

Projet Lode Runner : Rapport

ENSSAT

L A N N I O N

Table des matières

Introduction	Page 3
I. Préliminaires	Page 4
I.1. Listes chaînées	Page 4
I.2. Algorithme de Dijkstra	Page 5
II. Vocabulaire	Page 6
III. Modélisation	Page 7
III.1 Les parcelles (plots)	Page 8
III.2 Les Échelles (Objet) (Ladders)	Page 9
III.3 Le graphe d'accessibilité (plots_accessibility_graph)	Page 10
IV. Grandes lignes de la stratégie	Page 12
V. Points clés de la stratégie	Page 14
V.1. Vérification de la légalité d'une action	Page 14
V.2. Construction du graphe d'accessibilité	Page 15
V.3. Calculs de meilleur chemin et de poids utile	Page 18
V.4. Recherche de la position x de la première Échelle (Objet) à emprunter	Page 20
V.5. Mise à jour du graphe d'accessibilité en tenant compte des ennemis	Page 22
VI. Évaluation expérimentale	Page 24
Conclusion	Page 25
Sources	Page 26

Introduction

Ce rapport décrit les moyens algorithmiques employés dans le but de concevoir une Intelligence Artificielle capable de jouer par elle-même au jeu *Lode Runner*. La principale contrainte qui nous est imposée est qu'il n'y a aucune mémorisation d'information possible, autrement dit que notre algorithme doit prendre une décision avec l'état du plateau actuel et uniquement avec cet état.

Le jeu *Lode Runner* est doté de plusieurs difficultés, variant en fonction du nombre d'ennemis présents sur une map. 4 plateaux différents nous sont proposés, allant de 0 à 3 ennemis. Mon algorithme se révèle très efficace sur le premier niveau, puis de moins en moins en fonction de l'augmentation du nombre d'ennemis. Il fournit tout de même des parties gagnantes à chacun de ces niveaux.

Nous verrons dans un premier temps quelques préliminaires essentielles à la compréhension de mon algorithme, puis nous étudierons la modélisation que j'ai utilisé ainsi que le vocabulaire qui lui est associé, avant de nous intéresser aux grandes lignes de la stratégie, stratégie que nous étudierons ensuite plus en détails. Enfin, nous concluerons.

I. Préliminaires

I.1. Listes chaînées

Seront manipulés des objets du type suivant :

```
typedef struct s_list int_list;
struct s_list
{
    int_list *next;    //pointeur sur le reste de la liste
    int data;          //donnée
};
```

Les listes n'étant pas natives dans le langage C, cette alternative est suivant utilisée afin de manipuler des listes. La donnée d'un maillon est enregistrée dans `data`, tandis que `next` pointe vers le prochain maillon. Le dernier maillon pointe vers un maillon nul, c'est-à-dire vers `NULL`.

Plusieurs fonctions et procédures seront utilisées pour manipuler ces listes :

```
34 int_list* create_and_initialize_int_list(int);
35 bool is_in_int_list(int_list*, int);
36 void add_if_new_int_list(int_list**, int);
37 int_list* copy_int_list(int_list*);
38 void remove_first_element_int_list(int_list**);
39 void free_int_list(int_list*);
```

Les fonctions renvoient quelque chose, par exemple `copy_int_list` renvoie une copie de la liste passée en entrée, tandis que les procédures se basent sur des effets de bord en modifiant la liste passée en entrée. Ces modifications sont possibles car on effectue des passages par adresse dans les arguments. Par exemple, `add_if_new_int_list` ajoute un élément à la liste passée en entrée, car celle-ci est fournie par adresse. C'est bien un `int_list**` qui est passé en argument, car une liste quelconque est de type `int_list*` (correspond à un pointeur vers le premier maillon de la liste).

I.2. Algorithme de Dijkstra

Sera utilisé un graphe, modélisant certaines parties du plateau et les accessibilités entre ces différentes parties. Afin d'exploiter ce graphe, il est nécessaire de calculer un plus court chemin entre plusieurs de ses sommets. Pour ce faire, j'ai choisi d'utiliser l'algorithme de Dijkstra.

Celui-ci permet de construire 2 tableaux, Prédecesseurs et Distances, à partir desquels il est possible de retrouver tous les plus courts chemins à partir d'un même sommet source dans un graphe orienté pondéré, ainsi que les poids de tous ces chemins. Pour ce faire, il utilise notamment une file de priorité minimale, à savoir une file dans laquelle à chacun des éléments est associée une priorité. L'élément avec la file de priorité minimal sera extraite de la file en premier.

Le fonctionnement de l'algorithme est le suivant :

Entrée : Un graphe $G = (S, A)$ avec sa fonction de pondération associée $p(s_1 \in S, s_2 \in S) \rightarrow \text{entier}$.

Un sommet source $s_0 \in S$

Sortie : Tableau Distances à $|S|$ cases

Tableau Prédecesseurs à $|S|$ cases

On initialise toutes les cases de Distances à l'infini, sauf s_0 que l'on met à 0.

On initialise toutes les cases de Prédecesseurs à -1.

On crée une file de priorité minimale F et on y ajoute tous les sommets, avec comme priorité la distance qui leur est associée dans le tableau Distances.

Tant que la queue n'est pas vide :

On en extrait le sommet s_1 de priorité minimale.

Pour tout sommet s_2 accessible depuis s_1 et encore dans F :

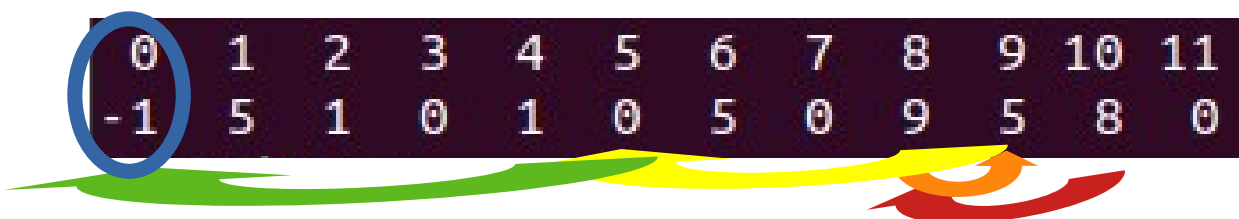
On calcule $D = \text{Distances}[s_1] + p(s_1, s_2)$.

