IA41 RAPPORT DE PROJET

Delirium 2

Résumé

Ce rapport de projet a pour objectif de présenter la méthode de résolution appliquée à la problématique posée par le projet, à savoir faire bouger le mineur de sorte à ce qu'il réussisse le niveau.

Victor Vogel, Anthony Gussy, Benjamin Jacquot

Groupe n°3

Table des matières

Introduction

Dans le cadre de l'UV IA41, nous avons été amenés à programmer une intelligence artificielle parmi un certain nombre de projets proposés. Nous avons choisi le projet Delirium2 car le principe nous semblait intéressant. Ce principe est le suivant : nous contrôlons un mineur qui doit récolter un certain nombre de diamants pour pouvoir passer aux niveaux suivants en traversant une porte. Il a donc la possibilité de creuser des galeries pour pouvoir récupérer un diamant. Le joueur devra aussi faire attention aux monstres se baladant dans les galeries, aux rochers qui peuvent chuter si le mineur créé un chemin permettant la chute du rocher. Le mineur doit également faire attention à ne pas se bloquer lui-même en faisant tomber une pierre à un endroit condamnant une zone close.

Identifications des sous problèmes et résolution

1) Identifier les éléments nécessaires à la réussite du niveau

Pour réussir un tableau, le mineur doit arriver à la sortie. Avant d'arriver à la sortie, le mineur doit rassembler un certain nombre de diamants. Lorsque le nombre de diamant est atteint la variable CanGotoExit passe à 1, indiquant au mineur qu'il peut se diriger vers la sortie. Pour assurer la réussite du niveau, nous devons donc vérifier tout d'abord si la variable CGE est à 1, auquel cas le mineur se dirige vers la sortie, sinon il se dirige vers un diamant. Si aucun des deux n'est en vue, il va errer dans le labyrinthe.

2) Se diriger vers un élément nécessaire

Afin de se diriger vers un élément donné (que ce soit la sortie, un diamant, ou une case donnée par la fonction errer), le programme va faire appel à un algorithme A*, qui va trouver le meilleur chemin vers la position de l'élément, si tel chemin existe, et renvoyer la direction à prendre pour s'y rendre.

3) Se diriger vers une zone inconnue du labyrinthe

Lorsque le programme ne peut ni se diriger vers la sortie, ni vers un diamant, il va tenter de se diriger vers un endroit encore inexploré du labyrinthe.

4) Eviter les pièges

Pour ne pas mourir, le mineur doit éviter de toucher les monstres et ne doit pas se trouver sous une pierre ou un diamant en chute libre.

5) Enregistrer les états connus du labyrinthe

Afin de ne pas avoir à chercher à chaque fois où se trouve la sortie ou les diamants, on enregistre au fur et à mesure de la progression dans le labyrinthe les différents états de celui-ci.

6) Eviter de créer une situation bloquant le jeu

Pour ne pas se retrouver dans une situation insoluble, le mineur doit vérifier à l'avance que son mouvement n'engendrera pas de telle situation. Cependant, nous n'avons pas réussi à implémenter une telle mécanique. Une fonction existe cela dit pour éviter que le mineur ne tourne en boucle sur les mêmes positions.

Représentation des connaissances

1) Variables d'états

Pour notre algorithme A* nous utilisons une liste de liste (une liste correspond à une ligne du labyrinthe) représentant tout le labyrinthe que l'on complète au fur et à mesure de l'avancement du mineur. Avec cela, nous fournissons à la fonction des coordonnées de fin, pouvant par exemple représenter la position d'un diamant, d'une porte ou une position quelconque dans le cas où le mineur doit errer, ainsi que les coordonnées du mineur.

2) Machines à états finis

Niveau 5 : Analyser ce qui est vu par le mineur pour déceler les positions dangereuses

Niveau 4 : Mettre à jour le labyrinthe d'après ce qui est vu par le mineur et les dangers perçus

Niveau 3 : S'arrêter sur la sortie si elle a été atteinte

Niveau 2 : Se diriger vers la sortie si le compte de diamant est atteint

Niveau 1 : Se diriger vers un diamant si le compte n'est pas atteint

Niveau 0 : Errer si aucun objectif n'est visible

3) Descriptions des machines à états finis

a) Niveau 5

Les positions dangereuses sont matérialisées par les endroits où une pierre qui tombe, un diamant qui tombe, ou un monstre se trouvera au prochain mouvement du mineur. Ces positions sont insérées dans le champ de vue du mineur afin qu'elles soient évitées lors de l'appel de l'algorithme A*.

L'évitement des objets qui tombent se fait en testant la valeur supérieure de chaque case du champ de vue en vérifiant qu'il s'agisse ou non d'un objet pouvant tomber et si la case actuelle est une case vide donc permettant la chute.

Pour éviter les monstres, le champ de vision et sa taille sont envoyés à la fonction. On recherche ensuite, les coordonnées des monstres grâce au labyrinthe. On compare ces coordonnées avec les coordonnées stocké dans la variable globale. Ce test permet de vérifier si le monstre est resté immobile ou s'il a effectué un mouvement (on compare les coordonnées au temps t-1 et t). On applique l'algorithme de mouvement. Une fois cela fait, nous obtenons une liste avec la position des monstres au temps t+1. Puis nous plaçons les monstres à t+1 sur une nouvelle liste en rajoutant une zone de danger autour d'eux (zone formant un + c'est-à-dire une interdiction en haut, à gauche, à droite et en bas). Puis nous renvoyons cette nouvelle liste qui fera office de nouvelle vision du mineur.

b) Niveau 4

La liste modifiée retournée par la détection des positions dangereuses est ensuite utilisée pour mettre à jour le labyrinthe. Cette mise à jour va reconstruire une nouvelle carte du labyrinthe en parcourant à la fois l'ancienne carte et la liste des objets perçus par le mineur. Les positions ne se trouvant dans aucune de ces deux listes seront mises à -1.

c) Niveau 3

Une variable globale est utilisée afin de signaler que la fin a été atteinte. Cette variable donne la position de la fin une fois celle-ci atteinte. On compare la position actuelle du mineur avec cette variable et on vérifie que CGE est bien à 1. Si ces deux conditions sont réunies, le mineur attend que le prochain niveau se charge.

d) Niveau 2

Si la position finale n'est pas atteinte mais que CGE est à 1, le mineur va tenter de se diriger vers la sortie. Il va donc tout d'abord vérifier qu'il connait la position de la sortie en la recherchant

dans le labyrinthe. S'il connait la position de la sortie, il fera appel à l'algorithme A* afin que celui-ci lui donne l'action à effectuer pour s'en rapprocher.

e) Niveau 1

Si CGE n'est pas à 1 alors le mineur doit chercher des diamants. Il en cherche donc dans la partie du labyrinthe déjà connue et va tenter de se diriger vers celui qui est le plus proche de lui en utilisant l'algorithme A*.

f) Niveau 0

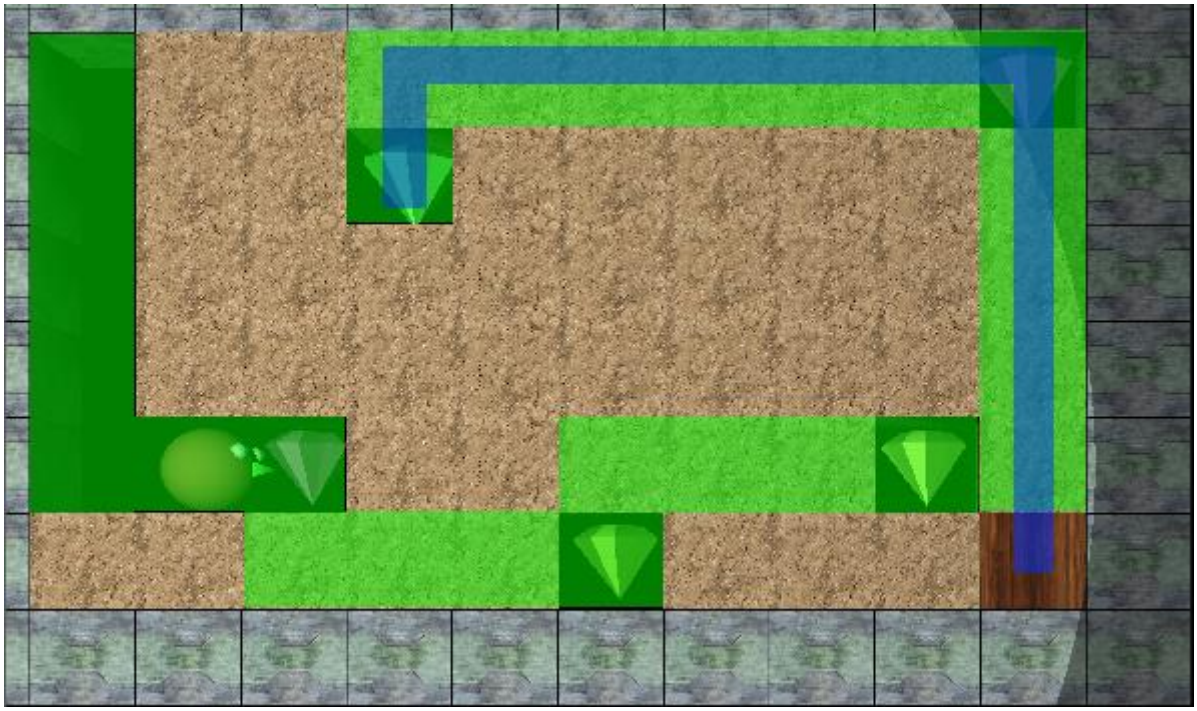
Enfin, si aucun des états précédent n'est possible, le mineur va errer dans le labyrinthe. Il va chercher la position inconnue (matérialisée par -1) la plus proche de lui. Si il n'y a pas de position inconnue dans la carte actuelle du labyrinthe, le mineur va se diriger en diagonale vers le coin inférieur droit encore inconnu de la carte.

Exemples

1) Récupération des diamants avant d'aller à la sortie

Dans cet exemple, le mineur doit faire en sorte de ramasser les diamants et d'aller à la sortie. Afin de réaliser cela, la partie connue du labyrinthe est parcourue. Chaque diamant est stocké dans une liste que l'on trie selon leur coût euclidien. Puis l'on essaye de se diriger vers le diamant le plus proche. Si le chemin est impossible, on prend le suivant et ainsi de suite jusqu'à épuisement de la liste.

