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 $2i + 2$.
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 `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`.*

2.1.3 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.

2.1.4 En C

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

```c
typedef struct min_heap{
    int size; // Nombre d'elements dans le tas
    int capacity; // Taille max du tas
    int* array; // Tableau des elements
    float* priority; // Prioritee des elements
  } min_heap;
```

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

On dispose de 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. On utilisera ici le tas-min définit précédemment 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 (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 foncion renvoie un chemin, qui est une structure contenant les informations nécessaires pour retrouver le chemin trouvé.

```
1        typedef struct path{
2            min_heap* heap;
3            int* p;
4            int* d;
5            bool found;
6            } path;
```

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

3

Stratégie

La stratégie utilisée pour terminer le jeu est construite autour de l'algorithme A*.
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* à une priorité sur toutes les autres décisions.
Si l'algorithme ne trouve aucun chemin vers aucun bonus (au plus il y a d'ennemi, 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 dangeureux, 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 l'algorithme 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 dangeureux), et donc de les éviter.

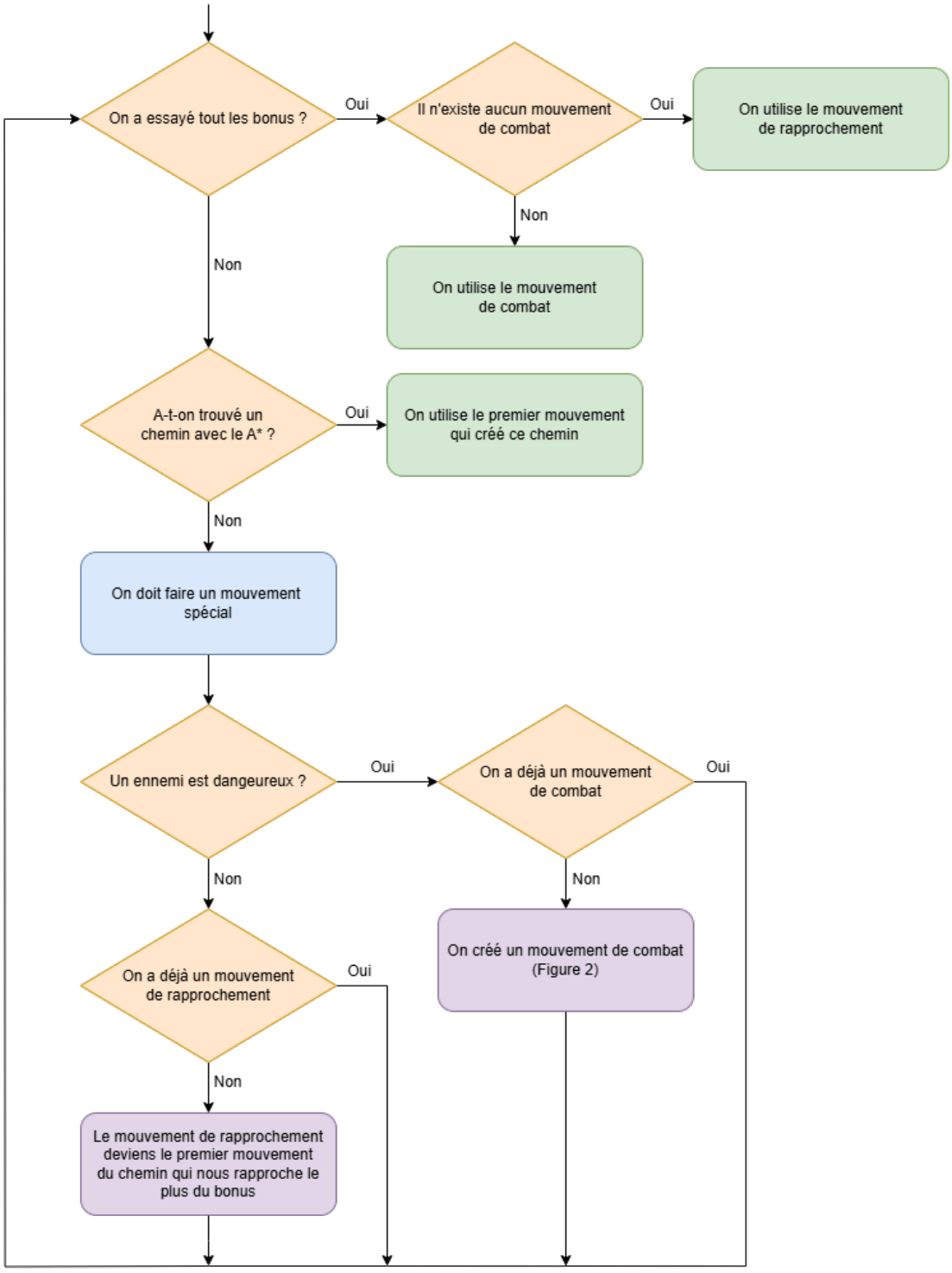

Figure 3.1: *Logigramme de la stratégie.*

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 dangeureux (**??**) et qu'il n'existe pas déjà un mouvement de combat. Alors, on va utiliser une suite de conditions (**??**) 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 dangeureux, 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 iterer sur touts 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

Conclusion

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

This page intentionally left blank.