Si $D < \text{Distances}[s_2]$:

On met à jour $\text{Distances}[s_2]$ en y enregistrant D .

On retient le nouveau chemin menant à s_2 en enregistrant s_1 dans $\text{Prédecesseurs}[s_2]$.

Ainsi, il est aisé de reconstruire tout plus court chemin entre s_0 et un autre sommet du graphe en remontant dans Prédecesseurs. Si Prédecesseurs, avec comme sommet source 0, ressemble à ça :



0	1	2	3	4	5	6	7	8	9	10	11
-1	5	1	0	1	0	5	0	9	5	8	0

The diagram shows a 2x12 grid representing the 'Prédecesseurs' table. The first row contains indices 0 through 11. The second row contains the predecessor values. The cell at (0, -1) is circled in blue. A green arrow points from this cell to the cell at (5, 0). A yellow arrow points from (5, 0) to the cell at (9, 5). An orange arrow points from (9, 5) to the cell at (8, 0). A red arrow points from (8, 0) to the cell at (10, 8).

Pour aller de 0 à 10, il faut effectuer le chemin suivant :

0 → 5 → 9 → 8 → 10

II. Vocabulaire

Français	Anglais	Description
bloc = case	bloc	C'est une case en particulier. Il y en a $xsize*ysize$ sur un plateau et leur donnée peut être : B pour Bonus/Bonus, C pour Cable/Câble, E pour Enemy/Ennemi, F pour Floor/Sol, L pour ladder/échelle, R pour Runner/Héro, W pour Wall/Mur, X pour Exit/Sortie, . pour Void/Vide.
plateau = carte	map	C'est la donnée d'un assortiment de $xsize*ysize$ cases.
parcelle	plot	C'est un groupe de cases accolées possédant la même coordonnée y et inter-accessibles. Cf modélisation page 8 pour plus de précision.
parcelle de câble	plot_cable	C'est un cas particulier de parcelle issue de la présence de plusieurs blocs Câbles (C) à la suite. Cf modélisation page 8 pour plus de précision.
Échelle (Objet)	Ladder	C'est un groupe de cases accolées possédant la même coordonnée x et inter-accessibles, issue d'une suite de blocs échelle (L). Attention à ne pas confondre l'Échelle (Objet), représentant un groupe de blocs, avec l'échelle (L), qui n'est qu'un seul bloc. Cf modélisation page 9 pour plus de précision.
menace	threat	Un ennemi à une distance euclidienne inférieure ou égale à 2 du Héro est considéré comme une menace.
graphe d'accessibilité	plots_accessibility_graph	C'est un graphe orienté pondéré. Chaque sommet représente une parcelle et les arcs représentent les accessibilités directes (sans avoir à passer par une autre parcelle) entre ces parcelles. Cf modélisation page 10 pour plus de précision.
poids	weight	Peut représenter la fonction de pondération utilisée dans le graphe d'accessibilité ou la valeur renvoyée par cette fonction. Le poids de l'arc entre plot1 et plot2 correspond à la plus courte distance euclidienne d'un chemin permettant d'aller d'une case de plot1 à une case de plot2.
poids utile	practical_weight	Peut également représenter une fonction ou une valeur. À la différence du poids, le poids utile représente la plus petite distance euclidienne entre 2 cases du plateau, et non entre 2 parcelles. Le poids utile de bloc1 à bloc2 n'est défini que si bloc1 et bloc2 sont sur des parcelles respectives plot1 et plot2, avec plot2 directement accessible depuis plot1.
identifiant	identifier	L'identifiant d'une case est un entier qui permet d'identifier cette case de manière unique. Il est calculé de la manière suivante : $identifier = x + y*(xsize)$

III. Modélisation

On divise le plateau en plusieurs zones afin de disposer d'une vue d'ensemble à chaque tour, qui nous permet de faire un choix de direction plus éclairé. Cette division rend également le parcours de graphe nécessaire à trouver un plus court chemin plus efficace.

Remarque : Ce gain d'efficacité se manifesterait de manière plus importante si nous disposions d'une mémoire d'un tour à l'autre, qui nous permettrait de ne pas avoir à reconstruire les modélisations des différentes zones à chaque tour. En effet, le fait d'avoir à reconstruire les modélisation à chaque tour est assez coûteux, ce qui compense en grande partie le gain d'efficacité gagné par le parcours d'un graphe possédant moins de sommets.

Les 2 types de zones principaux sont les parcelles et les Échelles (Objet).

III.1. Les parcelles (plots)

Définition : Une **parcelle** est un groupe de blocs, sur lesquels le Héro (R) peut se trouver, ayant la même coordonnée y tel que tout bloc de la parcelle est accessible depuis tout autre bloc de la parcelle en ne passant que par des blocs de la parcelle.

Sont éligibles à faire partie d'une parcelle les blocs suivants :

- Vide (.),
- échelle (L),
- Bonus (B).

Dans la suite de cette sous-partie, on parlera de bloc éligible pour se référer aux blocs d'un de ces 3 types.

Fait partie d'une parcelle tout bloc respectant l'une des conditions suivantes :

- Est (une échelle (L) ou (un bloc éligible directement sous un Câble (C))) adjacent¹ à (une échelle (L) ou (un bloc éligible directement sous un Câble (C))). Les parcelles formées à partir de blocs remplissant cette condition sont les parcelles de câble (plot_cable).
- Est (un bloc éligible au dessus d'(un Sol (F) ou une échelle (L)) adjacent¹ à (un bloc éligible au dessus d'(un Sol (F) ou une échelle (L))).

Dans l'implémentation en C de cette stratégie, on utilise une matrice **plots** de taille `ysize*xsize` représentant le plateau telle que $\forall (y, x) \in [0, ysize - 1] \times [0, xsize - 1]$:

$plots[y][x] = \begin{cases} -1 & \text{si le bloc correspondant n'appartient pas à une parcelle} \\ i & \text{entier positif identifiant de manière unique sa parcelle d'appartenance sinon} \end{cases}$

On construit aussi un tableau **plots_y_position** permettant, une fois construit, d'accéder à la hauteur de n'importe quelle parcelle facilement et en $O(1)$.