Résultat obtenu :

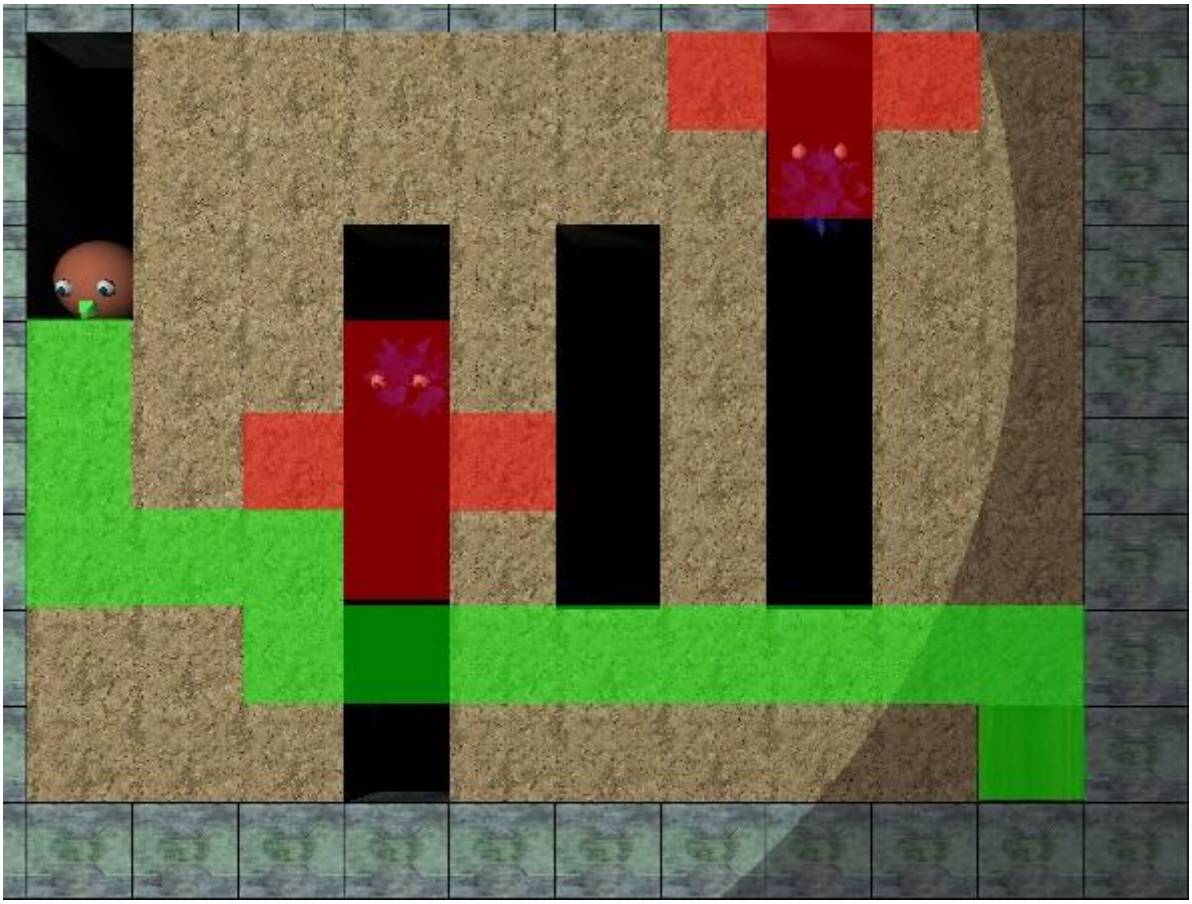


Le mineur va prendre successivement les diamants les plus proches de sa position actuelle. Une fois tous les diamants (nécessaires pour accéder à la sortie) récupérés, le mineur va revenir sur ses pas pour éviter de passer dans la clay et ainsi accéder à la sortie.

2) Éviter les monstres

Dans cet exemple, le mineur doit faire en sorte d'éviter un monstre en mouvement. Pour ce faire, nous fournissons à notre méthode la liste contenant la vision du mineur. Grâce à cette liste et une variable globale contenant la position des monstres au temps t-1, il va comparer la position au temps t-1 et t pour savoir dans quelle direction telle monstre est allé. Avec ceci, il va utiliser l'algorithme de déplacement d'un monstre pour renvoyer la prochaine action du monstre. En utilisant cette action, nous modifions la liste contenant la vision du mineur pour pouvoir ressortir une liste contenant la position du monstre au moment t+1 avec les positions autour de ce monstre (sous forme de +) comme étant dangereuses. Finalement, la liste est retournée au prédicat principal ce qui va permettre d'éviter de marcher sur la case contenant le monstre au temps t+1.

Résultat obtenu :

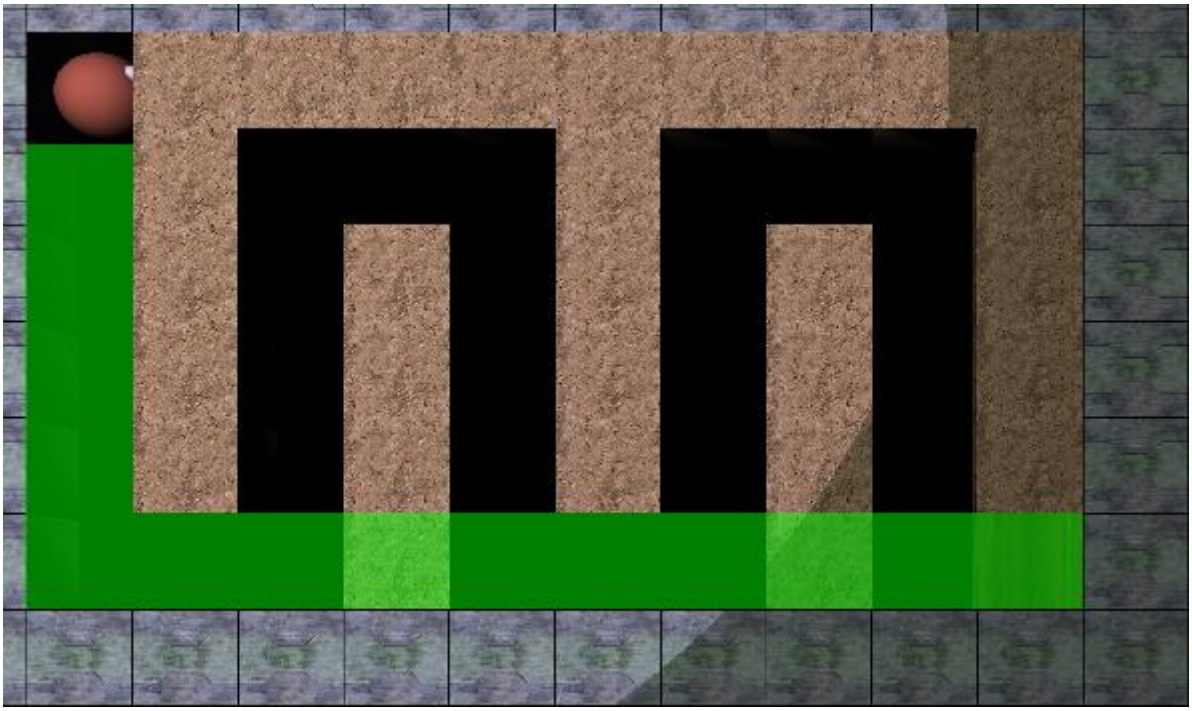


Les zones rouges représentent les zones de dangers calculées avant le trajet choisi par le mineur. Ainsi le mineur va contourner les monstres en évitant leurs zones de danger. Toutefois, comme les monstres vont se déplacer, le trajet sera recalculé tout au long de scénario pour pouvoir arriver à la sortie sans encombre.

3) Recherche d'une destination finale :

Dans cet exemple, nous devons nous déplacer jusqu'à une certaine zone. Pour ce faire, l'algorithme A* va nous aider. Nous prenons donc la position de notre mineur, le coût total pour arriver à cette case (ici 0 car nous commençons de cette case), le coût du chemin qui est égal à la distance entre la case courante et la case finale auquel nous ajoutons +3 si la case correspond à de la terre et +10 si elle correspond à une pierre, et son père (ici défini comme étant [-1,-1] car il n'a pas de père). Nous ajoutons cette information dans une variable globale notée closeList et nous cherchons les mouvements possibles. Nous vérifions que ces mouvements n'existent pas dans les variables globales open et close, si tel est le cas, nous calculons leur coût, leur heuristique et nous les ajoutons dans open. Si cela n'est pas le cas, nous vérifions que leur coût total actuel est inférieur au cout total contenu dans une des deux variables globales. Si tel est le cas, on sait que nous pouvons atteindre cette position d'une façon plus efficace, nous extrayons donc ce mouvement de la liste globale qui le contient, nous modifions ses valeurs et le rajoutons dans openList car nous devons revoir les chemins qui le succèdent car leurs coûts pourraient aussi changer. Si aucune de ces deux conditions sont réussies, nous abandonnons la position courante et passons à la suivante. Une fois cela fait, nous prenons la position ayant la meilleure valeur f (heuristique + coût) de openList et nous vérifions si la position finale est atteinte. Si cela n'est pas le cas nous recommençons les mêmes étapes que précédemment.

