

ENSSAT

L A N N I O N

Compte rendu de projet : Algorithme 1

Lode Runner

Julien BOURDET

ENSSAT

1^{ère} année - Informatique

Lannion, December 2024

TABLE DES MATIÈRES

1	Introduction	3
2	Préliminaires	4
2.1	Tas-Min	4
2.1.1	Procédure insert	5
2.1.2	Procédure extract_min	5
2.1.3	En C	6
2.2	Algorithme A*	6
2.2.1	Pseudo-code	7
2.2.2	En C	8
3	Stratégie	9
3.1	Mouvements spéciaux	11
3.1.1	Mouvement de combat	11
3.1.2	Mouvement de rapprochement	11
4	Paramètres et Valeur de retour	12
4.1	Détails	12
4.2	Structure levelinfo	13
5	Modules	15
5.1	Liste des modules	15
5.1.1	Tas-Min	15
5.1.2	A*	16
5.1.3	Outils	16
6	Présentation des modules	17
6.1	Module is_valid	17
6.1.1	Description	17
6.1.2	Paramètres	17
6.1.3	Choix d'Implémentation	17
6.1.4	Pseudo-code	18
6.2	Module get_closest_bonus	20
6.2.1	Description	20

6.2.2	Paramètres	20
6.2.3	Choix d'Implémentation	20
6.2.4	Pseudo-code	21
6.3	Module <code>add_ennemis</code>	22
6.3.1	Description	22
6.3.2	Paramètres	22
6.3.3	Choix d'Implémentation	22
6.3.4	Pseudo-code	22
6.4	Module <code>combat_moves</code>	24
6.4.1	Description	24
6.4.2	Paramètres	24
6.4.3	Choix d'Implémentation	24
6.4.4	Pseudo-code	24
6.5	Module <code>lode_runner</code>	28
6.5.1	Description	28
6.5.2	Paramètres	28
6.5.3	Choix d'Implémentation	28
6.5.4	Pseudo-code	28
7	Évaluation Expérimentale	32
7.1	Niveau 0	32
7.1.1	Résultats	32
7.1.2	Analyse des défaites	32
7.2	Niveau 1	33
7.2.1	Résultats	33
7.2.2	Analyse des défaites	33
7.3	Niveau 2	33
7.3.1	Résultats	33
7.3.2	Analyse des défaites	33
7.4	Niveau 3	34
7.4.1	Résultats	34
7.4.2	Analyse des défaites	34
7.5	Niveau 4	35
7.5.1	Résultats	35
7.5.2	Analyse des défaites	35
7.6	Niveau supplémentaire	36
7.6.1	Résultats	36
7.6.2	Analyse des défaites	36
7.7	Conclusion et persepctives d'amélioration	37
8	Conclusion	38
	<i>References</i>	39

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 intelligence artificielle (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 strict force l'IA à développer des stratégies approfondies permettant de se projeter dans le futur pour ne pas réaliser des actions inutiles ou contre productives

Au cours du développement, j'ai rencontré plusieurs défis. L'absence de mémorisation empêche d'utiliser une stratégie trop coûteuse en temps de calcul ou de planifier une stratégie sur plusieurs tours, tandis que le fait de 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.

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 ([Wikipédia, 2024](#)) 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 ayant 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.

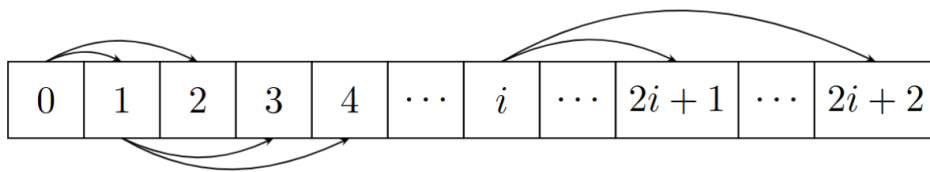


Figure 2.1: Représentation d'un Tas-min sous forme de tableau.

Pour gérer les Tas-min, nous avons besoin de deux procédures : `insert` et `extract_min`. La première permet d'ajouter un élément au tas et la deuxième 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.

```

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 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.