Exemple : Sur la carte `level0.map`, les parcelles sont définies comme suit :

```

0          1          2          3
01234567890123456789012345678901234
+++++
+ [red bar] = +1
+*****=***** = +2
+ = = +3
+ = [green bar] = # +4
+ = **= *****=** +5
+ = **= = +6
+ [blue bar] = **= [blue bar] # = +7
+***=***** *****=*****+8
+ = = +9
+ [green bar] = +10
+*****=*****= +11
+ = = +12
+ [yellow bar] # = [yellow bar] # +13
+ =***** *****= +14
+ [pink bar] = [pink bar] # = +15
+*****+16
+++++17

```

Parcelles :

0 1 2 3 4 5 6

Les parcelles 1 et 5 sont les 2 seules parcelles de câble de cet exemple.

Il y a `nb_plots` parcelles différentes (ici, `nb_plots = 7`).

¹ L'adjacence mentionnée dans cette section ne se réfère qu'à l'adjacence latérale ie 2 blocs sont adjacents si et seulement si ils ont la même coordonnée y et une coordonnée x différente de exactement 1.

Ladders

Définition : Une **Échelle (Objet)** est un groupe de blocs, sur lesquels le Héro (R) peut se trouver, ayant la même coordonnée x tel que tout bloc de l'Échelle (Objet) est accessible depuis tout autre bloc de l'Échelle (Objet) en ne passant que par des blocs de l'Échelle (Objet).

Sont éligibles à faire partie d'une Échelle (Objet) les blocs suivants :

- Vide (.),
- échelle (L),
- Bonus (B),
- Sortie (X).

Dans la suite de cette sous-partie, on parlera de bloc éligible pour se référer aux blocs d'un de ces 3 types.

Fait partie d'une Échelle (Objet) tout bloc respectant l'une des conditions suivantes :

- Est une échelle (L) ou la Sortie (X).
- Est (du Vide (.) ou un Bonus (B)) au dessus de (la Sortie (X) ou une échelle (L)).

Dans l'implémentation en C de cette stratégie, on utilise une matrice `ladders` de taille `ysize*ysize` représentant le plateau telle que $\forall (y, x) \in [0, ysize - 1] \times [0, xsize - 1]$:

<code>ladders[y][x] =</code>	-1 si le bloc correspondant n'appartient pas à une Échelle (Objet) i entier positif identifiant de manière unique son Échelle (Objet) d'appartenance sinon
------------------------------	---

On construit aussi un tableau `ladders_x_position` permettant, une fois construit, d'accéder à la coordonnée x de n'importe quelle Échelle (Objet) facilement et en $O(1)$.

Exemple : Sur la carte level0.map, les Échelles (Objet) sont définies comme suit :

The diagram is a 16x17 grid with rows and columns indexed from 0 to 16. The grid contains various patterns of asterisks, vertical bars, and horizontal lines, color-coded to represent different data or connections. The patterns are as follows:

- Row 0:** All cells contain a plus sign (+).
- Row 1:** All cells contain a plus sign (+).
- Row 2:** All cells contain a plus sign (+).
- Row 3:** All cells contain a plus sign (+).
- Row 4:** All cells contain a plus sign (+).
- Row 5:** All cells contain a plus sign (+).
- Row 6:** All cells contain a plus sign (+).
- Row 7:** All cells contain a plus sign (+).
- Row 8:** All cells contain a plus sign (+).
- Row 9:** All cells contain a plus sign (+).
- Row 10:** All cells contain a plus sign (+).
- Row 11:** All cells contain a plus sign (+).
- Row 12:** All cells contain a plus sign (+).
- Row 13:** All cells contain a plus sign (+).
- Row 14:** All cells contain a plus sign (+).
- Row 15:** All cells contain a plus sign (+).
- Row 16:** All cells contain a plus sign (+).

The patterns are color-coded and distributed as follows:

- Green:** Vertical bars in columns 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17.
- Red:** Vertical bars in columns 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17.
- Blue:** Vertical bars in columns 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17.
- Orange:** Vertical bars in columns 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17.
- Purple:** Vertical bars in columns 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17.
- Yellow:** Vertical bars in columns 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17.

The diagram represents a complex system of connections or data flow between the 17 rows and 17 columns.

Échelles (Objet):



Il y a nb_ladders Échelles (Objet)
différentes (ici, nb_ladders = 9)

III.3. Le graphe d'accessibilité (`plots_accessibility_graph`)

Définition : Le **graphe d'accessibilité** $G = (S, A)$ est un graphe orienté pondéré tel que :

- $S = [0, nb_plots - 1]$ représente les différentes parcelles,
- A représente les accessibilités entre les différentes parcelles ie il existe un arc de $plot1 \in S$ vers $plot2 \in S$ si et seulement si $plot2$ est accessible depuis $plot1$, sans passer par aucune autre parcelle. Ces accessibilités peuvent être dues à des Échelles (Objet), auquel cas on a symétrie de la relation d'accessibilité (ie $plot1$ accessible depuis $plot2$ si et seulement si $plot2$ accessible depuis $plot1$), ou à des chutes, auquel cas on n'a pas symétrie de la relation ($plot2$ peut être accessible depuis $plot1$ sans que $plot1$ ne soit accessible depuis $plot2$). C'est à cause des accessibilités dues aux chutes que G est orienté.
- La fonction de pondération `weight` fonctionne comme suit :

`weight(plot1, plot2)` = $\begin{cases} \text{INT_MAX (équivalent informatique de l'infini, pertinent pour le parcours grâce à l'algorithme de Dijkstra) si } plot2 \text{ n'est pas accessible depuis } plot1 \\ \text{La plus courte distance euclidienne d'un chemin permettant d'aller d'une case de } plot1 \text{ à une case de } plot2 \end{cases}$

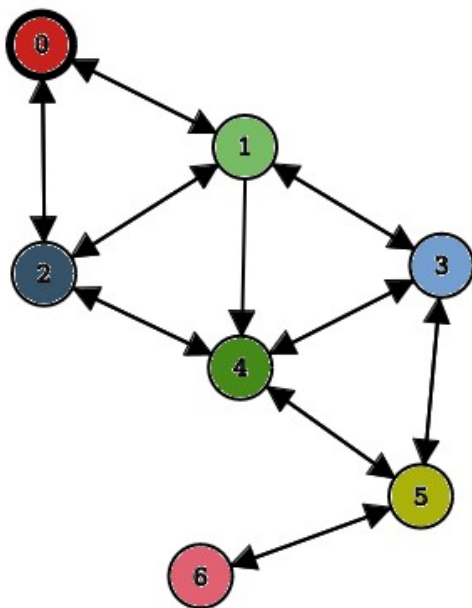
Pour l'implémentation en C de ce graphe, on utilise une matrice de liste d'entiers `plots_accessibility_graph` ie un objet du type `int_list***`.