Résultat obtenu :

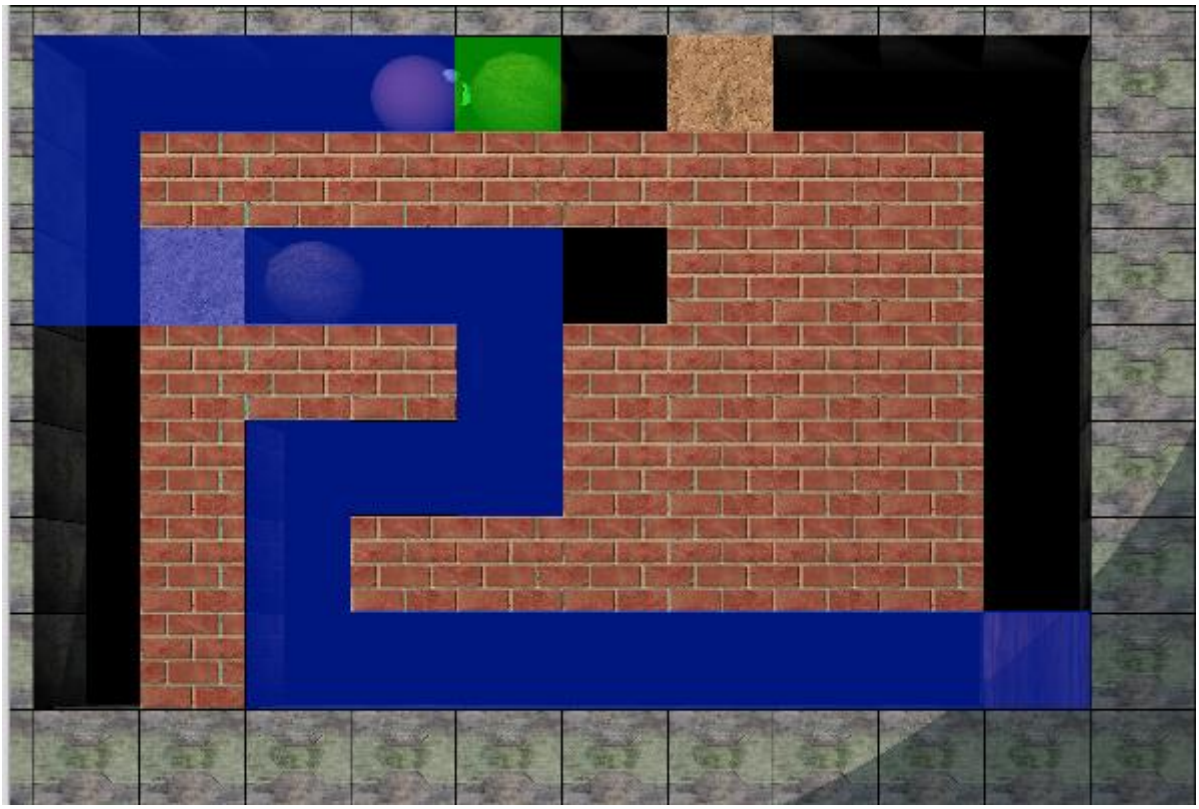


Le scénario est basique, le mineur calcule son chemin pour arriver jusqu'à la sortie, il n'évite pas la clay car le détour serai trop long.

4) Ne pas passer par les rocher bloqués

Comme vu dans l'exemple précédent, l'algorithme A* permet de trouver le chemin jusqu'à la sortie. Celui-ci accepte le passage par une pierre mais en considérant que le passage est plus « difficile », et donc que son coût est plus important. Ainsi, lorsque la pierre est suivie directement par un bloc de terre, l'algorithme considère que le chemin est faisable, mais pour un coût important. Or en réalité, ce chemin est impossible car une pierre ne peut être déplacée si elle est bloquée par « derrière ». Pour corriger ce problème, il faut identifier les pierres déplaçables et celles qui ne le sont pas. Pour ce faire, lorsqu'une pierre est un état successeur, on vérifie la direction vers laquelle arrive le mineur et ainsi on peut observer la case symétrique par rapport à la pierre, celle qui est possiblement bloquante pour la pierre, et tester si cette case est vide. Dans le cas où celle-ci est vide, le chemin est possible.

Résultat obtenu :

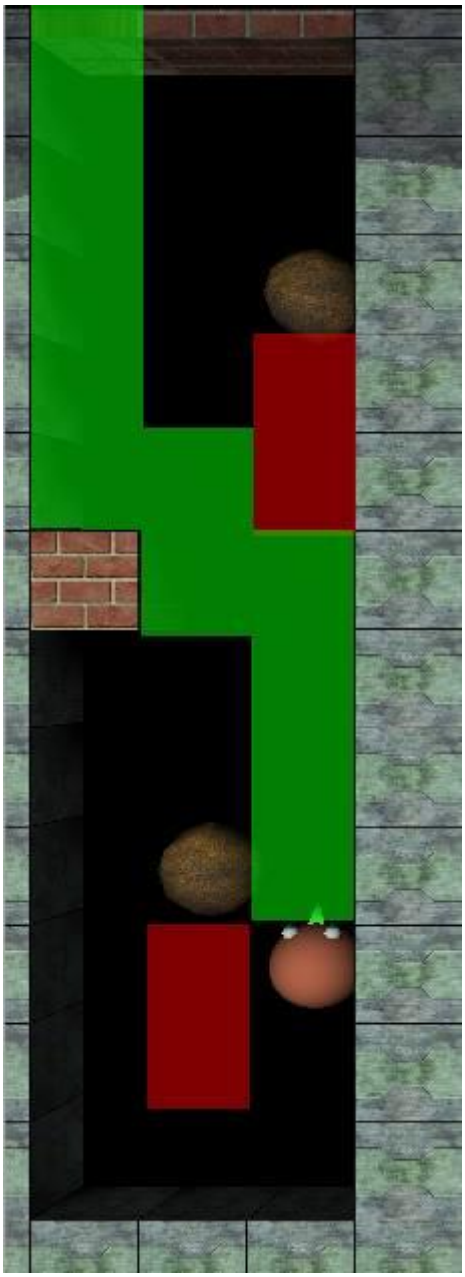


Le mineur va d'abord pousser le rocher (vert) d'une case car il se trouve entre lui et la sortie (sur le chemin le plus court). Mais une fois déplacé le rocher sera bloqué par la clay, c'est pourquoi il recalculera un autre chemin lui permettant d'accéder à la sortie. Ce second chemin est montré en bleu.

5) Éviter les objets en chute libre

La dangerosité des rochers est évaluée comme suit : si la case directement au-dessus et celle directement en dessous d'un rocher sont vides, alors le rocher va tomber. Dans ce cas les deux cases en dessous sont considérées comme dangereuses (deux car en n'en mettant qu'une, le mineur peut avancer sur la case juste en dessous du danger, danger qui va se déplacer sur le mineur. Le mineur mettant un temps pour se tourner, il serait tué par le rocher).

Résultat obtenu :



Comme pour l'exemple avec les monstres, les zones en rouge sont les zones de danger des rochers en chute libre (deux cases sous les rochers pour éviter la collision). Le mineur va donc calculer son chemin, en tenant compte des rochers, pour pouvoir accéder à la sortie. Encore une fois, la position des rochers étant mise à jour tout au long du scénario, le chemin vert ne sera pas respecté car un des rocher va faire dévier le mineur.

Difficultés rencontrées

Une des plus grandes difficultés que nous avons rencontrées était le temps de réflexion beaucoup trop important du mineur, lorsque l'on utilisait les scénarii *Boulder Dash*. Cette lenteur, toujours présente selon les cas mais dans une moindre mesure, était principalement due au fait que le choix de la direction pour un diamant ou pour errer était choisie selon le coût du chemin pour chaque diamant du labyrinthe, obtenu via l'algorithme A*. Cela nécessitait beaucoup d'appel à A* et ralentissait considérablement le jeu. Désormais, ce choix ce fait uniquement par une heuristique simplifiée donnée par la distance euclidienne entre les deux positions et A* n'est appelé qu'une fois à la fin pour vérifier que le chemin est tout de même possible.

L'autre majeure difficulté était de savoir comment errer dans le labyrinthe de la meilleure façon possible. Au départ la solution envisagée était de se diriger toujours vers la droite en longeant le mur, puis en arrivant au bout du labyrinthe, descendre d'un certain nombre de cases puis se diriger vers la gauche jusqu'au bout, et ainsi de suite. Cette solution a été abandonnée car trop coûteuse et peu efficace. La solution que nous avons implémentée utilise les positions inconnues (matérialisées par des -1) du labyrinthe. Si le labyrinthe possède des positions inconnues, le mineur va se diriger vers celle la plus proche, permettant ainsi de découvrir tout le labyrinthe petit à petit. Comme au début aucune position inconnue n'est indiquée (la carte du labyrinthe et la liste des objets perçus ayant la même taille), nous avons décidé que le mineur irait vers une position choisie arbitrairement en diagonale vers le bas et la droite, car dans cette direction se trouve le coin inférieur droit du labyrinthe, coin qui, s'il est atteint, signifie que toutes les positions du labyrinthe sont référencées.

Améliorations possibles

Une amélioration possible pour notre IA serait l'optimisation de la méthode seDirigerVers. Lorsque la position demandée est lointaine, l'heuristique devient très grande et le nombre de chemins possible aussi, ce qui rend l'algorithme assez lent. De ce fait, il faudrait trouver une solution pour élaguer des chemins ou faire en sorte de mettre de côté certains chemins (en utilisant une méthode tabou par exemple), ce qui permettrait de réduire le nombre de calculs et ainsi permettre une décision de direction plus rapide.