```

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.*

2.1.3 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; // Priorité 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 :

```
1     min_heap* create_min_heap(int);
2     void free_min_heap(min_heap*);
3     void percolate_up(min_heap*, int);
4     void percolate_down(min_heap*, int);
5     void insert(min_heap*, int, float);
6     void modify_priority(min_heap*, int, float);
7     int extract_min(min_heap*);
8     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, dans laquelle 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

```

1      Fonction a_star en @path
2      Parametres :
3          origin en entier
4          destination en entier
5          level en niveau
6      Déclarations :
7          pat en @path
8          u, v en entier
9          h_v en flottant
10     Début
11         pat <- create_path // On initialise le chemin
12         // pat.d est le tableau des distances, pat.p est le tableau des parents
13         pat.d[origin] <- origin
14         insert(pat.heap, origin, 0) // On ajoute le point d'origine a la file
15
16     Tant que pat.heap n'est pas vide
17         // Sommet courant, celui avec la priorite minimale
18         u <- extract_min(pat.heap)
19
20         Si u = destination
21             // On a trouve le bonus, on pourra remonter le chemin grace a pat.p
22             pat.found <- VRAI
23             Sortir de la boucle
24         Fin Si
25
26     Pour chaque action possible
27         Si l'action est valide // Dépend de level
28             v <- position apres l'action
29             h_v <- distance entre v et le bonus (l'heuristique)
30             Si pat.d[u] + poids de l'action < pat.d[v]
31                 // Si on a trouve un chemin plus court
32                 pat.d[v] <- pat.d[u] + poids de l'action
33                 pat.p[v] <- u
34                 Si v n'est pas dans la file
35                     insert(pat.heap, v, pat.d[v] + h_v) // On l'ajoute
36                 Sinon
37                     // On modifie sa priorite
38                     modify_priority(pat.heap, v, pat.d[v] + h_v)
39                 Fin Si
40             Fin Si
41         Fin Si
42     Fin Pour
43     Fin Tant que
44
45     Retourner pat
46     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.

```
1     typedef struct path{
2         min_heap* heap; // File de priorité
3         int* p; // Tableau des parents
4         int* d; // Tableau des distances
5         bool found; // Chemin trouvé ou non
6     } path;
```

Listing 2.6: Structure *path* en C.

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 essaie, 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 (ce qui devient plus probable à mesure que le nombre d'ennemis augmente), le runner utilise des fonctions de mouvements dits "spéciaux" pour se déplacer.

Ceci signifie 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, c'est le chemin le plus court.
3. Si A^* ne trouve pas de chemin, on est 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. Autrement dit 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, contrairement à l'imprécision de la position des ennemis qui augmente avec la distance, la dangerosité des ennemis diminue. L'algorithme peut être à peu près sûr de la position des ennemis proches (les ennemis dangereux), et donc de les éviter.

De plus, au lieu de passer la carte à A^* , on lui passe une carte modifiée, dans laquelle les ennemis sont ajoutés, ainsi qu'une zone d'ennemis autour de chaque ennemi. Cela permet d'éviter que le runner ne se rapproche à moins d'une case d'un ennemi, et ainsi de s'assurer que, même si le prochain tour est celui des ennemis, le runner ne sera pas en danger.