C'est une représentation par matrice d'adjacence suivant les règles suivantes :

$\forall (plot1, plot2) \in [0, nb_plots - 1]^2$:

`plots_accessibility_graph[plot1][plot2]` = $\begin{cases} \text{La liste vide si } plot2 \text{ n'est pas accessible depuis } plot1 \\ \text{Toutes les coordonnées x des accessibilités de } plot1 \text{ à } plot2 \text{ sinon} \end{cases}$

Exemple : Graphe d'accessibilités de `level0.map` :



Graphe d'accessibilité, représentation humainement parlante

```
(Départ = 0, Arrivée = 0) :  
(Départ = 0, Arrivée = 1) : 18 9  
(Départ = 0, Arrivée = 2) : 9  
(Départ = 0, Arrivée = 3) :  
(Départ = 0, Arrivée = 4) :  
(Départ = 0, Arrivée = 5) :  
(Départ = 0, Arrivée = 6) :  
  
(Départ = 1, Arrivée = 0) : 9  
(Départ = 1, Arrivée = 1) :  
(Départ = 1, Arrivée = 2) : 8 9  
(Départ = 1, Arrivée = 3) : 20 19 18 31 17  
(Départ = 1, Arrivée = 4) : 14 13 12 11 10  
(Départ = 1, Arrivée = 5) :  
(Départ = 1, Arrivée = 6) :  
  
(Départ = 2, Arrivée = 0) : 9  
(Départ = 2, Arrivée = 1) : 9  
(Départ = 2, Arrivée = 2) :  
(Départ = 2, Arrivée = 3) :  
(Départ = 2, Arrivée = 4) : 10 4  
(Départ = 2, Arrivée = 5) :  
(Départ = 2, Arrivée = 6) :
```

Graphe d'accessibilité, représentation par matrice d'adjacence avec liste d'entiers (partie 1/2)

```

(Départ = 3, Arrivée = 0) :
(Départ = 3, Arrivée = 1) : 31 17
(Départ = 3, Arrivée = 2) :
(Départ = 3, Arrivée = 3) :
(Départ = 3, Arrivée = 4) : 24
(Départ = 3, Arrivée = 5) : 24
(Départ = 3, Arrivée = 6) :

(Départ = 4, Arrivée = 0) :
(Départ = 4, Arrivée = 1) :
(Départ = 4, Arrivée = 2) : 4
(Départ = 4, Arrivée = 3) : 24
(Départ = 4, Arrivée = 4) :
(Départ = 4, Arrivée = 5) : 25 11 24
(Départ = 4, Arrivée = 6) :

(Départ = 5, Arrivée = 0) :
(Départ = 5, Arrivée = 1) :
(Départ = 5, Arrivée = 2) :
(Départ = 5, Arrivée = 3) : 24
(Départ = 5, Arrivée = 4) : 11 24
(Départ = 5, Arrivée = 5) :
(Départ = 5, Arrivée = 6) : 5 23 22 21 20 19 18 17 16 15 14 33 6

(Départ = 6, Arrivée = 0) :
(Départ = 6, Arrivée = 1) :
(Départ = 6, Arrivée = 2) :
(Départ = 6, Arrivée = 3) :
(Départ = 6, Arrivée = 4) :
(Départ = 6, Arrivée = 5) : 33 6
(Départ = 6, Arrivée = 6) :

```

Graphe d'accessibilité, représentation par matrice d'adjacence avec liste d'entiers (partie 2/2)

Mise à jour du graphe d'accessibilité au vu des ennemis :

Une fois le graphe d'accessibilité construit comme décrit précédemment, on le met à jour de manière à ce qu'il s'adapte à la présence d'ennemis. Pour ce faire, on supprime toutes les accessibilités émanant d'Échelles (Objet) sur lesquelles se trouvent des ennemis, car on considère qu'elles ne sont pas empruntables.

Remarque : On ne supprime pas les accessibilités émanant de chutes lorsqu'il y a un ennemi sur le chemin car il est possible de se décaler pendant une chute, et donc d'éviter ces potentiels ennemis.

Pour faciliter cette mise à jour du graphe d'accessibilité, on construit une matrice `enemies` dont le fonctionnement est analogue à celui de `plots` et de `ladders` ie $\forall (y, x) \in [0, ysize - 1] \times [0, xsize - 1]$:

$$enemies[y][x] = \begin{cases} -1 & \text{s'il n'y a pas d'ennemi sur le bloc correspondant} \\ 0 & \text{sinon} \end{cases}$$

IV. Grandes lignes de la stratégie

L'objectif est de déterminer le meilleur déplacement à effectuer, au vu de l'état courant du plateau.

Pour ce faire, la **première étape** consiste à déterminer notre case objectif, ie la case sur laquelle on essaye de se rendre. Cette case sera :

- La case du Bonus (B) le plus proche du Héro (calculé à l'aide d'une fonction `closest_bonus`, qui utilise la distance euclidienne) dans le cas où il reste des bonus à récupérer.
- La case de la Sortie (X) dans le cas où tous les bonus ont déjà été récupérés.

La **deuxième étape**, plus complexe, consiste à déterminer le meilleur déplacement à effectuer afin d'atteindre cette case objectif, tout en respectant les contraintes liées à la construction du plateau, aux ennemis et aux bombes.

