait tomber sur le
                    ↪  runner)
24                  Si level.map[current->c.y][current->c.x] == BOMB ou
                    ↪  level.map[current->c.y][current->c.x] == DEAD alors
25                      level.map[current->c.y][current->c.x] <- DEAD
26                  Sinon
27                      level.map[current->c.y][current->c.x] <- ENEMY
28                  Fin Si
29              Fin Si
30              current <- current->next // On passe au personnage suivant
31          Fin Tant que
32
33          Retourner level
34      Fin
```

Listing 6.6: *Pseudo-code de la fonction* `add_ennemis`.

6.4 Module combat_moves

```
1 void combat_moves(character_list runner, character_list closest_enemy, int*
  ↪ move_to_combat, levelinfo level);
```

Listing 6.7: *Prototype de combat_moves en C.*

6.4.1 Description

Si A* ne trouve aucun chemin pour aucun bonus, le mode de mouvement spécial est activé. Dans ce mode, le runner va essayer de se rapprocher du bonus le plus proche, tout en évitant les ennemis. Pour savoir comment se déplacer, si un ennemi est dangeureux, on utilise la procédure combat_moves.

6.4.2 Paramètres

Paramètre	Type	Description
runner	character_list	Runner
closest_enemy	character_list	Ennemi le plus proche.
move_to_combat	int*	Pointeur vers l'action à effectuer.
level	levelinfo	Structure contenant les informations du niveau.

Table 6.4: *Paramètres de la procédure combat_moves.*

6.4.3 Choix d'implementation

Cette procédure est assez complexe, surtout, elle n'a pas été implémentée en une seule fois, mais petit à petit. Cela vient du fait que les mouvements spéciaux sont assez complexes, et qu'il est difficile de tout prévoir dès le début, il faut tester et ajuster. J'ai tout de même essayé de faire en sorte que la procédure soit la plus lisible possible, en utilisant des commentaires et des variables explicites. On utilise donc des Si / Sinon pour tester les différentes conditions. De plus, on utilse une procédure car on modifie le pointeur move_to_combat en fonction de sa valeur actuelle.

6.4.4 Pseudo-code

```
1 Procéduer combat_moves
2 Parametres :
3     runner en character_list
4     closest_enemy en character_list
5     @move_to_combat en entier
6     level en levelinfo
7 Declarations :
```

```
8           down_left en caractère
9           down_right en caractère
10          top_left en caractère
11          top_right en caractère
12          left en caractère
13          right en caractère
14          center en caractère
15          can_right en booléen
16          can_left en booléen
17          distance en entier
18          can_up en booléen
19          can_down en booléen
20      Debut
21          // On récupère les cases autour du runner
22          down_left <- level.map[runner->c.y + 1][runner->c.x - 1]
23          down_right <- level.map[runner->c.y + 1][runner->c.x + 1]
24          top_left <- level.map[runner->c.y - 1][runner->c.x - 1]
25          top_right <- level.map[runner->c.y - 1][runner->c.x + 1]
26          left <- level.map[runner->c.y][runner->c.x - 1]
27          right <- level.map[runner->c.y][runner->c.x + 1]
28          center <- level.map[runner->c.y][runner->c.x]
29
30          // On vérifie si on peut se déplacer à droite ou à gauche, et si on ne
            ↪ tombe pas en le faisant
31          can_right <- is_valid(runner->c.y * level.xsize + runner->c.x, RIGHT,
            ↪ level, level)
32          can_left <- is_valid(runner->c.y * level.xsize + runner->c.x, LEFT, level,
            ↪ level)
33          can_right <- can_right et (down_right != BOMB et down_right != PATH)
34          can_left <- can_left et (down_left != BOMB et down_left != PATH)
35
36          Si closest_enemy != NULL alors
37              // Si il y a un ennemi dangeureux (donc que l'on est en mode combat)
38              Si *move_to_combat == -1 alors
39                  // Si on n'a pas encore décidé de comment se déplacer
40                  distance <- runner->c.y - closest_enemy->c.y // On calcule la
                    ↪ distance verticale
41                  Si distance == 0 alors
42                      // Combat horizontal
43                      distance <- runner->c.x - closest_enemy->c.x // On calcule la
                        ↪ distance horizontale
44                      Si level.map[runner->c.y - 1][runner->c.x] == CABLE et
                        ↪ level.map[runner->c.y + 1][runner->c.x] == PATH alors
45                          // On est sur un cable, on ne peut pas poser de bombe, on
                            ↪ saute
46                          *move_to_combat <- DOWN
47                      Sinon Si distance > 0 et distance < 4 alors
48                          // A gauche
49                          Si (down_left == FLOOR ou down_left == BOMB) et top_left !=
                            ↪ CABLE et left != ENEMY alors
50                              Si down_left == BOMB alors
```

