Hadrien BEAUFILS
Hubert LACÔTE
Julien MOINE
Guillaume STUBER

IA41

Projet : Création d'une Intelligence Artificielle pour un jeu vidéo

Delirium 2

Responsable de l'UV : Fabrice LAURI

Table des matières

Introduction

Dans le cadre de l'UV IA41, il nous est proposé de réaliser un projet. L'objectif de ce projet est de développer l'intelligence artificielle d'un jeu vidéo en langage prolog. Parmi les trois sujets proposés, nous avons personnellement choisi le jeu Delirium 2 qui nous paraissait être le sujet le plus attractif.

Le but de ce jeu est de déplacer un mineur et de lui faire ramasser un certain nombre de diamants en évitant qu'il se fasse tuer, que ce soit par les ennemis, ou par la chute de pierre ou de diamants. Une fois un nombre suffisant de diamants ramassés, notre mineur pourra se rendre à la sortie du niveau pour aller au suivant.

Pour ce faire, nous avons été amenés à modifier un prédicat exécuté par le jeu, faisant bouger le mineur. Dans ce rapport, nous allons détailler la phase de conception de l'intelligence artificielle de notre mineur. Nous ne détaillerons pas l'ensemble du code prolog, mais nous ciblerons nos explications sur les parties importantes. Vous pourrez bien entendu vous référer au code source commenté qui sera joint avec ce rapport si vous souhaitez approfondir le fonctionnement.

1. Analyse du sujet

1.1. Énoncé du problème

a) Description du jeu

Le joueur incarne un mineur (tête rose, nez vert, dans l'image ci-dessus) qui évolue en creusant dans des sous-terrains. Ceux-ci sont composés entre autres de diamants et de pierres et sont peuplés de créatures ennemis.

Le but du jeu est de récolter un certain nombre de diamants (dépendant du sous-terrain), pour pouvoir s'engouffrer vers l'accès qui mène au prochain sous-terrain. Cet accès reste bloqué tant que ce nombre n'est pas atteint.

Lorsque le mineur se déplace, il laisse derrière lui une case vide, si bien que diamant et pierre peuvent tomber en chaîne. Le mineur et les créatures explosent s'ils reçoivent un diamant ou une pierre sur la tête. Le mineur peut également exploser s'il entre en contact avec une créature.

b) Objectifs du projet

Il nous faut programmer l'IA du mineur afin qu'il puisse :

- récolter tous les diamants nécessaires à l'accès au sous-terrain suivant,
- éviter les créatures rouges si l'une d'elle vient à s'approcher trop près du mineur,
- faire exploser les créatures bleues pour récupérer les diamants qu'ils contiennent.

Notre programme devra donner une réponse satisfaisante sur un ensemble donné de sous-terrains.

1.2. Formalisation du problème

Ce projet est très intéressant puisqu'il y a énormément de fonctionnalités à implémenter. Les possibilités sont énormes puisqu'il est possible de choisir de réaliser une simple recherche de diamants, ou bien de programmer l'intelligence artificielle d'un mineur surdoué qui ne se bloquerait dans aucun labyrinthe ou encore, qui tuerait des ennemis avec plus de dextérité que le ferait un joueur humain.

Bien que notre engouement à la vue de ce sujet aurait voulu de prime abord que nous réalisions l'intelligence artificielle d'un mineur surdoué, nous nous sommes très vite rendu compte de la difficulté liée au sujet ce qui nous a amené à nous fixer des objectifs clairs qui soient atteignables.

Nous avons donc choisi que dans un premier temps, notre mineur se limiterait à ramasser les diamants présents sur la map en évitant les dangers qui pourrait le mettre en péril. Nous avons choisi que notre mineur se rendrait à la sortie aussi vite que possible et que dans un premier temps, il ne serait pas capable de réussir des niveaux complexes nécessitant le déblocage de certains passages ou l'extermination d'ennemis.

1.3. Analyse du problème

Au départ, nous avons réfléchi généralement au déplacement du mineur. Nous avons un peu analysé notre méthode de réflexion au cour du jeu afin de comprendre comment organiser les tests. Il nous a fallu comprendre comment le jeu fonctionnait, et ce dans les détails, pour qu'ensuite notre analyse ne conduise pas à des erreurs de conception.

Le premier test que nous avons étudié à été celui des restrictions de sûreté. Dans le jeu fournit, le mineur se déplaçait aléatoirement et il était frappé par des chutes de pierres ou même des monstres. Nous avons donc implémenté un test placé après la décision du mouvement permettant de le modifier si il y avait un danger immédiat ou une addition de plusieurs dangers indirects, et si tout allait bien on confirmait ce mouvement.