Pour cette deuxième étape, on considérera 4 cas différents, chacun agrémenté de sous-cas qui permettent une première prise de décision de déplacement. Avant de choisir un déplacement, on s'assure que celui-ci est possible grâce à la fonction `is possible action`. S'il n'est pas possible, d'autres déplacements sont essayés un à un. Ces déplacements de secours ne sont pas précisés ici pour des soucis de lisibilité.

Cas 1 : La case objectif correspond à la Sortie (X) et le Héro est proche (cf sous-cas pour plus de précision) de la sortie :

Cas 1.1 : Le Héro est sur la bonne parcelle pour atteindre la sortie (parcelle juste en dessous de la sortie, trouvée à l'aide de la fonction `find plot under exit`) :

Si le Héro est déjà sur l'Échelle (Objet) qui mène à la sortie, monter (ou descendre) en fonction de la position du Héro par rapport à la sortie, et sinon s'approcher de cette Échelle (Objet) (donc déplacement latéral).

Cas 1.2 : Le Héro est entre la parcelle sous la sortie et la sortie :

Le Héro est donc sur une Échelle (Objet), donc monter (ou descendre) en fonction de la position du Héro par rapport à la sortie.

Cas 1.3 : Sinon, le Héro n'est pas considéré comme proche de la Sortie (X) donc on se réfère à un des cas suivants.

Cas 2 : Le Héro se trouve sur une parcelle :

Cas 2.1 : Le Héro se trouve sur la même parcelle que la case objectif :

Se rapprocher de cette case (déplacement latéral)

Cas 2.2 : Le Héro se trouve sur une parcelle différente de la case objectif :

Trouver la position x de la première Échelle (Objet) à emprunter afin de se rendre sur la parcelle de la case objectif depuis la parcelle courante à l'aide d'une fonction `find x position first ladder`, qui utilise le `plots accessibility graph` et l'algorithme de Dijkstra. Se rapprocher de cette position x (déplacement latéral).

Cas 3 : Le Héro se trouve sur une Échelle (Objet) :

Le Héro se rapproche de la case objectif (monte ou descend).

Cas 4 : Le Héro ne se trouve ni sur une parcelle, ni sur une Échelle (Objet) :

Le Héro se rapproche de la case objectif comme possible.

Remarque : Les cas 3 et 4 représentent la limite de cette stratégie, car ils ne garantissent pas que le choix qui sera effectué sera optimal, mais une telle limite est difficilement évitable avec un raisonnement tour par tour.

Enfin, la **troisième étape** prend en compte les ennemis. Les ennemis sont pris en compte une première fois dans la mise à jour du graphe d'accessibilité, mais cela ne suffit pas à assurer que l'on ne va pas en rencontrer. C'est pourquoi cette 3^e étape adapte le choix de déplacement effectué lors de la 2^e étape en fonction des menaces potentielles. Pour rappel, une menace est un ennemi à une distance euclidienne inférieure ou égale à 2 du Héro.

Pour cette troisième étape, on commence donc par compter le nombre de menace à l'aide de la fonction `count threats to runner`. Si ce nombre est 0, cette 3^e étape s'arrête ici, sinon elle continue avec les 3 cas suivants. Encore une fois, avant de choisir un déplacement, on s'assure que celui-ci est possible grâce à la fonction `is possible action`. S'il n'est pas possible, d'autres déplacements sont essayés un à un. Ces déplacements de secours ne sont pas précisés ici pour des soucis de lisibilité.

Cas 1 : Il y a un ennemi directement accolé au Héro (distance de 1) :

Aller dans la direction opposée à cet ennemi.

Cas 2 : Il y a un ennemi à une distance de 2 du Héro, pas en diagonal :

Cas 2.1 : L'ennemi est 2 cases à droite ou à gauche du Héro :

Placer une bombe face à cet ennemi.

Cas 2.2 : L'ennemi est à 2 cases en haut ou en bas du Héro :

Aller dans la direction opposée à l'ennemi.

Cas 3 : Il y a un ennemi à une distance de 2 du Héro, en diagonal :

S'écarter de cet ennemi (déplacement latéral essayé en premier, puis déplacement haut-bas).

V. Points clés de la stratégie

V.1. Vérification de la légalité d'une action

Fonction `is_possible_action` en booléen

//Renvoie true si l'action a est possible à partir de la position (x, y) dans level et false sinon

//On considère comme impossible le fait de foncer sur un ennemi

Déclarations

Paramètres level en levelinfo, a en action, x en entier, y en entier, enemies en matrice d'entier, bombs en matrice d'entier (bombs a un fonctionnement analogue à plots, ladder et enemies et repère les bombes avec un 0, alors que les autres cases sont repérés par un -1)

Variables ok en booléen

Début

```
    ok ← faux ;
    Si a = NONE :
        ok ← vrai ;
    Fin Si ;
    Si (a = UP et ((level.map[y][x] = LADDER ou level.map[y][x] = EXIT) et enemies[y-1][x] = -1)) :
        ok ← vrai;
    Fin Si ;
    Si (a = DOWN et ((level.map[y + 1][x] = LADDER ou level.map[y + 1][x] = EXIT ou level.map[y + 1][x] = PATH ou level.map[y + 1][x] = CABLE) et enemies[y+1][x] = -1)) :
        ok ← vrai;
    Fin Si ;
    Si (a = LEFT et (((level.map[y][x - 1] != WALL et level.map[y][x - 1] != FLOOR) et enemies[y][x-1] = -1) et bombs[y+1][x-1] = -1)) :
        ok ← true;
    Fin Si ;
    Si (a = RIGHT et (((level.map[y][x + 1] != WALL et level.map[y][x + 1] != FLOOR) et enemies[y][x+1] = -1) et bombs[y+1][x+1] = -1)) :
        ok ← vrai;
    Fin Si ;
    Si (a = BOMB_LEFT et (level.map[y + 1][x - 1] = FLOOR et level.map[y][x - 1] = PATH)) :
        ok ← vrai;
    Fin Si ;
    Si (a = BOMB_RIGHT et (level.map[y + 1][x + 1] = FLOOR et level.map[y][x + 1] = PATH)) :
        ok ← vrai;
    Fin Si ;
    Renvoyer ok;
```

Fin

On choisit une fonction et non une procédure ici car on n'a besoin d'une unique valeur de retour (ok). Les paramètres sont passés par adresse et non par valeur car ils ne doivent pas être modifiés (fonction sans effet de bord).