```
51                                  // Il y a deja une bombe, on attend
52                                  *move_to_combat <- NONE
53                              Sinon
54                                  // On pose une bombe a gauche
55                                  *move_to_combat <- BOMB_LEFT
56                              Fin Si
57                          Sinon
58                              // On ne peut pas poser de bombe, on se deplace a
                                ↪ droite
59                              Si can_right alors *move_to_combat <- RIGHT
60                          Fin Si
61                      Sinon Si distance < 0 et distance > -4 alors
62                          // A droite
63                          Si (down_right == FLOOR ou down_right == BOMB) et top_right
                            ↪ != CABLE et right != ENEMY alors
64                              Si down_right == BOMB alors
65                                  // Il y a deja une bombe, on attend
66                                  *move_to_combat <- NONE
67                              Sinon
68                                  // On pose une bombe a droite
69                                  *move_to_combat <- BOMB_RIGHT
70                              Fin Si
71                          Sinon
72                              // On ne peut pas poser de bombe, on se deplace a
                                ↪ gauche
73                              Si can_left alors *move_to_combat <- LEFT
74                          Fin Si
75                      Fin Si
76
77                      // Ces mouvements ont priorite sur les autres
78
79                      Si can_right alors
80                          Si (left == LADDER ou center == LADDER ou (left == ENEMY et
                            ↪ top_left == LADDER)) et distance > 0 alors
81                              // On se deplace a droite si on est sur une echelle, ou
                                ↪ si on a une echelle a gauche
82                              *move_to_combat <- RIGHT
83                          Fin Si
84                      Fin Si
85
86                      Si can_left alors
87                          Si (right == LADDER ou center == LADDER ou (right == ENEMY
                            ↪ et top_right == LADDER)) et distance < 0 alors
88                              // On se deplace a gauche si on est sur une echelle, ou
                                ↪ si on a une echelle a droite
89                              *move_to_combat <- LEFT
90                          Fin Si
91                      Fin Si
92                  Sinon
93                      // Combat vertical (sur une echelle)
94                      bool can_up <- is_valid(runner->c.y * level.xsize +
                        ↪ runner->c.x, UP, level, level)
```

```
95                      bool can_down <- is_valid(runner->c.y * level.xsize +
                        ↪  runner->c.x, DOWN, level, level)
96                      Si distance > 0 et can_down alors
97                          // On descend si un ennemi est au dessus
98                          *move_to_combat <- DOWN
99                      Sinon Si distance < 0 et can_up alors
100                         // On monte si un ennemi est en dessous
101                         *move_to_combat <- UP
102                     Fin Si
103
104                     Si level.map[runner->c.y][runner->c.x] == PATH ou
                        ↪  level.map[runner->c.y + 1][runner->c.x] == FLOOR alors
105                         // Si on est en haut ou en bas d'une echelle, alors c'est
                            ↪  une erreur et on est en pas en mode combat, on remmet
                            ↪  move_to_combat a -1
106                         *move_to_combat <- -1
107                     Fin Si
108                 Fin Si
109             Fin Si
110         Fin Si
111     Fin
```

Listing 6.8: *Pseudo-code de la procédure* `combat_moves`.

6.5 Module lode_runner