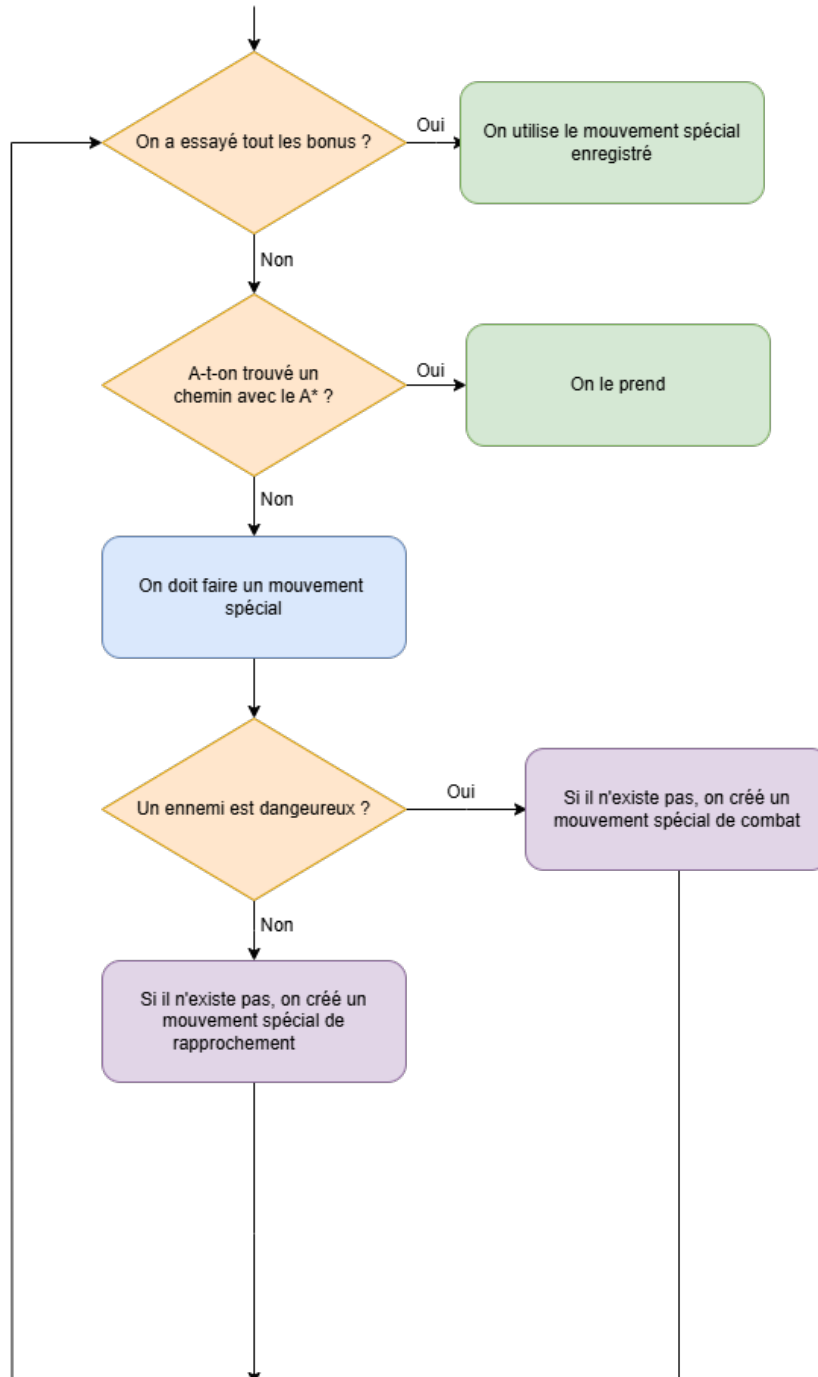


Figure 3.1: Logigramme de la stratégie.

3.1 Mouvements 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 lors d'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.

PARAMÈTRES ET VALEUR DE RETOUR

Afin de permettre à notre code d'interagir 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 Détails

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*.

Cette fonction est appelée 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.

4.2 Structure 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. 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.

MODULES

La stratégie se décompose en plusieurs modules, chacun ayant un rôle spécifique. Ces modules sont indépendants les uns des autres et peuvent être utilisés individuellement.

5.1 Liste des modules

5.1.1 Tas-Min

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

Table 5.1: Modules relatifs au tas-min.

5.1.2 A*

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

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

5.1.3 Outils

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

Table 5.3: Modules relatifs aux outils.

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 tous les voisins d'une case donnée. On appelle voisin d'une case une case atteignable en un seul mouvement.

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
<code>pos</code>	<code>int</code>	Position de la case à tester.
<code>a</code>	<code>action</code>	Action à tester.
<code>level</code>	<code>levelinfo</code>	Structure contenant les informations du niveau.
<code>air_level</code>	<code>levelinfo</code>	Structure contenant les informations du niveau sans les ennemis.