Ensuite, nous avons cherché à modifier le comportement primitif du mineur afin qu'il aille chercher les diamants lorsqu'ils sont une case à côté de lui. Cette phase de la réflexion a été assez simple.

Après ça, nous avons réfléchi au problème du choix des directions en général par rapport à la manière avec laquelle notre mineur devrait se diriger. Il en est ressorti que notre mineur devrait rechercher un certain nombre de diamants qui lui sont proches afin d'explorer une zone de la carte (en essayant de ne pas en oublier). Il devrait donc créer un liste de cibles dans un périmètre définit. Pour cette recherche, nous avons vite comprit qu'il faudrait incrémenter la distance de recherche en partant de 1 case d'écart jusqu'au maximum définit. Ainsi on complète la liste de cible par ordre de proximité au mineur.

A ce moment là nous savions qu'il nous faudrait une fonction de recherche du meilleur chemin et nous avons pensé que le A* était un algorithme qui correspondait bien à nos besoin. Malgré ça, le développement de cette fonction n'a pas demandé trop d'analyse car l'algorithme nous a été expliqué en cours. Nous avons tout de même compris qu'ajouter des coûts supplémentaires liés à l'environnement pouvait être bénéfiques (éviter les dangers de chute de pierre, et ne pas aller dans les cases voisines à

un ennemi).

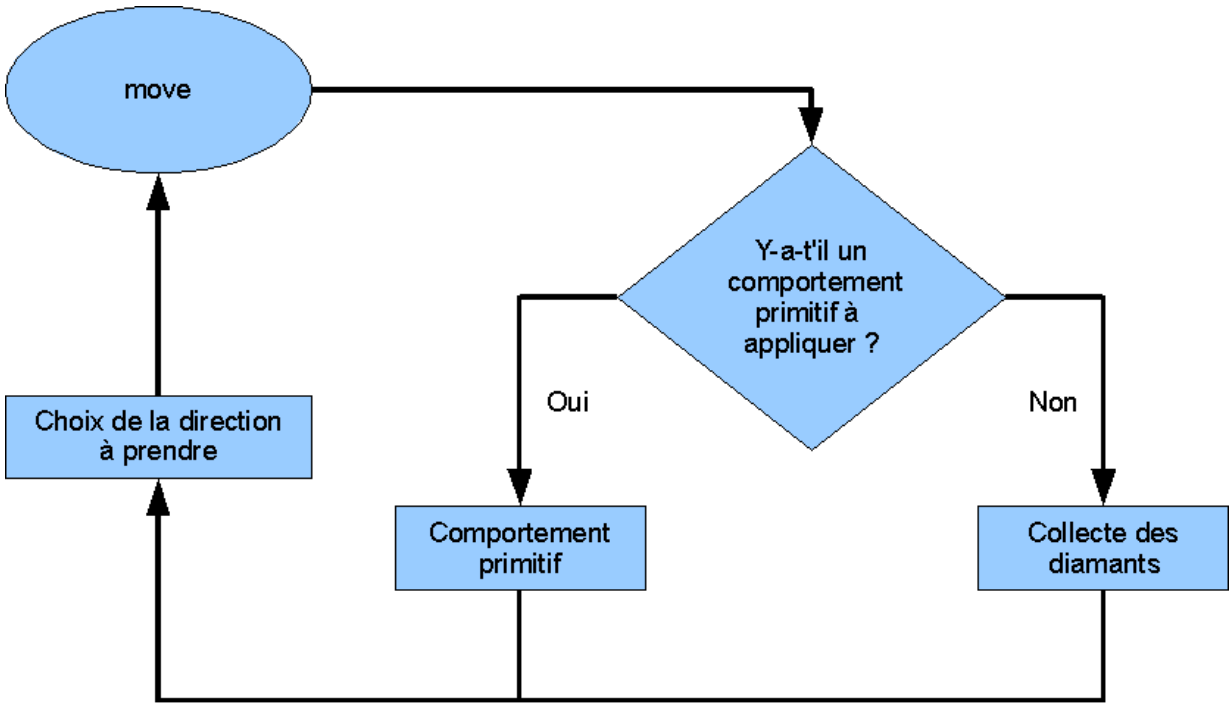
Durant la phase d'analyse on a aussi essayé de trouver comment tuer un monstre mais finalement nous avons conclu qu'il y avait beaucoup de cas à développer et que trouver une méthode générale allait s'avérer très difficile. Nous n'avons pas eu le temps de prolonger cette phase de l'analyse pour créer quelque chose de concret.