```
1       action lode_runner(levelinfo level, character_list characterl, bonus_list
        ↪ bonusl, bomb_list bombl);
```

Listing 6.9: *Prototype de lode_runner en C.*

6.5.1 Description

La fonction lode_runner est la fonction principale de notre programme, c'est elle qui va appeler toutes les autres fonctions pour trouver le meilleur chemin pour le runner. Elle renvoie l'action à effectuer par le runner.

6.5.2 Paramètres

Paramètre	Type	Description
level	levelinfo	Structure contenant les informations du niveau.
characterl	character_list	Liste des personnages.
bonusl	bonus_list	Liste des bonus.
bombl	bomb_list	Liste des bombes.

Table 6.5: *Paramètres de la fonction lode_runner.*

6.5.3 Choix d'implementation

Le point principal de cette fonction est l'itération sur les bonus. Elle est donc centrée autour d'une boucle tant que (les bonnus sont des listes chaînées). On utilise des variables pour stocker les actions à effectuer, et on les retourne à la fin de la fonction.

6.5.4 Pseudo-code

```
1       Fonction lode_runner en action
2       Parametres :
3           level en levelinfo
4           characterl en character_list
5           bonusl en bonus_list
6           bombl en bomb_list
7       Declarations :
8           runner en character_list
9           astar_level en levelinfo
10          already_seen en bonus_list
11          closest_bonus en bonus_list
12          to_exit en booléen
13          move_to_combat en entier
```

```
14          move_to_closest en entier
15          move_to_path en entier
16          move_to_skipped en entier
17          @pat en path
18          @c en child
19          tmp en bonus_list
20          v en entier
21      Debut
22          runner <- get_runner(characterl) // On récupère le runner
23          level <- add_enemies(level, characterl, bombl) // On ajoute les ennemis à
            ↪ la carte
24          // On créé un niveau pour A* avec des zones autour des ennemis. Le but est
            ↪ que le runner ne s'approche pas trop des ennemis
25          astar_level <- get_astar_level(level, characterl)
26
27          already_seen <- NULL // On initialise la liste des bonus déjà vus
28          closest_bonus <- get_closest_bonus(bonusl, runner, already_seen) // On
            ↪ récupère le bonus le plus proche
29
30          Si bonusl == NULL alors
31              // Si la liste des bonus est vide, c'est qu'on les a tous récupérés, on
                ↪ va donc vers la sortie
32              // On crée un bonus fictif qui possède les coordonnées de la sortie
33              closest_bonus <- malloc(sizeof(bonus_list))
34              bonus b <- {level.xexit, level.yexit}
35              closest_bonus->b <- b
36              to_exit <- VRAI
37          Fin Si
38
39          // Initialisation des variables
40          // Ces variables vont stocker les actions à effectuer, elles sont
            ↪ initialisées à -1 pour qu'on sache si elles ont été modifiées
41          move_to_combat <- -1
42          move_to_closest <- -1
43          move_to_path <- -1
44          move_to_skipped <- -1
45
46          Tant que closest_bonus != NULL faire
47              // On itère sur les bonus
48              pat <- a_star(runner, closest_bonus, astar_level, level) // On calcule
                ↪ le chemin vers le bonus le plus proche
49              Si level.map[closest_bonus->b.y][closest_bonus->b.x] == ENEMY alors
50                  // Cas spécial : si un ennemi est sur le bonus, on ne peut pas y
                    ↪ aller (A* ne trouvera pas de chemin)
51                  int runner_pos <- runner->c.y * level.xsize + runner->c.x
52                  int closest_bonus_pos <- closest_bonus->b.y * level.xsize +
                    ↪ closest_bonus->b.x
53                  c <- find_closest_child(pat->p, runner_pos, closest_bonus_pos,
                    ↪ level) // On trouve le chemin qui nous rapproche le plus du
                    ↪ bonus
54                  move_to_skipped <- get_action(runner_pos, c->pos, level) // On
                    ↪ stocke l'action à effectuer
```