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

6.1.3 Choix d'Implémentation

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 &&
21             ↪ air_map[y + 1][x] != BOMB) || air_map[y - 1][x] == CABLE
22         // Si la case en dessous du joueur n'est pas un chemin, un cable ou une
23             ↪ bombe, ou si la case au dessus du joueur est un cable, alors le joueur
24             ↪ n'est pas en l'air
25
26     Selon a faire
27         Cas NONE :
28             Retourner VRAI
29         Cas UP :
30             // On ne peut monter que si on est sur une echelle et qu'il n'y a
31             ↪ pas de mur au dessus
32             Si map[y][x] == LADDER et map[y - 1][x] != WALL et map[y - 1][x] !=
33             ↪ FLOOR et map[y - 1][x] != ENEMY alors
34                 Retourner VRAI
35         Cas DOWN :
36             // On ne peut descendre que si il y a une echelle, un chemin ou un
37             ↪ cable en dessous
38             // On laisse la possibilite de descendre si le joueur est en l'air
39             Si (map[y + 1][x] == LADDER ou map[y + 1][x] == PATH ou map[y +
40             ↪ 1][x] == CABLE) et map[y + 1][x] != ENEMY alors
41                 Retourner VRAI
42         Cas LEFT :
43             // On ne peut aller a gauche que si il n'y a pas de mur a gauche et
44             ↪ que le joueur n'est pas en l'air
45             // C'est un petit hack car le moteur avance de plusieurs tour de
46             ↪ jeu sans utiliser le code du joueur tant qu'il tombe, mais le
47             ↪ A* ne le sait pas,
48             // alors on se débrouille pour que la seule action possible soit
49             ↪ 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     Default :
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

A*, nécessite un point de départ et un point d'arrivée. Dans notre cas, le point de départ est le runner, et les points d'arrivée sont les bonus. Mais, afin de finir le niveau plus rapidement (et pour éviter des cas de boucles infinies), on cherche à se rendre sur le bonus le plus proche du runner.

De plus, si le bonus le plus proche est inaccessible pour une raison quelconque, on passe au deuxième plus proche, et ainsi de suite. C'est pourquoi on a besoin de la liste des bonus déjà traités, pour ne pas les traiter une seconde fois.

6.2.2 Paramètres

Paramètre	Type	Description
<code>bonusl</code>	<code>bonus_list</code>	Liste des bonus.
<code>runner</code>	<code>character_list</code>	Liste des personnages.
<code>already_seen</code>	<code>bonus_list</code>	Liste des bonus déjà vus.

Table 6.2: *Paramètres de la fonction `get_closest_bonus`.*

6.2.3 Choix d'Implémentation

Les listes de bonus sont des listes chaînées, on choisit donc une boucle `Tant que` pour parcourir la liste des bonus. 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
19     ↪ très grande
20     current <- bonusl // On initialise le bonus courant à la tête de la liste
21     ↪ des bonus
22
23     Tant que current != NULL faire
24         // On parcourt la liste des bonus
25         Si dist(current->b.x, current->b.y, runner->c.x, runner->c.y) <
26         ↪ best_dist et non is_in_bonus_list(current, already_seen) alors
27             closest_bonus <- current
28             best_dist <- dist(current->b.x, current->b.y, runner->c.x,
29             ↪ runner->c.y)
30         Fin Si
31         current <- current->next // On passe au bonus suivant
32     Fin Tant que
33
34     Retourner closest_bonus // On retourne le bonus le plus proche
35 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
      ↪ bomb_l);

```

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 créés par les bombes, on ajoute les ennemis et les bombes à la carte. C'est-à-dire qu'on crée une nouvelle carte, où les cases contenant des ennemis sont marquées comme telles. 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'Implémentation

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 personnage, 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 <- character1 // On initialise current à la tête de la liste des
19     ↪ personnages
20
21     Tant que current != NULL faire
22         // On itère sur les personnages
23         Si current->c.item == ENEMY alors
24             // Si le personnage est un ennemi (on pourrait tomber sur le
25             ↪ runner)
26             Si level.map[current->c.y][current->c.x] == BOMB ou
27             ↪ level.map[current->c.y][current->c.x] == DEAD alors
28                 level.map[current->c.y][current->c.x] <- DEAD
29             Sinon
30                 level.map[current->c.y][current->c.x] <- ENEMY
31             Fin Si
32         Fin Si
33         current <- current->next // On passe au personnage suivant
34     Fin Tant que
35
36     Retourner level
37
38 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 dangereux, on utilise la procédure `combat_moves`.

6.4.2 Paramètres

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

Table 6.4: Paramètres de la procédure `combat_moves`.

6.4.3 Choix d'Implémentation

La procédure étant assez complexe, 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 utilise une procédure car on modifie le pointeur `move_to_combat` en fonction de sa valeur actuelle.

6.4.4 Pseudo-code