V.2. Construction du graphe d'accessibilité

Procédure **build_plots_accessibility_graph**

//Remplit la matrice d'adjacence plots_accessibility_graph, représentant le graphe d'accessibilité des parcelles, de taille nb_plots * nb_plots

Déclarations

Paramètres level en levelinfo, plots_accessibility_graph en matrice de liste d'entiers, nb_ladders en entier, plots en matrice d'entier, ladders en matrice d'entier, ladders_x_position en tableau d'entier

Variables current_ladder en entier, y en entier, plot1 en entier, y2 en entier, x en entier, higher_plot en entier, y_next_plot en entier, lower_plot en entier

Début

//Adjacences liées aux Échelles (Objet) :

Pour current_ladder allant de 0 à nb_ladders – 1 :

Pour y allant de 1 à level.ysize – 1 :

//Inutile de parcourir les bordures car on sait que ce ne sont pas des parcelles, d'où le choix de bornes

Si ladders[y][ladders_x_position[current_ladder]] = current_ladder :

//Cas où on a atteint l'Échelle (Objet) current_ladder dans notre descente du plateau sur la colonne où elle se trouve

Si plots[y][ladders_x_position[current_ladder]] != -1 :

//Cas où le bloc courant est l'intersection entre current_ladder et une parcelle

//Cette parcelle est donc accessible depuis toutes les autres parcelles traversées par current_ladder

plot1 ← plots[y][ladders_x_position[current_ladder]] ;

Pour y2 allant de y + 1 à level.ysize – 1 :

Si (plots[y2][ladders_x_position[current_ladder]] != -1) et (ladders[y2][ladders_x_position[current_ladder]] = current_ladder) :

//Cas où on rencontre une deuxième parcelle et on est toujours sur l'Échelle (Objet)

Ajouter ladders_x_position[current_ladder] aux listes plots_accessibility_graph[plot1][plot2] et plots_accessibility_graph[plot2][plot1] ;

Fin Si ;

Fin Pour ;

Fin Si ;

Fin Si ;

Fin Pour ;

Fin Pour ;

//Adjacences liées aux chutes :

//Chutes par tombée de Câble (C) :

Pour y allant de 1 à level.ysize – 1 :

Pour x allant de 1 à level.xsize – 1 :

Si level.map[y][x] = CABLE :

higher_plot ← plots[y+1][x] ;

Si level.map[y+2][x] != FLOOR :

y_next_plot ← y+2 ;

```

Tant que plots[y_next_plot][x] = -1 et y_next_plot < le-
vel.ysize - 1 :
    //On descend sous le cable (parcelle haute) jusqu'à tom-
ber sur une autre parcelle (parcelle basse) ou jusqu'à atteindre la fin du plateau
    y_next_plot ← y_next_plot + 1 ;
    Fin Tant que ;
    Si plots[y_next_plot][x] != - 1 :
        //Cas où on est sortis du while parce qu'une parcelle a
été atteinte
        lower_plot ← plots[y_next_plot][x] ;
        Ajouter x à la liste plots_accessibility_graph[hi-
gher_plot][lower_plot] ;
    Fin Si ;
    Fin Si ;
    Fin Si ;
    Fin Pour ;
    Fin Pour ;

//Chutes pas sortie de parcelle (déplacement à gauche ou à droite lorsque position-
né sur un dernier bloc de parcelle) :
Pour x allant de 2 à level.xsize - 2 :
    //Cas où on peut tomber à gauche de la parcelle en (y, x) :
    Si plots[y][x - 1] = -1 et plots[y][x] != -1 et level.map[y][x - 1] != FLOOR :
        higher_plot ← plots[y][x] ;
        y_next_plot ← y ;
        Tant que plots[y_next_plot][x - 1] = -1 et y_next_plot < level.ysize - 1 :
            //On descend la colonne à gauche de la parcelle haute jusqu'à tomber
sur une autre parcelle (parcelle basse) ou jusqu'à atteindre la fin du plateau
            y_next_plot ← y_next_plot + 1 ;
        Fin Tant que ;
        Si plots[y_next_plot][x-1] != -1 :
            //Cas où on est sortis du while parce qu'une parcelle a été atteinte
            lower_plot ← plots[y_next_plot][x-1] ;
            Ajouter x-1 à plots_accessibility_graph[higher_plot][lower_plot] ;
        Fin Si ;
    Fin Si ;

//Cas où on peut tomber à droite de la parcelle en (y, x) :
Si plots[y][x + 1] = -1 et plots[y][x] != -1 et level.map[y][x + 1] != FLOOR :
    higher_plot ← plots[y][x] ;
    y_next_plot ← y ;
    Tant que plots[y_next_plot][x + 1] = -1 et y_next_plot < level.ysize - 1 :
        //On descend la colonne à gauche de la parcelle haute jusqu'à tomber
sur une autre parcelle (parcelle basse) ou jusqu'à atteindre la fin du plateau
        y_next_plot ← y_next_plot + 1 ;
    Fin Tant que ;
    Si plots[y_next_plot][x+1] != -1 :
        //Cas où on est sortis du while parce qu'une parcelle a été atteinte
        lower_plot ← plots[y_next_plot][x+1] ;

```



```

                                Ajouter    x+1    à    plots_accessibility_graph[higher_plot]
[lower_plot] ;
                                Fin Si ;
                                Fin Si ;
                                Fin Pour ;

Fin

```

On choisit une procédure et non une fonction car on veut initialiser un objet du type `int_list***` et C ne peut pas renvoyer plus d'un élément. Par construction du type `int_list*`, donner une liste en paramètre revient à donner un pointeur vers le premier élément de cette liste donc une liste de ce type ne peut être passée que par adresse. Ici, la matrice de liste d'entier `plots_accessibility_graph` est donc forcément passée par adresse, avec toutes les listes modifiables, ce qui correspond à ce qu'on veut ici car on souhaite remplir le graphe d'accessibilité grâce à des effets de bord.