```
55                      // On ajoute le bonus à la liste des bonus déjà vus
56                      tmp <- malloc(sizeof(bonus_list))
57                      tmp->b <- closest_bonus->b
58                      tmp->next <- already_seen
59                      already_seen <- tmp
60                      Si non to_exit alors
61                          // Si il y a d'autres bonus, on récupère le plus proche pour
                            ↪  continuer la boucle
62                          closest_bonus <- get_closest_bonus(bonusl, runner,
                            ↪  already_seen)
63                      Fin Si
64                      Continuer
65                  Fin Si
66
67                  Si move_to_closest == -1 alors
68                      // Si on n'a toujours initialisé move_to_closest, on le fait
69                      // move_to_closest est l'action qui nous rapproche du bonus le plus
                        ↪  proche (même s'il est inaccessible)
70                      int runner_pos <- runner->c.y * level.xsize + runner->c.x
71                      int closest_bonus_pos <- closest_bonus->b.y * level.xsize +
                        ↪  closest_bonus->b
72
73                      c <- find_closest_child(pat->p, runner_pos, closest_bonus_pos,
                        ↪  level) // On trouve le chemin qui nous rapproche le plus du
                        ↪  bonus
74                      move_to_closest <- get_action(runner_pos, c->pos, level) // On
                        ↪  stocke l'action à effectuer
75                  Fin Si
76
77                  Si pat->found alors
78                      // Si on a trouvé un chemin, on le suit
79                      v <- closest_bonus->b.y * level.xsize + closest_bonus->b.x
80                      Tant que pat->p[v] != runner->c.y * level.xsize + runner->c.x faire
81                          // pat->p[v] est le parent de v, on remonte le chemin pour
                            ↪  trouver l'action à effectuer
82                          v <- pat->p[v]
83                      Fin Tant que
84
85                      move_to_path <- get_action(runner->c.y * level.xsize + runner->c.x,
                        ↪  v, level) // On stocke l'action à effectuer
86
87                  Sinon Si pat->heap->size != 0 alors
88                      // Si le chemin est plus long que la taille du tas, c'est une
                        ↪  erreur
89                      Afficher "ERROR: Path is longer than heap size"
90                      Sortir du programme
91                  Sinon
92                      // Si on n'a pas trouvé de chemin, on est en mode combat
93                      combat_moves(runner, get_closest_enemy(characterl, runner, level),
                        ↪  &move_to_combat, level)
94                  Fin Si
```