Annexes

```
/*
  Définition de l'IA du mineur
  Le prédicat move/12 est consulté à chaque itération du jeu.
*/

:- module( decision, [
    init/1,
    move/12
] ).
:- use_module(seDirigerVers).
:- use_module(errer).
:- use_module(eviterPieges).
:- use_module(fonctions).

% Initialiser les différentes variables globales du programme
/*
  init( _ )
*/
init(_) :- nb_setval(labyrinthe, [[-1]]), nb_setval(sortie, []), nb_setval(positions, [[-1,
-1], [-2, -2], [-3, -3], [-4, -4], [-5, -5], [-6, -6]]), nb_setval(blacklist, []),
init_astar(_), init_GlobaleMonster(_).

% Mettre à jour le labyrinthe en fonction des objets perçus, du labyrinthe précédent, des
positions dangereuses et des positions blacklistées avant de choisir le mouvement
/*
  move( +L,+LP, +X,+Y,+Pos, +Size, +CanGotoExit, +Energy,+GEnergy, +VPx,+VPy, -ActionId )
*/
move(L, _, X, Y, Pos, Size, CGE, _, _, _, _, Action) :- eviterPieges(L, X, Y, Pos, Size,
L1), nb_getval(labyrinthe, Laby), avoidLoop(_), updateLaby(Laby, L1, X, Y, Pos, Size),
nb_getval(labyrinthe, Laby1), move(X, Y, CGE, Laby1, Action), updateMovements([X, Y]).
/*
  move( +X, +Y, +CanGotoExit, +Labyrinthe, -Action )
*/
% S'arrêter sur la sortie
move(X, Y, 1, _, Action) :- nb_getval(sortie, Coord), [X, Y] = Coord, Action is 0,
write("J'attend que le prochain niveau se charge"), nl.
% Se diriger vers la sortie
move(X, Y, 1, Laby, Action) :- premiereOccurence(Laby, 21, X1, Y1), nb_setval(sortie, [X1,
Y1]), seDirigerVers([X, Y], [X1, Y1], Laby, Action), write('Je me dirige vers la sortie'), nl.
% Se diriger vers un diamant
move(X, Y, 0, Laby, Action) :- meilleureOption(Laby, X, Y, 2, Action), write('Je me dirige
vers un diamant'), nl.
% Errer
move(X, Y, _, Laby, Action) :- errer(X, Y, Laby, Action), write("J'erre"), nl.

% Mettre à jour le labyrinthe
/*
  updateLaby( +OldLaby, +L, +X, +Y, +Pos, +Size )
*/
updateLaby(OldLaby, L, X, Y, Pos, Size) :- coordLastElement(OldLaby, X1, Y1),
posLastElement(L, Pos1), updateLaby(L, X, Y, X1, Y1, Pos, Pos1, Size, OldLaby, Laby, 0, 0),
nb_setval(labyrinthe, Laby).
/*
  updateLaby( +L, +X, +Y, +X1, +Y1, +Pos, +Pos1, +Size, +OldLaby, -NewLaby, +Xe, +Ye )
*/
updateLaby(_, _, _, _, _, _, _, _, _, [[4]], Xe, Ye) :- nb_getval(blacklist, List),
member([Xe, Ye], List).
updateLaby(L, X, Y, X1, Y1, Pos, Pos1, Size, _, [[E]], Xe, Ye) :- Xe >= X1, Ye >= Y1,
posToCoord(X, Y, Pos, Size, Pos1, X2, Y2), Xe = X2, Ye = Y2, elemAtPos(L, Pos1, E).
updateLaby(_, X, Y, X1, Y1, Pos, Pos1, Size, OldLaby, [[E]], Xe, Ye) :- Xe = X1, Ye = Y1,
posToCoord(X, Y, Pos, Size, Pos1, X2, Y2), Xe >= X2, Ye >= Y2, elemAtCoord(OldLaby, X1, Y1,
E).
updateLaby(_, X, Y, X1, Y1, Pos, Pos1, Size, _, [[-1]], Xe, Ye) :- Xe >= X1, Ye >= Y1,
posToCoord(X, Y, Pos, Size, Pos1, X2, Y2), Xe >= X2, Ye >= Y2.
updateLaby(L, X, Y, X1, Y1, Pos, Pos1, Size, OldLaby, [[4]|R2], Xe, Ye) :-
nb_getval(blacklist, List), member([Xe, Ye], List), Ye1 is Ye + 1, updateLaby(L, X, Y, X1,
Y1, Pos, Pos1, Size, OldLaby, R2, 0, Ye1).
updateLaby(L, X, Y, X1, Y1, Pos, Pos1, Size, OldLaby, [[E]|R2], Xe, Ye) :- Xe >= X1,
posToCoord(X, Y, Pos, Size, Pos1, X2, Y2), Xe = X2, Ye =< Y2, posToCoord(X, Y, Pos, Size, 0,
```

```
_, Y3), Ye >= Y3, Pos2 is (Pos + (Xe - X) + Size * (Ye - Y)), elemAtPos(L, Pos2, E), Ye1 is
Ye + 1, updateLaby(L, X, Y, X1, Y1, Pos, Pos1, Size, OldLaby, R2, 0, Ye1).
updateLaby(L, X, Y, X1, Y1, Pos, Pos1, Size, OldLaby, [[E]|R2], Xe, Ye) :- Xe = X1, Ye =<
Y1, posToCoord(X, Y, Pos, Size, Pos1, X2, _), Xe >= X2, elemAtCoord(OldLaby, Xe, Ye, E), Ye1
is Ye + 1, updateLaby(L, X, Y, X1, Y1, Pos, Pos1, Size, OldLaby, R2, 0, Ye1).
updateLaby(L, X, Y, X1, Y1, Pos, Pos1, Size, OldLaby, [[-1]|R2], Xe, Ye) :- Xe >= X1,
posToCoord(X, Y, Pos, Size, Pos1, X2, _), Xe >= X2, Ye1 is Ye + 1, updateLaby(L, X, Y, X1,
Y1, Pos, Pos1, Size, OldLaby, R2, 0, Ye1).
updateLaby(L, X, Y, X1, Y1, Pos, Pos1, Size, OldLaby, [[4|R1]|R2], Xe, Ye) :-
nb_getval(blacklist, List), member([Xe, Ye], List), Xe1 is Xe + 1, updateLaby(L, X, Y, X1,
Y1, Pos, Pos1, Size, OldLaby, [R1|R2], Xe1, Ye).
updateLaby(L, X, Y, X1, Y1, Pos, Pos1, Size, OldLaby, [[E|R1]|R2], Xe, Ye) :- posToCoord(X,
Y, Pos, Size, Pos1, X2, Y2), Xe =< X2, Ye =< Y2, posToCoord(X, Y, Pos, Size, 0, X3, Y3), Xe
>= X3, Ye >= Y3, Pos2 is (Pos + (Xe - X) + Size * (Ye - Y)), elemAtPos(L, Pos2, E), Xe1 is
Xe + 1, updateLaby(L, X, Y, X1, Y1, Pos, Pos1, Size, OldLaby, [R1|R2], Xe1, Ye).
updateLaby(L, X, Y, X1, Y1, Pos, Pos1, Size, OldLaby, [[E|R1]|R2], Xe, Ye) :- Xe =< X1, Ye
=< Y1, elemAtCoord(OldLaby, Xe, Ye, E), Xe1 is Xe + 1, updateLaby(L, X, Y, X1, Y1, Pos,
Pos1, Size, OldLaby, [R1|R2], Xe1, Ye).
updateLaby(L, X, Y, X1, Y1, Pos, Pos1, Size, OldLaby, [[-1|R1]|R2], Xe, Ye) :- Xe1 is Xe +
1, updateLaby(L, X, Y, X1, Y1, Pos, Pos1, Size, OldLaby, [R1|R2], Xe1, Ye).

% Blacklister les positions causant une boucle infinie
/*
  avoidLoop( _ )
*/
avoidLoop(_) :- nb_getval(positions, [E1, E2, E1, E2, E1, E2]), nb_getval(blacklist, List),
append(List, [E1, E2], List1), nb_setval(blacklist, List1).
avoidLoop(_).

% Mettre à jour la liste des 6 dernières actions effectuées par le mineur
/*
  updateMovements( +Coord )
*/
updateMovements(Coord) :- nb_getval(positions, [E1, E2, E3, E4, E5, _]),
nb_setval(positions, [Coord, E1, E2, E3, E4, E5]).
```

```
:- module( eviterPieges, [
    eviterPieges/6,
    init_GlobaleMonster/1
] ).

:- use_module(fonctions).

% Initialiser la variable globale utilisée pour stocker les positions des monstres
/*
  init_GlobaleMonster( _ )
*/
init_GlobaleMonster(_):- nb_setval(posMonstre,[]).

% Ajoute les zones dangereuses à la liste des objets perçus
/*
  eviterPieges( +L, +X, +Y, +Pos, +Size, -L2 )
*/
eviterPieges(L, X, Y, Pos, Size, L2) :- eviterPieges(L, L, 0, Size, L1), eviterMonstres(L1, X,
Y, Pos, Size, L2).
/*
  eviterPieges( +L, +L1, +Pos, +Size, -L2 )
*/
eviterPieges(_, [], _, _, []).
eviterPieges(L, [0|R1], Pos, Size, [4|R2]) :- Pos1 is Pos - Size, elemAtPos(L, Pos1, 3), Pos2
is Pos + 1, eviterPieges(L, R1, Pos2, Size, R2).
eviterPieges(L, [0|R1], Pos, Size, [4|R2]) :- Pos1 is Pos - Size, elemAtPos(L, Pos1, 0), Pos2
is Pos1 - Size, elemAtPos(L, Pos2, 3), Pos3 is Pos + 1, eviterPieges(L, R1, Pos3, Size, R2).
eviterPieges(L, [0|R1], Pos, Size, [4|R2]) :- Pos1 is Pos - Size, elemAtPos(L, Pos1, 2), Pos2
is Pos + 1, eviterPieges(L, R1, Pos2, Size, R2).
eviterPieges(L, [0|R1], Pos, Size, [4|R2]) :- Pos1 is Pos - Size, elemAtPos(L, Pos1, 0), Pos2
is Pos1 - Size, elemAtPos(L, Pos2, 2), Pos3 is Pos + 1, eviterPieges(L, R1, Pos3, Size, R2).
eviterPieges(L, [E|R1], Pos, Size, [E|R2]) :- Pos1 is Pos + 1, eviterPieges(L, R1, Pos1, Size,
R2).

/*
  eviterMonstres(+Map,+X,+Y,+PosMineur, +Size,-NewMap)
  Méthode permettant de récupérer une nouvelleListe contenant les monstres à l'état t+1
*/
eviterMonstres(Map,X,Y,PosMineur, Size,NewMap):-
  recupPosMonstre(Map,0,PosMonstres),
  coordonneeMonstre(PosMonstres,X,Y,PosMineur,Size,CoordMonstres),
  lancerRecupDirection(CoordMonstres,PosMonstres,ListeD),
  nouvellePosition(Map,Size,ListeD,ListeP),
  modifCarte(Map,ListeP,NewMap).

/*
  lancerRecupDirection(+CoordMonstres,+PosMonstres,-ListeDirectionPrecedent)
  Méthode permettant de récupérer une liste contenant la direction précédente ainsi que la
  position du monstre lié à cette direction
*/
lancerRecupDirection(CoordMonstres,PosMonstres,ListeDirectionPrecedent):-
  nb_getval(posMonstre,ListeCoordMonstre),
  verifMonstre(CoordMonstres,PosMonstres,ListeDirectionPrecedent,ListeCoordMonstre,[]).