V.3. Calculs de meilleur chemin et de poids utile

Procédure **initialize_best_x_path_and_practical_weight**

//Initialise les valeurs de best_path et practical_weight correspondant au chemin de position1 à position2

//Si position2 non accessible depuis position1 : best_path = -1 et practical_weight = l'infini

Déclarations

Paramètres best_path en pointeur sur un entier, practical_weight en pointeur sur un entier, plots_accessibility_graph en matrice de liste d'entiers, plots en matrice d'entier, plots_y_position en tableau d'entier, x1 en entier, y1 en entier, x2 en entier, y2 en entier

Variables plot1 en entier, plot2 en entier, paths en liste d'entiers, current_path en entier, current_best_path en entier, current_best_distance en entier, current_distance en entier

Début

plot1 \leftarrow plots[y1][x1] ;

plot2 \leftarrow plots[y1][x2] ;

Si weight(plots_accessibility_graph, plots_y_position, plot1, plot2) = l'infini :

//Cas où plot2 n'est pas accessible depuis plot1

↑best_path \leftarrow -1 ;

↑practical_weight \leftarrow l'infini ;

Fin Si ;

Sinon :

//Cas où plots_accessibility_graph[plot1][plot2] n'est pas la liste vide

paths \leftarrow Copie de la liste plots_accessibility_graph[plot1][plot2] ;

current_path \leftarrow paths → data ;

current_best_path \leftarrow current_path ;

Si (current_path \geq x1 et current_path \leq x2) ou (current_path \leq x1 et current_path \geq x2) :

//Cas 1 : le chemin courant est latéralement entre la position1 et la position2 (alignement latérale (position1 | (current_path | position2) OU (position2 | current_path | position1))

current_best_distance \leftarrow |x1 - x2| ;

//current_best_distance correspond à la distance latérale minimum à parcourir pour aller en position1 à position2 en passant par current_path

Fin Si ;

Sinon si (current_path \leq x1 et x1 \leq x2) ou (current_path \geq x1 et x1 \geq x2) :

//Cas 2 : alignement latérale (current_path | position1 | position2) OU (position2 | position1 | current_path)

current_best_distance = 2*|current_best_path - x1| + |x1 - x2|;

Fin Sinon si ;

Sinon :

//Cas 3 : alignement latérale (current_path | position2 | position1) OU (position1 | position2 | current_path)

current_best_distance = 2*|current_best_path - x2| + abs|x1 - x2|;

Fin Sinon ;

```

    Tant que paths n'est pas vide :
        Si (current_path >= x1 et current_path <= x2) ou (current_path <= x1
et current_path >= x2) :
            //Cas 1
            current_distance ← |x1 - x2|;
            Fin Si ;
            Sinon si (current_path <= x1 et x1 <= x2) ou (current_path >= x1 et x1
>= x2) :
                //Cas 2
                current_distance ← 2*|current_path - x1| + |x1 - x2|;
                Fin Sinon si ;
                Sinon :
                //Cas 3
                current_distance ← 2*|current_path - x2| + |x1 - x2|;
                Fin Sinon ;
                Si current_distance < current_best_distance :
                    current_best_distance ← current_distance ;
                    current_best_path ← current_path ;
                Fin Si ;
                current_path ← paths → data ;
                Supprimer le premier élément de paths ;
            Fin Tant que ;
            ↑best_path ← current_best_path ;
            ↑practical_weight = current_best_distance + weight(plots_accessibility_graph
, plots_y_position, plot1, plot2) ;
            Fin Sinon ;
Fin

```

On choisit une procédure et non une fonction car on souhaite récupérer 2 valeurs : **best_path** et **practical_weight**, donc un passage par adresse de ces 2 paramètres est nécessaire car une fonction ne peut pas renvoyer 2 valeurs en C. Pour cette procédure, on présuppose l'existence d'une fonction **weight** qui renvoie la pondération d'un arc, comme décrite dans la modélisation (page 10).

V.4. Recherche de la position x de la première Échelle (Objet) à emprunter

Fonction **find_x_position_first_ladder** en entier

//Renvoie la coordonnée y de la première Échelle (Objet) à utiliser pour s'approcher de (x2, y2) à partir de (x1, y1)

//Présume que les positions 1 et 2 ne sont pas sur la même parcelle

//Dans le cas où plot2 n'est momentanément pas accessible depuis plot1 (possible avec les suppressions d'arcs dues aux ennemis), renvoie -1

Déclarations

Paramètres level en levelinfo, plots_accessibility_graph en matrice de liste d'entier, plots en matrice d'entier, plots_y_position en matrice d'entier, nb_plots en entier, x1 en entier, y1 en entier, x2 en entier, y2 en entier

Variables plot1 en entier, plot2 en entier, best_x_path en entier, distances en tableau d'entier, predecessors en tableau d'entier, queue en liste d'entier, new_distance en entier, first_plot_in_path en entier, current_plot en entier, x_first_plot_in_path en entier, y_first_plot_in_path en entier

Début

plot1 ← plots[y1][x1] ;

plot2 ← plots[y2][x2] ;

best_x_path ← find_best_x_path(plots_accessibility_graph, plots, plots_y_position, x1, y1, x2, y2) ;

Si best_x_path! = -1 :

//Cas où position2 est accessible à partir de position1 en passant par une unique Échelle (Objet)

Renvoyer best_x_path ;

Fin Si ;

Sinon :

//Application de l'algorithme de Dijkstra au graphe d'accessibilités pondéré par les poids utiles :

distances ← Tableau à nb_plots cases, rempli de l'infini ;

distances[plot1] ← 0 ;

predecessors ← Tableau à nb_plots cases, rempli de -1 ;

queue ← Liste d'entier vide ;

Ajouter tous les sommets de plots_accessibility_graph (entiers de 0 à nb_plots - 1) à queue ;

Tant que la queue n'est pas vide :

Extraire le sommet vertex de priorité minimale de queue (les priorités sont les distances) ;

Pour tout voisin neighbor de vertex dans plots_accessibility_graph qui est encore dans la queue :

new_distance ← distances[vertex] + weight(plots_accessibility_graph, plots_y_position, vertex, neighbor) ;