```
95
96              // On ajoute le bonus à la liste des bonus déjà vus
97              tmp <- malloc(sizeof(bonus_list))
98              tmp->b <- closest_bonus->b
99              tmp->next <- already_seen
100             already_seen <- tmp
101             Si non to_exit alors
102                 // Si il y a d'autres bonus, on récupère le plus proche pour
                    ↪  continuer la boucle
103                 closest_bonus <- get_closest_bonus(bonusl, runner, already_seen)
104             Sinon
105                 closest_bonus <- NULL
106             Fin Si
107         Fin Tant que
108
109         Si move_to_path != -1 alors
110             // On a trouvé un chemin, on le suit
111             Retourner move_to_path
112         Sinon Si move_to_combat != -1 alors
113             // On est en mode combat
114             Retourner move_to_combat
115         Sinon Si move_to_closest != -1 alors
116             // On n'a pas trouvé de chemin, on se rapproche du bonus
117             Si is_valid_closest(runner->c.y * level.xsize + runner->c.x,
                ↪  move_to_closest, astar_level) alors
118                 Retourner move_to_closest
119             Sinon
120                 Retourner NONE
121             Fin Si
122         Sinon
123             // Les bonus sont inaccessibles, on cherche le chemin qui nous
                ↪  rapproche le plus d'un bonus
124             Retourner move_to_skipped
125         Fin Si
126     Fin
```

Listing 6.10: *Pseudo-code de la fonction* `lode_runner`.

7

Évaluation Expérimentale

Afin de valider les performances de notre stratégie, nous avons utilisé un script `sh`, qui permet de lancer plusieurs parties et de récupérer les résultats. Ce script nous renvoie, pour un niveau donnée et un nombre de parties donné, le pourcentage de victoire, et le nombre moyen de déplacements effectués par le runner.

7.1 Niveau 0

7.1.1 Résultats

Pourcentage de victoire	Moyenne de déplacements	Moyenne de bombes
100%	87.0	0.0

Table 7.1: *Résultats pour le niveau 0 sur 1000 parties*

7.1.2 Analyse des défaites

Il n'y a pas de défaites pour le niveau 0 et le nombre moyen de déplacements est entier, car n'y ayant pas d'ennemi, le runner prend toujours le même chemin.

7.2 Niveau 1

7.2.1 Résultats

Pourcentage de victoire	Moyenne de déplacements	Moyenne de bombes
100%	125.0	0.0

Table 7.2: *Résultats pour le niveau 1 sur 1000 parties*

7.2.2 Analyse des défaites

Il n'y a pas de défaites pour le niveau 1 et le nombre moyen de déplacements est entier. Pourtant il y a un ennemi, mais le runner le contourne, et prend toujours le même chemin.
On remarque que la seule mécanique qui rend le jeu non-déterministe est la position des ennemis quand ils réapparaissent. Or, le solution que trouve notre stratégie contourne les ennemis, donc si elle réussi le niveau une fois, elle le réussira toujours.

7.3 Niveau 2

7.3.1 Résultats

Pourcentage de victoire	Moyenne de déplacements	Moyenne de bombes
100%	174.0	0.0

Table 7.3: *Résultats pour le niveau 2 sur 1000 parties*

7.3.2 Analyse des défaites

Il n'y a pas de défaites pour le niveau 2, pour les mêmes raisons que pour le niveau 1.

7.4 Niveau 3

7.4.1 Résultats

Pourcentage de victoire	Moyenne de déplacements	Moyenne de bombes
98.2%	161.7	2.7

Table 7.4: *Résultats pour le niveau 3 sur 10000 parties*

7.4.2 Analyse des défaites

A faire

7.5 Niveau 4

7.5.1 Résultats

Pourcentage de victoire	Moyenne de déplacements	Moyenne de bombes
96.9%	189.4	1.5

Table 7.5: *Résultats pour le niveau 4 sur 10000 parties*

7.5.2 Analyse des défaites

A faire

7.6 Niveau supplémentaire

Afin de tester notre stratégie sur un niveau plus difficile, nous avons créé un niveau supplémentaire. Il reprend les mêmes plateformes que le niveau 3, mais avec des ennemis en plus.

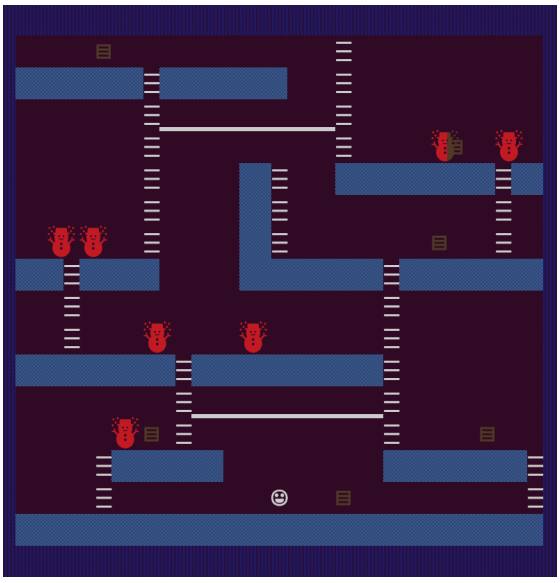

Figure 7.1: *Niveau supplémentaire*

7.6.1 Résultats

Pourcentage de victoire	Moyenne de déplacements	Moyenne de bombes
50.9%	209.2	13.9

Table 7.6: *Résultats pour le niveau supplémentaire sur 1000 parties*

7.6.2 Analyse des défaites

A faire

8

Conclusion

Ce projet autour du jeu Lode Runner nous a permis de concevoir et d'implémenter une intelligence artificielle pour jouer au jeu. Malgré des contraintes, nous avons pu élaborer une stratégie robuste basée sur l'algorithme A* pour gérer au mieux les situations rencontrées.
L'évaluation expérimentale a démontré l'efficacité de notre IA sur des niveaux variés, atteignant un taux de réussite de 100% pour les niveaux les plus simples et maintenant des performances honorables face à des niveaux plus difficiles. Cela reflète d'une cohérence entre la conception algorithmique de notre stratégie et son application.
Ces résultats montrent aussi les limites de notre IA, notamment face à des niveaux plus complexes, où des améliorations pourraient être apportées pour gérer des situations avec plus d'ennemis.

Même si j'étais en filière MPI l'année dernière, j'ai apprécié de travailler sur ce projet, qui m'a permis de pousser mes compétences en algorithmique et en programmation. Enfin, la rédaction de ce rapport m'a fait beaucoup apprendre, car c'est la première fois que je rédige un rapport aussi long et complet, j'ai aussi pu m'initier à LaTeX, que je n'avais jamais utilisé auparavant.