/*
  coordonneeMonstre(+PosMonstres,+X,+Y,+PosMineur,+Size,-CoordMonstres)
  Méthode permettant de récupérer les coordonnée X,Y du monstre
*/

coordonneeMonstre([],_,_,_,_,[]).
coordonneeMonstre([Pos1|Reste],X,Y,Pos,Size,ListeCord):-
    posToCoord(X, Y, Pos, Size, Pos1, X1, Y1),
    !,
    append([[X1,Y1]],NewList,ListeCord),
    coordonneeMonstre(Reste,X,Y,Pos,Size,NewList).
```

```
/*
  nouvellePosition(+Map,+Size,+ListeD,-ListeP)
  Méthode permettant de récupérer la position du monstre à l'état t+1
*/

nouvellePosition(_,_,[],[]).
nouvellePosition(Map,Size,[Val|Reste],Liste):-
    Val = [5,Pos],
    Pos1 is Pos+1,
    ajouteCroix(Pos1,Pos,Size,Pos2),
    Pos3 is Pos+Size,
    Pos4 is Pos-1,
    ajouteCroix(Pos4,Pos,Size,Pos5),
    Pos6 is Pos-Size,
    Pos7 is Pos2+Size,
    Pos8 is Pos2-Size,
    Pos9 is Pos5+Size,
    Pos10 is Pos5-Size,

    append([[Pos,Pos2],[Pos,Pos3],[Pos,Pos5],[Pos,Pos6],[Pos,Pos7],[Pos,Pos8],[Pos,Pos9],[Pos,Pos
    10]],NewListe,Liste),
    !,
    nouvellePosition(Map,Size,Reste,NewListe).
nouvellePosition(Map,Size,[Val|Reste],Liste):-
    Val = [D,Pos],
    moveMonster(Map, Pos, Size, D, Pos1, Pos2, Pos3, Pos4),
    append([[Pos,Pos1],[Pos,Pos2],[Pos,Pos3],[Pos,Pos4]],NewListe,Liste),
    !,
    nouvellePosition(Map,Size,Reste,NewListe).

nouvellePosition(Map,Size,[_|Reste],Liste):-
    nouvellePosition(Map,Size,Reste,Liste).

/*
 ajouteCroix(PosEtat1,PosInitiale,Size,NewPos)
 Vérifie que la position droite et gauche sont possible (si le monstre est tout à droite du
 labyrinthe il ne peut pas
 avoir une position = position +1).

*/
ajouteCroix(Pos,_,Size,Pos1):-
    Pos > -1,
    Mod is mod(Pos,Size),
    Mod \=0,
    Pos1 = Pos.
ajouteCroix(_,PosInitiale,_,PosInitiale).

/*
  recupPosMonstre(+Map,+Pos,-PosMonstres)
  Renvoie la position du monstre
*/
recupPosMonstre([],_,[]).
recupPosMonstre([X|R],Nb,[Nb|R2]):-
    X > 23,
    Nb1 is Nb+1,
    !,
    recupPosMonstre(R,Nb1,R2).
recupPosMonstre([_|R],Nb,PosMonstres):-
    Nb1 is Nb+1,
    recupPosMonstre(R,Nb1,PosMonstres).
/*
  verifMonstre(+CoordMonstre,+PosMonstre,-ListeD,-ListCoordMonstre)
  Méthode permettant de parcourir toute les positions des monstres et de récupérer leur
  direction ainsi que set
  la variable globale posMonstre au position courante.
```

```
*/

verifMonstre([],[],[],_,PosMonstre):-
  flushPosMonstre(PosMonstre,NewPosMonstre),
  nb_setval(posMonstre,NewPosMonstre).
verifMonstre([Monstre|AutresMonstre],[PosMonstre|Reste],ListeD,ListCoordMonstre,NewListe):-
  addPosition(ListCoordMonstre,Monstre,PosMonstre,NewCoord,Direction),
  append([Direction],NewListeD,ListeD),
  !,
  append([NewCoord],NewListe,NewListCoordMonstre),
  verifMonstre(AutresMonstre,Reste,NewListeD,ListCoordMonstre,NewListCoordMonstre).
verifMonstre([_|AutresMonstre],[_|Reste],ListeD,ListCoordMonstre,NewListe):-
  verifMonstre(AutresMonstre,Reste,ListeD,ListCoordMonstre,NewListe).

replace([_|T], 0, X, [X|T]).
replace([H|T], I, X, [H|R]):-
    I > -1, NI is I-1, replace(T, NI, X, R), !.

/*
  modifCarte(+Map,+ListeD,-NewMap)
  Remplace les positions courante par une case vide et celle au temps t+1 par un monstre.
*/
modifCarte(N,[],N):-!.
modifCarte(Map,[X|Reste],NewMap):-
    X = [XAvant,XApres],
    replace(Map,XApres,24,MapSuivante),
    !,
    modifCarte(MapSuivante,Reste,NewMap).
modifCarte(Map,[_|Reste],NewMap):-
    modifCarte(Map,Reste,NewMap).


/*
  flushPosMonstre(+PosMonstre,-NewPosMonstre)
  Méthode permettant de récupérer les coordonnées des monstres au temps t (permet la maj de la
  variable globale posMonstre).
*/

flushPosMonstre([],[]).
flushPosMonstre([Monstre|AutresMonstre],PosMonstre):-
    Monstre = [_,Pos2],
    append([Pos2],NewPosMonstre,PosMonstre),
    !,
    flushPosMonstre(AutresMonstre,NewPosMonstre).
  flushPosMonstre([_|AutresMonstre],PosMonstre):-
    flushPosMonstre(AutresMonstre,PosMonstre).

/*
  addPosition(+CoordMonstres,+CoordMonstreCourant,+Pos,-NewPos,-D)
  Donne la direction du monstre en fonction des coordonnée de la variable globale et des
  coordonnées du monstre courant.
  Il donne aussi une liste contenant la position précédente et courante.
*/

% droite = 1 , haut = 2, gauche = 3, bas = 4
addPosition([],[],_,[],[]).
addPosition([],X,_,NewPos,[]):-
  append([X,X],[],NewPos).
addPosition(CoordMonstres,X,Pos,NewPos,D):-
  member(X,CoordMonstres),
  D = [5,Pos],
  !,
  append([X,X],[],NewPos).
addPosition([PosAvant|_],Coord,Pos,NewPos,D):-
  PosAvant = [XA,YA],
```

```
  Coord = [X,Y],
  X1 is X+1,
  YA = Y,
  XA = X1,
  D = [3,Pos],
  !,
  append([PosAvant,[X,Y]],[],NewPos).

addPosition([PosAvant|_],Coord,Pos,NewPos,D):-
  PosAvant = [XA,YA],
  Coord = [X,Y],
  X1 is X-1,
  YA = Y,
  XA = X1,
  D = [1,Pos],
  !,
  append([PosAvant,Coord],[],NewPos).

addPosition([PosAvant|_],Coord,Pos,NewPos,D):-
  Coord = [X,Y],
  PosAvant = [XA,YA],
  Y1 is Y+1,
  YA = Y1,
  XA = X,
  D = [2,Pos],
  !,
  append([PosAvant,Coord],[],NewPos).

addPosition([PosAvant|_],Coord,Pos,NewPos,D):-
  Coord = [X,Y],
  PosAvant = [XA,YA],
  Y1 is Y-1,
  YA = Y1,
  XA = X,
  D = [4,Pos],
  !,
  append([PosAvant,Coord],[],NewPos).

addPosition([_|Reste],Coord,Pos,NewPos,D):-
  addPosition(Reste,Coord,Pos,NewPos,D).


% Prochaine position possible pour monstre
possibleMoveMonster(L, Pos) :-
    elemAtPos(L, Pos, E),
    E = 0.

/*
  moveMonster(+L, +Pos, +Size, +D, -Pos1)
  donne la position au temps t+1 en fonction de la direction du monstre
*/

% f(d)
% Vers la gauche
moveMonster(L, Pos, Size, D, Pos1, Pos2, Pos4, Pos5) :-
    D = 2,
    Pos1 is Pos - 1,
    Pos1 > -1,
    Mod is mod(Pos1,Size),
    Mod \=0,
    possibleMoveMonster(L, Pos1),
    !,
    Pos2 is Pos1+Size,
    Pos3 is Pos1-1,
    ajouteCroix(Pos3,Pos1,Size,Pos4),
```

```
    Pos5 is Pos1-Size.
% Vers le bas
moveMonster(L, Pos, Size, D, Pos1, Pos2, Pos4, Pos6) :-
    D = 3,
    Pos1 is Pos + Size,
    possibleMoveMonster(L, Pos1),
    !,
    Pos2 is Pos1+Size,
    Pos3 is Pos1-1,
    ajouteCroix(Pos3,Pos1,Size,Pos4),
    Pos5 is Pos1+1,
    ajouteCroix(Pos5,Pos1,Size,Pos6).
% Vers la droite
moveMonster(L, Pos, Size, D, Pos1, Pos2, Pos4, Pos5) :-
    D = 4,
    Pos1 is Pos + 1,
    Mod is mod(Pos1,Size),
    Mod \=0,
    possibleMoveMonster(L, Pos1),
    !,
    Pos2 is Pos1+Size,
    Pos3 is Pos1+1,
    ajouteCroix(Pos3,Pos1,Size,Pos4),
    Pos5 is Pos1-Size.
% Vers le haut
moveMonster(L, Pos, Size, D, Pos1, Pos3, Pos5, Pos6) :-
    D = 1,
    Pos1 is Pos - Size,
    possibleMoveMonster(L, Pos1),
    !,
    Pos2 is Pos1-1,
    ajouteCroix(Pos2,Pos1,Size,Pos3),
    Pos4 is Pos1+1,
    ajouteCroix(Pos4,Pos1,Size,Pos5),
    Pos6 is Pos1-Size.
% d
% Vers le haut
moveMonster(L, Pos, Size, D, Pos1, Pos3, Pos5, Pos6) :-
    D = 2,
    Pos1 is Pos - Size,
    possibleMoveMonster(L, Pos1),
    !,
    Pos2 is Pos1-1,
    ajouteCroix(Pos2,Pos1,Size,Pos3),
    Pos4 is Pos1+1,
    ajouteCroix(Pos4,Pos1,Size,Pos5),
    Pos6 is Pos1-Size.
% Vers la gauche
moveMonster(L, Pos, Size, D, Pos1, Pos2, Pos4, Pos5) :-
    D = 3,
    Pos1 is Pos - 1,
    Pos1 > -1,
    Mod is mod(Pos1,Size),
    Mod \=0,
    possibleMoveMonster(L, Pos1),
    !,
    Pos2 is Pos1+Size,
    Pos3 is Pos1-1,
    ajouteCroix(Pos3,Pos1,Size,Pos4),
    Pos5 is Pos1-Size.
% Vers le bas
moveMonster(L, Pos, Size, D, Pos1, Pos2, Pos4, Pos6) :-
    D = 4,
    Pos1 is Pos + Size,
    possibleMoveMonster(L, Pos1),
```

```
    !,
    Pos2 is Pos1+Size,
    Pos3 is Pos1-1,