Si new_distance < distances[neighbor] :

distances[neighbor] ← new_distance ;

predecessor[neighbor] ← vertex ;

Fin Si ;

Fin Pour tout ;

Fin Tant que ;

```

//Reconstruction du chemin à partir des tableaux distances et predecessors
fournis par dijkstra :
    first_plot_in_path ← predecessors[plot2]; //Première parcelle traversée sur le
chemin de plot1 à plot2
    Si first_plot_in_path = -1 ;
    //Cas où plot2 n'est momentanément pas accessible depuis plot1
        best_x_path = -1 ;
    Fin Si ;
    Sinon :
        current_plot ← predecessors[current_plot] ;
        Tant que current_plot != plot1 :
            current_plot ← predecessors[current_plot] ;
            Si current_plot != -1 :
                first_plot_in_path ← current_plot ;
        Fin Tant que ;
        x_first_plot_in_path ← find_middle_of_plot(level, plots, plots_y_posi-
tion, first_plot_in_path) ;
        y_first_plot_in_path ← plots_y_position[first_plot_in_path] ;
        best_x_path ← find_best_x_path(plots_accessibility_graph, plots,
plots_y_position, x1, y1, x_first_plot_in_path, y_first_plot_in_path) ;
    Fin Sinon :
    Renvoyer best_x_path ;
Fin Sinon ;
Fin

```

On choisit une fonction et non une procédure ici car on n'a besoin d'une unique valeur de retour (best_x_path). Les paramètres sont passés par adresse et non par valeur car ils ne doivent pas être modifiés (fonction sans effet de bord).

Pour cette fonction, on présuppose l'existence d'une fonction `find_best_x_path` qui renvoie le meilleur chemin calculé par la procédure `initialize_best_x_path_and_practical_weight`, d'une fonction `find_vertex_minimum_distance` qui trouve le sommet de priorité minimal d'une liste d'entier, d'une fonction `is_accessible_from` qui renvoie les voisins d'un sommet passé en paramètre dans `plots_accessibility_graph`, d'une fonction `weight` qui renvoie la pondération d'un arc, comme décrite dans la modélisation (page 10) et d'une fonction `find_middle_of_plot` qui renvoie le milieu d'une parcelle.

V.5. Mise à jour du graphe d'accessibilité en tenant comptes des ennemis

Procédure **update_plots_accessibility_graph_with_enemies**

//Si un ennemi se trouve sur une Échelle (Objet), cette fonction supprime tous les arcs dûs à cette Échelle (Objet) dans plots_accessibility_graph

Déclarations

Paramètres level en levelinfo, enemies_list en liste d'entier, plots_accessibility_graph en matrice de liste d'entier, plots en matrice d'entier, ladders en matrice d'entier

Variables enemy en entier, enemy_x_position en entier, enemy_y_position en entier, ladder_enemy en entier, plots_crossed_by_ladder_enemy en liste d'entier, y en entier, current_plot_1 en entier, current_plot_2 en entier

Début

Pour tout enemy dans enemies_list :

 enemy_x_position ← get_x_position_from_identifieur(level, enemy);

 enemy_y_position ← get_y_position_from_identifieur(level, enemy);

 ladder_enemy ← ladders[enemy_y_position][enemy_x_position];

 //Échelle (Objet) sur laquelle se trouve l'ennemi courant

 Si ladder_enemy != -1 :

 //L'ennemi courant est sur une Échelle (Objet)

 plots_crossed_by_ladder_enemy ← Liste d'entier vide

 Pour tout y entre 0 et level.ysize - 1 :

 //Parcours de la ligne verticale sur laquelle se trouve ladder_enemy

(ligne de coordonnée x = enemy_x_position)

 Si ladders[y][enemy_x_position] = ladder_enemy :

 //La case courante est sur l'Échelle (Objet) de l'ennemi

 Si plots[y][enemy_x_position] != -1 :

 //La case courante est sur une parcelle

 Ajouter plots[y][enemy_x_position] à plots_crossed_by_ladder_enemy ;

sed_by_ladder_enemy ;

 Fin Si ;

 Fin Si ;

Fin Pour tout ;

 //Suppression des y dûs à ladder_enemy dans plots_accessibility_graph :

 Pour tout current_plot_1 dans plots_crossed_by_ladder_enemy :

 Pour tout current_plot_2 dans plots_crossed_by_ladder_enemy :

my : Enlever enemy_x_position de plots_accessibility_graph[current_plot_1][current_plot_2] ;

 Fin Pour tout ;

 Fin Pour tout ;

Fin Si ;

Fin Pour tout ;

Fin

On choisit une procédure et non une fonction car on ne veut rien renvoyer, simplement mettre à jour le graphe d'accessibilité `plots_accessibility_graph`.
Pour cette fonction, on présuppose l'existence d'une fonction `get_x_position_from_identifie` qui renvoie la coordonnée x d'une case à partir de son identifiant (cf vocabulaire page 6) et d'une fonction avec un comportement analogue pour la coordonnée y `get_y_position_from_identifie`.

VI. Évaluation expérimentale

Après avoir effectué quelques tests, on remarque que cette algorithme semble réussir à passer level0 dans 100 % des cas, level1 dans 70 % des cas, level2 dans 50 % des cas et level3 dans 30 % des cas.

Remarque : Ces tests n'ont pas été effectués à grande échelle.

Conclusion

La stratégie fournie consiste en un calcul de plus court chemin dans un graphe modélisant une version simplifiée du plateau, adapté à la présence d'ennemis. Elle est bonne quand il n'y a pas d'ennemis, mais elle pourrait être parfaite dans le cas où il y a des ennemis. Il aurait par exemple été possible d'apporter plus de finesse dans les cas décrits en page 12 pour adapter l'algorithme à plus de situations (différencier plusieurs situations différentes au sein des situations déjà décrites), et ainsi le rendre plus efficace.

Sources

https://fr.wikibooks.org/wiki/Structures_de_donn%C3%A9es_en_C/Les_listes_simples :
pour la structure de liste décrite en page 4

https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra : pour la description de l'algorithme de
Disjktra en page 5

https://csacademy.com/app/graph_editor/ : pour construction du graphe en page 10