```
1 Procéder 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
31        ↪ tombe pas en le faisant
32        can_right <- is_valid(runner->c.y * level.xsize + runner->c.x, RIGHT,
33        ↪ level, level)
34        can_left <- is_valid(runner->c.y * level.xsize + runner->c.x, LEFT, level,
35        ↪ level)
36        can_right <- can_right et (down_right != BOMB et down_right != PATH)
37        can_left <- can_left et (down_left != BOMB et down_left != PATH)
38
39        Si closest_enemy != NULL alors
40            // Si il y a un ennemi dangereux (donc que l'on est en mode combat)
41            Si *move_to_combat == -1 alors
42                // Si on n'a pas encore décidé de comment se déplacer
43                distance <- runner->c.y - closest_enemy->c.y // On calcule la
44                ↪ distance verticale
45                Si distance == 0 alors
46                    // Combat horizontal
47                    distance <- runner->c.x - closest_enemy->c.x // On calcule la
48                    ↪ distance horizontale
49                Si level.map[runner->c.y - 1][runner->c.x] == CABLE et
50                ↪ level.map[runner->c.y + 1][runner->c.x] == PATH alors
51                    // On est sur un cable, on ne peut pas poser de bombe, on
52                    ↪ saute
53                    *move_to_combat <- DOWN
54                Sinon Si distance > 0 et distance < 4 alors
55                    // 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 +
95             ↪ runner->c.x, UP, level, level)
96             bool can_down <- is_valid(runner->c.y * level.xsize +
97             ↪ runner->c.x, DOWN, level, level)
98             Si distance > 0 et can_down alors
99                 // On descend si un ennemi est au dessus
100                 *move_to_combat <- DOWN
101             Sinon Si distance < 0 et can_up alors
102                 // On monte si un ennemi est en dessous
103                 *move_to_combat <- UP
104             Fin Si
105
106             Si level.map[runner->c.y][runner->c.x] == PATH ou
107             ↪ level.map[runner->c.y + 1][runner->c.x] == FLOOR alors
108                 // Si on est en haut ou en bas d'une echelle, alors c'est
109                 ↪ une erreur et on est en pas en mode combat, on remet
110                 ↪ move_to_combat a -1
111                 *move_to_combat <- -1
112             Fin Si
113         Fin Si
114     Fin Si
115 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
<code>level</code>	<code>levelinfo</code>	Structure contenant les informations du niveau.
<code>characterl</code>	<code>character_list</code>	Liste des personnages.
<code>bonusl</code>	<code>bonus_list</code>	Liste des bonus.
<code>bombl</code>	<code>bomb_list</code>	Liste des bombes.