    ajouteCroix(Pos3,Pos1,Size,Pos4),
    Pos5 is Pos1+1,
    ajouteCroix(Pos5,Pos1,Size,Pos6).
% Vers la droite
moveMonster(L, Pos, Size, D, Pos1, Pos2, Pos4, Pos5) :-
    D = 1,
    Pos1 is Pos + 1,
    Mod is mod(Pos1,Size),
    Mod \=0,
    possibleMoveMonster(L, Pos1),
    !,
    Pos2 is Pos1+Size,
    Pos3 is Pos1+1,
    ajouteCroix(Pos3,Pos1,Size,Pos4),
    Pos5 is Pos1-Size.
% f(f(d))
% Vers le bas
moveMonster(L, Pos, Size, D, Pos1, Pos2, Pos4, Pos6) :-
    D = 2,
    Pos1 is Pos + Size,
    possibleMoveMonster(L, Pos1),
    !,
    Pos2 is Pos1+Size,
    Pos3 is Pos1-1,
    ajouteCroix(Pos3,Pos1,Size,Pos4),
    Pos5 is Pos1+1,
    ajouteCroix(Pos5,Pos1,Size,Pos6).
% Vers la droite
moveMonster(L, Pos, Size, D, Pos1, Pos2, Pos4, Pos5) :-
    D = 3,
    Pos1 is Pos + 1,
    Mod is mod(Pos1,Size),
    Mod \=0,
    possibleMoveMonster(L, Pos1),
    !,
    Pos2 is Pos1+Size,
    Pos3 is Pos1+1,
    ajouteCroix(Pos3,Pos1,Size,Pos4),
    Pos5 is Pos1-Size.
% Vers le haut
moveMonster(L, Pos, Size, D, Pos1, Pos3, Pos5, Pos6) :-
    D = 4,
    Pos1 is Pos - Size,
    possibleMoveMonster(L, Pos1),
    !,
    Pos2 is Pos1-1,
    ajouteCroix(Pos2,Pos1,Size,Pos3),
    Pos4 is Pos1+1,
    ajouteCroix(Pos4,Pos1,Size,Pos5),
    Pos6 is Pos1-Size.
% Vers la gauche
moveMonster(L, Pos, Size, D, Pos1, Pos2, Pos4, Pos5) :-
    D = 1,
    Pos1 is Pos - 1,
    Pos1 > -1,
    Mod is mod(Pos1,Size),
    Mod \=0,
    possibleMoveMonster(L, Pos1),
    !,
    Pos2 is Pos1+Size,
    Pos3 is Pos1-1,
    ajouteCroix(Pos3,Pos1,Size,Pos4),
```

```
    Pos5 is Pos1-Size.
% f(f(f(d)))
% Vers la droite
moveMonster(L, Pos, Size, D, Pos1, Pos2, Pos4, Pos5) :-
    D = 2,
    Pos1 is Pos + 1,
    Mod is mod(Pos1,Size),
    Mod \=0,
    possibleMoveMonster(L, Pos1),
    !,
    Pos2 is Pos1+Size,
    Pos3 is Pos1+1,
    ajouteCroix(Pos3,Pos1,Size,Pos4),
    Pos5 is Pos1-Size.
% Vers le haut
moveMonster(L, Pos, Size, D, Pos1, Pos3, Pos5, Pos6) :-
    D = 3,
    Pos1 is Pos - Size,
    possibleMoveMonster(L, Pos1),
    !,
    Pos2 is Pos1-1,
    ajouteCroix(Pos2,Pos1,Size,Pos3),
    Pos4 is Pos1+1,
    ajouteCroix(Pos4,Pos1,Size,Pos5),
    Pos6 is Pos1-Size.
% Vers la gauche
moveMonster(L, Pos, Size, D, Pos1, Pos2, Pos4, Pos5) :-
    D = 4,
    Pos1 is Pos - 1,
    Pos1 > -1,
    Mod is mod(Pos1,Size),
    Mod \=0,
    possibleMoveMonster(L, Pos1),
    !,
    Pos2 is Pos1+Size,
    Pos3 is Pos1-1,
    ajouteCroix(Pos3,Pos1,Size,Pos4),
    Pos5 is Pos1-Size.
% Vers le bas
moveMonster(L, Pos, Size, D, Pos1, Pos2, Pos4, Pos6) :-
    D = 1,
    Pos1 is Pos + Size,
    possibleMoveMonster(L, Pos1),
    !,
    Pos2 is Pos1+Size,
    Pos3 is Pos1-1,
    ajouteCroix(Pos3,Pos1,Size,Pos4),
    Pos5 is Pos1+1,
    ajouteCroix(Pos5,Pos1,Size,Pos6).
```

```
:- module( seDirigerVers, [
    init_astar/1,
    seDirigerVers/4,
    getHeuristicValue/3,
    possibleMove/3
] ).

:- use_module(fonctions).

seDirigerVers(Coordonnee,Fin,Laby,Action):-
    initialisationGlobale(Coordonnee,Fin),
    a_star(Coordonnee,Fin,Laby,Path),
    !,
    getSuite(Coordonnee,Path,Suite),
    getAction(Coordonnee,Suite,Action).


getSuite(Depart,[Suite,Depart,[-1,-1]],Suite):- !.
getSuite(Depart,[_|Reste],Suite):- getSuite(Depart,Reste,Suite).

getAction([X,_],[XSuite,_],Action):-
    XSuite is X + 1,
    Action = 1.
getAction([X,_],[XSuite,_],Action):-
    XSuite is X - 1,
    Action = 3.
getAction([_,Y],[_,YSuite],Action):-
    YSuite is Y + 1,
    Action = 4.
getAction([_,Y],[_,YSuite],Action):-
    YSuite is Y - 1,
    Action = 2.


% Init variables globales
init_astar(_):-
    nb_setval(openList,[]),
    nb_setval(closeList,[]).

initialisationGlobale(Coordonnee,Fin):-
    getHeuristicValue(Coordonnee,Fin,H),
    nb_setval(openList,[[Coordonnee,0,H,[-1,-1]]]),
    nb_setval(closeList,[]).

% Predicats A*
/*
  a_star(+CourantState, +FinalState, +Labyrinth, -Path)
*/
% openList vide pas de solution
a_star(_,_,_,_,_):-
    nb_getval(openList, []),
    !,
    fail.

% etat final atteint par -1
a_star([X,Y],_,Laby,Path):-
    elemAtCoord(Laby, X, Y, E),
    E = -1,
    buildPath([X,Y],Path).

% etat final atteint
a_star([X,Y],[X,Y],_,Path):-
    buildPath([X,Y],Path).

a_star([X,Y],[XFinal,YFinal],Laby,Path):-
```

```
    extractBestNodeFromOpenList(Node),
    addNodeToClose(Node),
    Node = [[X,Y],G,_,_],
    trouverSuccesseurs([X,Y],Laby,Successeurs),
    ajouterChemin([X,Y],Successeurs,[XFinal,YFinal],G,Laby),
    getBestNodeFromOpenList(BestNode),
    BestNode = [NewState,_,_,_],
    a_star(NewState,[XFinal,YFinal],Laby,Path).

ajouterChemin(_,[],_,_,_):-
    !.
ajouterChemin([X,Y],[[]|Reste],Fin,G,Laby):-
    !,
    ajouterChemin([X,Y],Reste,Fin,G,Laby).

% test si il n'est pas dans closed et open
ajouterChemin([X,Y],[Successeur|Reste],Fin,G,Laby):-
    not(isInOpen(Successeur)),
    not(isInClose(Successeur)),
    !,
    getHeuristicValue(Successeur,Fin,H),
    getCValue(C, Successeur, Laby),
    GPlus is (G + C),
    %GPlus is (G+1),
    F is (GPlus + H),
    NewNode = [Successeur,GPlus,F,[X,Y]],
    addNodeToOpen(NewNode),
    ajouterChemin([X,Y],Reste,Fin,G,Laby).

% test si g(y) > g(N.e)+c(N.e,y)
ajouterChemin([X,Y],[Successeur|Reste],Fin,G,_):-
    testBestCostInOpenOrClose([X,Y],Successeur,G,Fin),
    !,
    ajouterChemin([X,Y],Reste,Fin,G,_).

% aucun des deux critères n'est possible, on passe au successeur suivant
ajouterChemin([X,Y],[_|Reste],Fin,G,_):-
    ajouterChemin([X,Y],Reste,Fin,G,_).

% 10 de cout pour un rocher
getCValue(C,[X,Y],Laby):-
    elemAtCoord(Laby,X,Y,3),
    !,
    C is 10.
% 3 de cout pour de la dirt
getCValue(C,[X,Y],Laby):-
    elemAtCoord(Laby,X,Y,1),
    !,
    C is 3.
% 1 pour le reste
getCValue(C,_,_):-
    !,
    C is 1.

% trouver les successeur
trouverSuccesseurs([X,Y],Laby,Successeurs):-
    movementH([X,Y],Laby,CoordH),
    movementG([X,Y],Laby,CoordG),
    movementB([X,Y],Laby,CoordB),
    movementD([X,Y],Laby,CoordD),
    Successeurs = [CoordD,CoordH,CoordG,CoordB].

% verifie que le mouvement est possible
movementD([X,Y],Laby,CoordD):-
    XNew is X+1,
```

```
    possibleMove([XNew,Y],Laby,1),
    !,
    CoordD = [XNew,Y].

movementD(_,_,[]).

movementH([X,Y],Laby,CoordH):-
    YNew is Y-1,
    possibleMove([X,YNew],Laby,2),
    !,
    CoordH = [X,YNew].

movementH(_,_,[]).

movementB([X,Y],Laby,CoordB):-
    YNew is Y+1,
    possibleMove([X,YNew],Laby,4),
    !,
    CoordB = [X,YNew].

movementB(_,_,[]).

movementG([X,Y],Laby,CoordG):-
    XNew is X-1,
    possibleMove([XNew,Y],Laby,3),
    !,
    CoordG = [XNew,Y].

movementG(_,_,[]).

% Predicats A* spécifique problème
/*
  getHeuristicValue([+XATester,+YATester],[+XFinal,+YFinal],-V)
  calcul l'heuristique dans notre cas la distance euclidienne entre le point de départ et
  d'arrivé
*/
getHeuristicValue([XATester,YATester],[XFinal,YFinal],V):- V is
sqrt((XFinal-XATester)*(XFinal-XATester)+(YATester-YFinal)*(YATester-YFinal)).

/*
  extractBestNodeFromOpenList(-Node)
  recupere le noeud avec le cout le moins élevé et l'extrait (prend la valeur et l'enleve de la
  liste)
*/
%si un seul elem
extractBestNodeFromOpenList(BestNode):-
    nb_getval(openList,OpenList),
    OpenList = [BestNode|[]],
    !,
    substractFromOpenList(BestNode).

extractBestNodeFromOpenList(BestNode):-
    nb_getval(openList,OpenList),
    OpenList = [Node|ResteOpen],
    Node=[_,_,F,_],
    extractBestNodeFromOpenList(Node,BestNode,F,_,ResteOpen),
    substractFromOpenList(BestNode).

extractBestNodeFromOpenList(BestNode,BestNode,BestF,BestF,[]):-
    !.
extractBestNodeFromOpenList(_,BestNode,F,BestF,[Node|AutreNode]):-
    Node = [_,_,FNode,_],
    F > FNode,
    !,
    extractBestNodeFromOpenList(Node,BestNode,FNode,BestF,AutreNode).
```

```
extractBestNodeFromOpenList(CourNode,BestNode,F,BestF,[_|AutreNode]):-
    !,
    extractBestNodeFromOpenList(CourNode,BestNode,F,BestF,AutreNode).

/*
  getBestNodeFromOpenList(-Node)
  recupere le noeud avec le cout le moins eleve (sans l'extraire)
*/
getBestNodeFromOpenList(BestNode):-
    nb_getval(openList,OpenList),
    OpenList = [Node|ResteOpen],
    Node=[_,_,F,_],
    getBestNodeFromOpenList(Node,BestNode,F,_,ResteOpen).

getBestNodeFromOpenList(BestNode,BestNode,BestF,BestF,[]):-
    !.
getBestNodeFromOpenList(_,BestNode,F,BestF,[Node|AutreNode]):-
    Node = [_,_,FNode,_],
    F > FNode,
    !,
    getBestNodeFromOpenList(Node,BestNode,FNode,BestF,AutreNode).
getBestNodeFromOpenList(CourNode,BestNode,F,BestF,[_|AutreNode]):-
    getBestNodeFromOpenList(CourNode,BestNode,F,BestF,AutreNode).

/*
  extractNodeFromOpen(+State, -Node)
  extrait le noeud contenant les coordonnée fournit en paramètre (recupération et suppression
  de la liste)
*/
extractNodeFromOpen(State,Node):-
    nb_getval(openList,OpenList),
    extractNodeFromOpen(State,Node,OpenList),
    substractFromOpenList(Node).

extractNodeFromOpen(_,_,[]):- fail.
extractNodeFromOpen(State,Node,[Node|_]):-
    Node = [State,_,_,_].
extractNodeFromOpen(State,Node,[_|AutreNode]):-
    extractNodeFromOpen(State,Node,AutreNode).

/*
  getNodeFromOpen(+State, -Node)
  récupère le noeud contenant les coordonnée fournit en paramètre
*/
getNodeFromOpen(State,Node):-
    nb_getval(openList,OpenList),
    getNodeFromOpen(State,Node,OpenList).

getNodeFromOpen(_,_,[]):- fail.
getNodeFromOpen(State,Node,[Node|_]):-
    Node = [State,_,_,_].
getNodeFromOpen(State,Node,[_|AutreNode]):-
    getNodeFromOpen(State,Node,AutreNode).


/*
  substractFromOpenList(+Node)
  enleve la node de la liste open
*/
substractFromOpenList(Node):-
    nb_getval(openList,OpenList),
    substractFromOpenList(Node,OpenList,[]).

substractFromOpenList(Node,[Node|AutreNode],DebutNode):-
    append(DebutNode,AutreNode,NewOpenList),
```

```
    !,
    nb_setval(openList,NewOpenList).
substractFromOpenList(Node,[PasNode|AutreNode],DebutNode):-
    append(DebutNode,[PasNode],ListeNode),
    substractFromOpenList(Node,AutreNode,ListeNode).

/*
  addNodeToOpen(+Node)
  ajoute le noeud à la liste open
*/
addNodeToOpen(Node):-
    nb_getval(openList,OpenList),
    append([Node],OpenList,NewOpenList),
    nb_setval(openList,NewOpenList).

/*
  extractNodeFromClose(+State, -Node)
  extrait le noeud contenant les coordonnée fournit en paramètre
*/
extractNodeFromClose(State,Node):-
    nb_getval(closeList,CloseList),
    extractNodeFromClose(State,Node,CloseList),
    substractFromCloseList(Node).

extractNodeFromClose(_,_,[]):- fail.
extractNodeFromClose(State,Node,[Node|_]):-
    Node = [State,_,_,_].
extractNodeFromClose(State,Node,[_|AutreNode]):-
    extractNodeFromClose(State,Node,AutreNode).

/*
  getNodeFromClose(+State, -Node)
  recupere le noeud contenant les coordonnée fournit en paramètre
*/
getNodeFromClose(State,Node):-
    nb_getval(closeList,CloseList),
    getNodeFromClose(State,Node,CloseList).

getNodeFromClose(_,_,[]):- fail.
getNodeFromClose(State,Node,[Node|_]):-
    Node = [State,_,_,_].
getNodeFromClose(State,Node,[_|AutreNode]):-
    getNodeFromClose(State,Node,AutreNode).

/*
  substractFromCloseList(+Node)
  enleve la node de la liste close
*/
substractFromCloseList(Node):-
    nb_getval(closeList,CloseList),
    substractFromCloseList(Node,CloseList,[]).

substractFromCloseList(Node,[Node|AutreNode],DebutNode):-
    append(DebutNode,AutreNode,NewCloseList),
    !,
    nb_setval(closeList,NewCloseList).
substractFromCloseList(Node,[PasNode|AutreNode],DebutNode):-
    append(DebutNode,[PasNode],ListeNode),
    substractFromCloseList(Node,AutreNode,ListeNode).

/*
  addNodeToClose(+Node)
  ajoute le noeud a la liste close
*/
addNodeToClose(Node):-
```

```
    nb_getval(closeList,CloseList),
    append([Node],CloseList,NewCloseList),
    nb_setval(closeList,NewCloseList).

/*
  testBestCostInOpenOrClose(+Pere,+Successeur,+GPere,+Fin)
  vérifie que le successeur fournit en parametre existe dans une des deux liste (close ou open)
  et qu'il a un meilleur
  cout que le cout dans la liste
*/
testBestCostInOpenOrClose([X,Y],Successeur,GPere,Fin):-
    isInOpenWithBestCost(Successeur,GPere),
    !,
    extractNodeFromOpen(Successeur,_),
    GPlus is (GPere + 1),
    getHeuristicValue(Successeur,Fin,H),
    F is (GPlus + H),
    NewNode = [Successeur,GPlus,F,[X,Y]],
    addNodeToOpen(NewNode).

testBestCostInOpenOrClose([X,Y],Successeur,GPere,Fin):-
    isInCloseWithBestCost(Successeur,GPere),
    !,
    extractNodeFromClose(Successeur,_),
    GPlus is (GPere + 1),
    getHeuristicValue(Successeur,Fin,H),
    F is (GPlus + H),
    NewNode = [Successeur,GPlus,F,[X,Y]],
    addNodeToOpen(NewNode).

/*
  isInOpenWithBestCost(+Successeur,+GPere)
  regarde si le successeur est dans open avec un meilleur cout
*/
isInOpenWithBestCost(Successeur,GPere):-
    nb_getval(openList,OpenList),
    isInOpenWithBestCost(Successeur,GPere,OpenList).

isInOpenWithBestCost(_,_,[]):- fail.

isInOpenWithBestCost(Successeur,GPere,[Node|_]):-
    Node = [Successeur,G,_,_],
    GPlus is (GPere + 1),
    GPlus < G,
    !.

isInOpenWithBestCost(Successeur,GPere,[_|AutreNode]):-
    isInOpenWithBestCost(Successeur,GPere,AutreNode).



/*
  isInCloseWithBestCost(+Successeur,+GPere)
  regarde si le successeur est dans close avec un meilleur cout
*/
isInCloseWithBestCost(Successeur,GPere):-
    nb_getval(closeList,CloseList),
    isInCloseWithBestCost(Successeur,GPere,CloseList).

isInCloseWithBestCost(_,_,[]):- fail.

isInCloseWithBestCost(Successeur,GPere,[Node|_]):-
    Node = [Successeur,G,_,_],
    GPlus is (GPere + 1),
    GPlus < G,
```

```
    !.

isInCloseWithBestCost(Successeur,GPere,[_|AutreNode]):-
    isInCloseWithBestCost(Successeur,GPere,AutreNode).

/*
  buildPath(+Coordonnee,-Path,-Cout)
  construit le chemin en remontant grâce à la coordonnée fournit en paramètre
*/
buildPath([X,Y],Path):-
    buildPath([X,Y],PathSansPere,[]),
    append([[X,Y]],PathSansPere,Path).

buildPath([-1,-1],Path,Path).

buildPath([X,Y],Path,CourPath):-
    isInOpen([X,Y]),
    extractNodeFromOpen([X,Y],Node),
    Node = [[X,Y],_,_,Pere],
    append(CourPath,[Pere],NewPath),
    buildPath(Pere,Path,NewPath).

buildPath([X,Y],Path,CourPath):-
    isInClose([X,Y]),
    extractNodeFromClose([X,Y],Node),
    Node = [[X,Y],_,_,Pere],
    append(CourPath,[Pere],NewPath),
    buildPath(Pere,Path,NewPath).




/*
  isInOpen(+Coordonnee)
  vérifie que le successeur se trouve dans open
*/
isInOpen(Successeur):-
    nb_getval(openList,OpenList),
    isInOpen(Successeur,OpenList).

isInOpen(_,[]):- fail.

isInOpen(Successeur,[Node|_]):-
    Node = [Successeur,_,_,_],
    !.

isInOpen(Successeur,[_|AutreNode]):-
    isInOpen(Successeur,AutreNode).

/*
  isInClose(+Coordonnee)
  verifie que le successeur se trouve dans close
*/
isInClose(Successeur):-
    nb_getval(closeList,CloseList),
    isInClose(Successeur,CloseList).

isInClose(_,[]):- fail.

isInClose(Successeur,[Node|_]):-
    Node = [Successeur,_,_,_],
    !.

isInClose(Successeur,[_|AutreNode]):-