Table 6.5: *Paramètres de la fonction `lode_runner`.*

6.5.3 Choix d'Implémentation

Le point principal de cette fonction est l'itération sur les bonus. Elle est donc centrée autour d'une boucle tant que (les bonus 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(character1) // On récupère le runner
23         level <- add_enemies(level, character1, bombl) // On ajoute les ennemis à
           ↳ la carte
24         // On crée 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, character1)
26
27         already_seen <- NULL // On initialise la liste des bonus déjà vus
28         closest_bonus <- get_closest_bonus(bonus1, runner, already_seen) // On
           ↳ récupère le bonus le plus proche
29
30         Si bonus1 == 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
62             ↪ continuer la boucle
63             closest_bonus <- get_closest_bonus(bonusl, runner,
64             ↪ already_seen)
65         Fin Si
66         Continuer
67     Fin Si
68
69     Si move_to_closest == -1 alors
70         // Si on n'a toujours initialisé move_to_closest, on le fait
71         // move_to_closest est l'action qui nous rapproche du bonus le plus
72         ↪ proche (même s'il est inaccessible)
73         int runner_pos <- runner->c.y * level.xsize + runner->c.x
74         int closest_bonus_pos <- closest_bonus->b.y * level.xsize +
75         ↪ closest_bonus->b
76
77         c <- find_closest_child(pat->p, runner_pos, closest_bonus_pos,
78         ↪ level) // On trouve le chemin qui nous rapproche le plus du
79         ↪ bonus
80         move_to_closest <- get_action(runner_pos, c->pos, level) // On
81         ↪ stocke l'action à effectuer
82     Fin Si
83
84     Si pat->found alors
85         // Si on a trouvé un chemin, on le suit
86         v <- closest_bonus->b.y * level.xsize + closest_bonus->b.x
87         Tant que pat->p[v] != runner->c.y * level.xsize + runner->c.x faire
88             // pat->p[v] est le parent de v, on remonte le chemin pour
89             ↪ trouver l'action à effectuer
90             v <- pat->p[v]
91         Fin Tant que
92
93         move_to_path <- get_action(runner->c.y * level.xsize + runner->c.x,
94         ↪ v, level) // On stocke l'action à effectuer
95
96     Sinon Si pat->heap->size != 0 alors
97         // Si le chemin est plus long que la taille du tas, c'est une
98         ↪ erreur
99         Afficher "ERROR: Path is longer than heap size"
100        Sortir du programme
101    Sinon
102        // Si on n'a pas trouvé de chemin, on est en mode combat
103        combat_moves(runner, get_closest_enemy(characterl, runner, level),
104        ↪ &move_to_combat, level)
105    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
103            ↪ continuer la boucle
104            closest_bonus <- get_closest_bonus(bonusl, runner, already_seen)
105        Sinon
106            closest_bonus <- NULL
107        Fin Si
108    Fin Tant que
109
110    Si move_to_path != -1 alors
111        // On a trouvé un chemin, on le suit
112        Retourner move_to_path
113    Sinon Si move_to_combat != -1 alors
114        // On est en mode combat
115        Retourner move_to_combat
116    Sinon Si move_to_closest != -1 alors
117        // On n'a pas trouvé de chemin, on se rapproche du bonus
118        Si is_valid_closest(runner->c.y * level.xsize + runner->c.x,
119        ↪ move_to_closest, astar_level) alors
120            Retourner move_to_closest
121        Sinon
122            Retourner NONE
123        Fin Si
124    Sinon
125        // Les bonus sont inaccessibles, on cherche le chemin qui nous
126        ↪ rapproche le plus d'un bonus
127        Retourner move_to_skipped
128    Fin Si
129    Fin

```

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

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 donné et un nombre de parties donné, le pourcentage de victoires, et le nombre moyen de déplacements effectués par le runner.

7.1 Niveau 0

7.1.1 Résultats