    isInClose(Successeur,AutreNode).
```

```
% Prochaine position possible
possibleMove([X,Y], Laby,_):- elemAtCoord(Laby,X,Y, E), E =< 2.
possibleMove([X,Y], Laby,_):- elemAtCoord(Laby,X,Y, E), E = 21.

possibleMove([X,Y],Laby,3):- elemAtCoord(Laby,X,Y,3), XGauche is (X-1),
elemAtCoord(Laby,XGauche,Y,0).
possibleMove([X,Y],Laby,2):- elemAtCoord(Laby,X,Y,3), YHaut is (Y-1),
elemAtCoord(Laby,X,YHaut,0).
possibleMove([X,Y],Laby,2):- elemAtCoord(Laby,X,Y,3), YHaut is (Y-1),
elemAtCoord(Laby,X,YHaut,4).
possibleMove([X,Y],Laby,4):- elemAtCoord(Laby,X,Y,3), YBas is (Y+1), elemAtCoord(Laby,X,YBas,0).
possibleMove([X,Y],Laby,4):- elemAtCoord(Laby,X,Y,3), YBas is (Y+1), elemAtCoord(Laby,X,YBas,4).
possibleMove([X,Y],Laby,1):- elemAtCoord(Laby,X,Y,3), XDroit is (X+1),
elemAtCoord(Laby,XDroit,Y,0).
```

```
:- module( errer, [
    errer/4
] ).

:- use_module(seDirigerVers).
:- use_module(fonctions).

% Se diriger vers un endroit inexploré ou au moins atteignable
/*
  errer( +X, +Y, +Laby, -Action )
*/
errer(X, Y, Laby, Action) :- meilleureOption(Laby, X, Y, -1, Action).
errer(X, Y, Laby, Action) :- endroitAtteignable(X, Y, Laby, Action).
errer(X, Y, Laby, Action) :- X1 is X + 1, possibleMove([X1, Y], Laby, 1), Action is 1.
errer(X, Y, Laby, Action) :- X1 is X + 1, elemAtCoord(Laby, X1, Y, 1), Action is 5.
errer(X, Y, Laby, Action) :- Y1 is Y - 1, possibleMove([X, Y1], Laby, 2), Action is 2.
errer(X, Y, Laby, Action) :- Y1 is Y - 1, elemAtCoord(Laby, X, Y1, 1), Action is 6.
errer(X, Y, Laby, Action) :- X1 is X - 1, possibleMove([X1, Y], Laby, 3), Action is 3.
errer(X, Y, Laby, Action) :- X1 is X - 1, elemAtCoord(Laby, X1, Y, 1), Action is 7.
errer(X, Y, Laby, Action) :- Y1 is Y + 1, possibleMove([X, Y1], Laby, 4), Action is 4.
errer(X, Y, Laby, Action) :- Y1 is Y + 1, elemAtCoord(Laby, X, Y1, 1), Action is 8.

% Trouver un endroit atteignable en partant du coin inférieur droit du labyrinthe
/*
  endroitAtteignable( +X, +Y, +Laby, -Action )
*/
endroitAtteignable(X, Y, Laby, Action) :- X0 is X + 2, Y0 is Y + 2, coordLastElement(Laby, XL,
YL), endroitAtteignable(X, Y, Laby, X0, Y0, 20, XL, YL, Action), !.
/*
  endroitAtteignable( +X, +Y, +Laby, +X0, +Y0, +DistMax, +XL, +YL, -Action )
*/
endroitAtteignable(X, Y, Laby, X0, Y0, _, _, _, Action) :- elemAtCoord(Laby, X0, Y0, E),
destinationPossible(E), seDirigerVers([X, Y], [X0, Y0], Laby, Action).
endroitAtteignable(X, Y, Laby, X0, Y0, DistMax, XL, YL, Action) :- X1 is X0 + 1, X1 < (X +
DistMax), X1 < XL, !, endroitAtteignable(X, Y, Laby, X1, Y0, DistMax, XL, YL, Action).
endroitAtteignable(X, Y, Laby, X0, Y0, DistMax, XL, YL, Action) :- Y1 is Y0 + 1, Y1 < (Y +
DistMax), Y1 < YL, X1 is X + 2, endroitAtteignable(X, Y, Laby, X1, Y1, DistMax, XL, YL, Action).

% Cases sur lesquelles le mineur peut se rendre
/*
  destinationPossible( +E )
*/
destinationPossible(E) :- E < 3.
```

```
:- module( fonctions, [
    meilleureOption/5,
    posToCoord/7,
    actionToCoord/5,
    elemAtPos/3,
    elemAtCoord/4,
    posLastElement/2,
    coordLastElement/3,
    premiereOccurence/4
] ).

:- use_module(seDirigerVers).

% Trouver la meilleure option possible
/*
  meilleureOption( +Laby, +X, +Y, +Elem, -Action )
*/
meilleureOption(Laby, X, Y, Elem, Action) :- triOptions(Laby, X, Y, Elem, List), !,
meilleureOption(Laby, [X, Y], List, Action).
/*
  meilleureOption( +Laby, +Coord, +CoordList, -Action )
*/
meilleureOption(Laby, Coord, [Coord1|_], Action) :- seDirigerVers(Coord, Coord1, Laby, Action).
meilleureOption(Laby, Coord, [_|R], Action) :- meilleureOption(Laby, Coord, R, Action).

% Trier les options par coût croissant
/*
  triOptions( +Laby, +X, +Y, +Elem, -List )
*/
triOptions(Laby, X, Y, Elem, List) :- occurencesElement(Laby, 0, 0, Elem, List1),
triOptions([X, Y], List1, List).
/*
  triOptions( +Coord, +L, -List )
*/
triOptions(_, [], []) :- !.
triOptions(Coord, L, [E|R]) :- moindreCout(Coord, L, 500, [-1, -1], E), retirerElement(E, L,
L1), triOptions(Coord, L1, R).

% Retirer un élément de la liste
/*
  retirerElement( +E, +L1, -L2 )
*/
retirerElement(_, [], []) :- !.
retirerElement(E, [E|R], L) :- retirerElement(E, R, L).
retirerElement(E, [X|R1], [X|R2]) :- retirerElement(E, R1, R2).

% Trouver l'élément ayant le plus faible coût de la liste
/*
  moindreCout( +Coord, +L, +Min, +Coord1, -Coord2 )
*/
moindreCout(_, [], _, Coord1, Coord1) :- not(Coord1 = [-1, -1]), !.
moindreCout(Coord, [Coord1|R], Min, _, Coord2) :- getHeuristicValue(Coord, Coord1, Cout), Cout
=< Min, moindreCout(Coord, R, Cout, Coord1, Coord2).
moindreCout(Coord, [_|R], Min, Coord1, Coord2) :- moindreCout(Coord, R, Min, Coord1, Coord2).

% Trouver toutes les occurences de l'élément dans le labyrinthe
/*
  occurencesElement( +Laby, +X, +Y, +Element, -List )
*/
occurencesElement([[E]], X, Y, E, [[X,Y]]) :- !.
occurencesElement([[_]], _, _, _, []) :- !.
occurencesElement([[]|R2], _, Y, E, L) :- Y1 is Y + 1, occurencesElement(R2, 0, Y1, E, L).
occurencesElement([[E|R1]|R2], X, Y, E, [[X, Y]|R3]) :- X1 is X + 1, occurencesElement([R1|R2],
X1, Y, E, R3).
occurencesElement([[_|R1]|R2], X, Y, E, L) :- X1 is X + 1, occurencesElement([R1|R2], X1, Y, E,
```

```
L).

% Transformer position dans la liste en coordonnées dans le labyrinthe
/*
  posToCoord( +X, +Y, +Pos, +Size, +Pos1, -X1, -Y1)
*/
posToCoord(X, Y, Pos, Size, Pos1, X1, Y1) :- X1 is (X + (Pos1 mod Size) - (Pos mod Size)), Y1
is (Y + (Pos1 // Size) - (Pos // Size)).

% Transformer une action en coordonnées d'arrivée
/*
  actionToCoord( +X, +Y, +Action, -X1, -Y1 )
*/
actionToCoord(X, Y, 1, X1, Y) :- X1 is X + 1.
actionToCoord(X, Y, 2, X, Y1) :- Y1 is Y - 1.
actionToCoord(X, Y, 3, X1, Y) :- X1 is X - 1.
actionToCoord(X, Y, 4, X, Y1) :- Y1 is Y + 1.

% Element à la position donnée
/*
  elemAtPos( +L, +Pos, -Element )
*/
elemAtPos([E|_], 0, E) :- !.
elemAtPos([_|R], Pos, E) :- Pos1 is Pos - 1, elemAtPos(R, Pos1, E).

% Element aux coordonnées données
/*
  elemAtCoord( +Laby, +X, +Y, -Element )
*/
elemAtCoord([[E|_]|_], 0, 0, E) :- !.
elemAtCoord([[_|R1]|R2], X, 0, E) :- X > 0, X1 is X - 1, elemAtCoord([R1|R2], X1, 0, E).
elemAtCoord([_|R2], X, Y, E) :- Y > 0, Y1 is Y - 1, elemAtCoord(R2, X, Y1, E).

% Position du dernier élément
/*
  posLastElement( +L, -Pos )
*/
posLastElement(L, Pos) :- posLastElement(L, 0, Pos), !.
/*
  posLastElement( +L, +Pos0, -Pos )
*/
posLastElement([_], Pos, Pos).
posLastElement([_|R], Pos0, Pos) :- Pos1 is Pos0 + 1, posLastElement(R, Pos1, Pos).

% Coordonnées du dernier élément
/*
  coordLastElement( +Laby, -X, -Y )
*/
coordLastElement(Laby, X, Y) :- coordLastElement(Laby, 0, 0, X, Y), !.
/*
  coordLastElement( +Laby, +X0, +Y0, -X, -Y )
*/
coordLastElement([[_]], X, Y, X, Y).
coordLastElement([[]|R2], _, Y0, X, Y) :- Y1 is Y0 + 1, coordLastElement(R2, 0, Y1, X, Y).
coordLastElement([[_|R1]|R2], X0, Y0, X, Y) :- X1 is X0 + 1, coordLastElement([R1|R2], X1, Y0,
X, Y).

% Trouver la première occurence d'un élément dans le labyrinthe
/*
  premiereOccurence( +Laby, +Element, -X, -Y )
*/
premiereOccurence(Laby, Element, X, Y) :- premiereOccurence(Laby, Element, 0, 0, X, Y).
/*
  premiereOccurence( +Laby, +Element, +X0, +Y0, -X, -Y )
*/
```

```
premiereOccurence([[Element|_]|_], Element, X, Y, X, Y) :- !.
premiereOccurence([[]|R], Element, _, Y0, X, Y) :- Y1 is Y0 + 1, X1 is 0, premiereOccurence(R,
Element, X1, Y1, X, Y).
premiereOccurence([[_|R1]|R2], Element, X0, Y0, X, Y) :- X1 is X0 + 1,
premiereOccurence([R1|R2], Element, X1, Y0, X, Y).
```