Pourcentage de victoires	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 a pas d'ennemi, le runner prend toujours le même chemin.

7.2 Niveau 1

7.2.1 Résultats

Pourcentage de victoires	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 lorsqu'ils réapparaissent. Or, la solution que trouve notre stratégie est de contourner les ennemis, donc si elle réussit le niveau une fois, elle le réussira toujours.

7.3 Niveau 2

7.3.1 Résultats

Pourcentage de victoires	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 victoires	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

Malgré un pourcentage de victoires élevé, il y a des défaites pour le niveau 3. Analysons une de ces défaites.

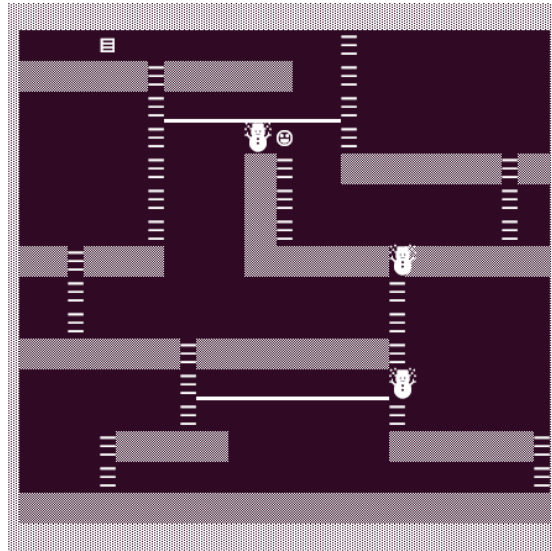


Figure 7.1: Défaite pour le niveau 3.

À cette position, le runner joue DOWN, ce qui le coince entre deux ennemis, l'empêchant d'avoir le temps ou l'espace nécessaire pour utiliser ses bombes. En mode débogage, on constate que c'est le mode de rapprochement qui était activé.

À ce moment, il ne reste qu'un seul bonus, mais il est inatteignable car les ennemis bloquent tous les chemins. Dans ce cas, il est normal que le mode de mouvement spécial soit utilisé. Cependant, le runner aurait dû se diriger vers la droite.

Cette situation résulte d'une erreur : le mode de combat aurait dû être activé.

7.5 Niveau 4

7.5.1 Résultats

Pourcentage de victoires	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

Une des défaites les plus courantes pour le niveau 4 est la suivante :

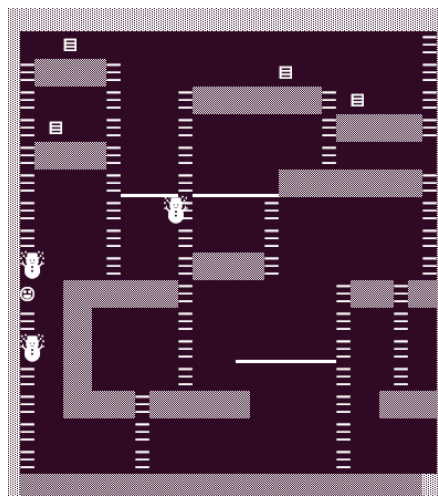


Figure 7.2: Défaite pour le niveau 4.

À cette position, le runner reste immobile, car il est bloqué entre deux ennemis. Pourtant, il aurait pu jouer RIGHT pour se dégager.

Cette erreur provient du mode de mouvement en combat : lorsqu'un combat se déroule sur une échelle, le runner se limite à essayer de monter ou descendre. Il aurait dû également envisager de se déplacer vers la gauche ou la droite pour se libérer.

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.

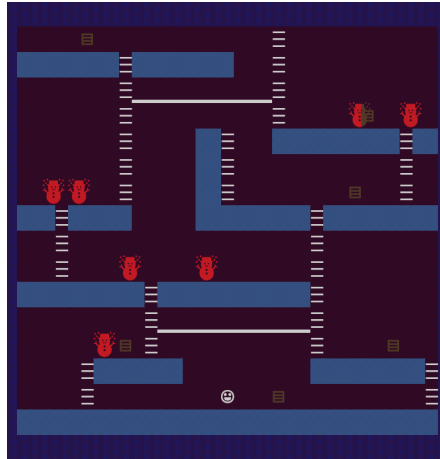


Figure 7.3: Niveau supplémentaire

7.6.1 Résultats

Pourcentage de victoires	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

Sur ce dernier niveau, il y a beaucoup de défaites, car il y a beaucoup d'ennemis.

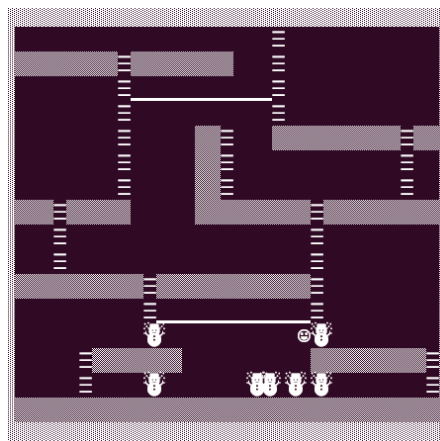


Figure 7.4: Défaite pour le niveau supplémentaire.

À cette position, le runner joue DOWN. C'est un mouvement de combat qui vise, initialement, à ne pas combattre sur les cables, car on ne peut pas y poser de bombes. De plus, il n'y a pas d'ennemi pile en dessous du runner, il joue donc DOWN. C'est une erreur du mode de mouvement en combat, car le runner aurait dû se déplacer vers la gauche pour éviter les ennemis.

7.7 Conclusion et perspectives d'amélioration

Notre stratégie a montré de bons résultats sur les premiers niveaux, mais elle a montré ses limites sur les niveaux plus difficiles.

Les erreurs proviennent principalement des modes de mouvement en combat, qui ne prennent pas en compte toutes les possibilités de déplacement.

C'est la limite de notre stratégie : elle ne peut pas prendre en compte toutes les situations possibles. Il faudrait trouver une stratégie plus générale, qui ne prennent pas en compte des cas particuliers, mais qui soit capable de s'adapter à toutes les situations. Je pense qu'estimer la position futures des ennemis et l'utiliser dans A* pourrait être une piste d'amélioration.

CONCLUSION

Ce projet nous a permis de concevoir et d'implémenter une intelligence artificielle pour jouer au jeu Lode Runner. 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 davantage d'ennemis.

Bien qu'ayant été en MPI l'année dernière, j'ai apprécié 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 beaucoup appris, 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.

REFERENCES

Wikipédia (2024). "Tas binaire — Wikipédia, l'encyclopédie libre". In: [En ligne; Page disponible le 2-août-2024]. URL: http://fr.wikipedia.org/w/index.php?title=Tas_binaire.