Finalement nous avons terminé notre phase d'analyse par l'étude de petits problèmes tels que la possibilité d'aller dans une direction donnée (en fonction des éléments de la carte) ou encore le calcul de la distance séparant un point d'un autre (addition des écarts sur X et Y).

2. Description du fonctionnement global

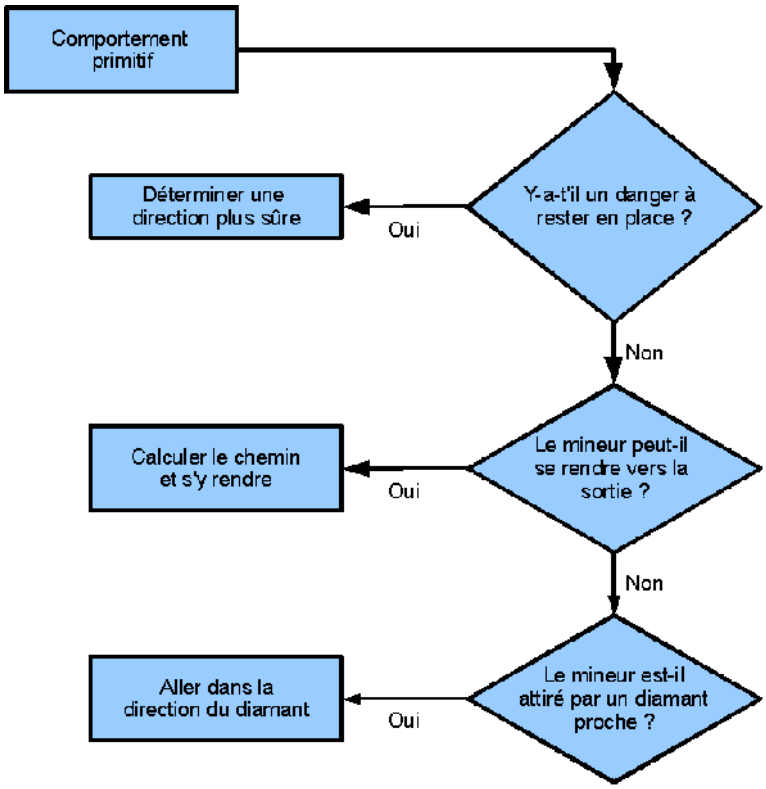
2.1. Structure générale

Pour implémenter l'intelligence artificielle du mineur, nous avons eu la possibilité de modifier le prédicat move appelé par le jeu. C'est donc ce prédicat qui va se charger de commander le mineur. Voici un schéma présentant la structure de notre prédicat move :



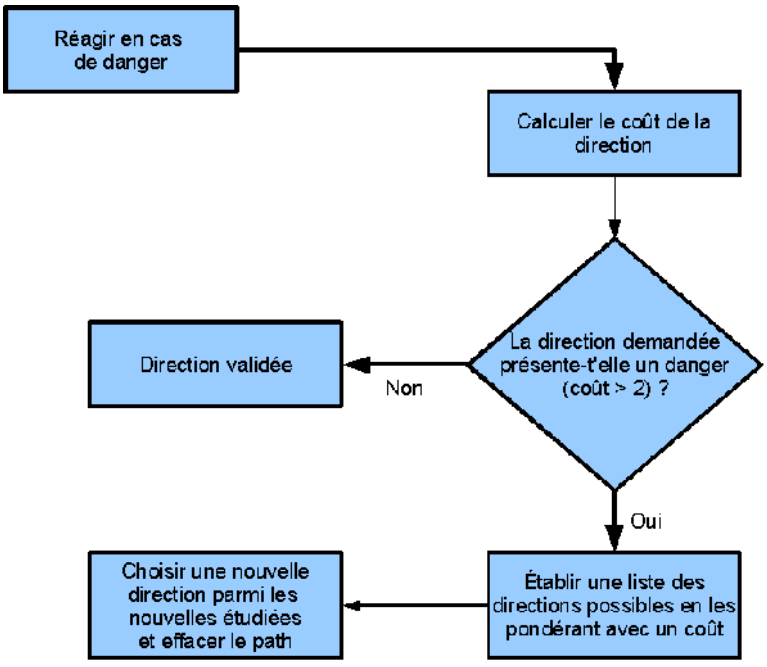
2.2. Détail du comportement primitif

Nous avons fait le choix de doter notre mineur d'un comportement primitif ce qui lui permet de réagir en premier lieu et dans un certain ordre de priorité aux comportements suivants : survie, aller vers la sortie dès que possible ou encore manger des diamants se trouvant juste à coté de lui.

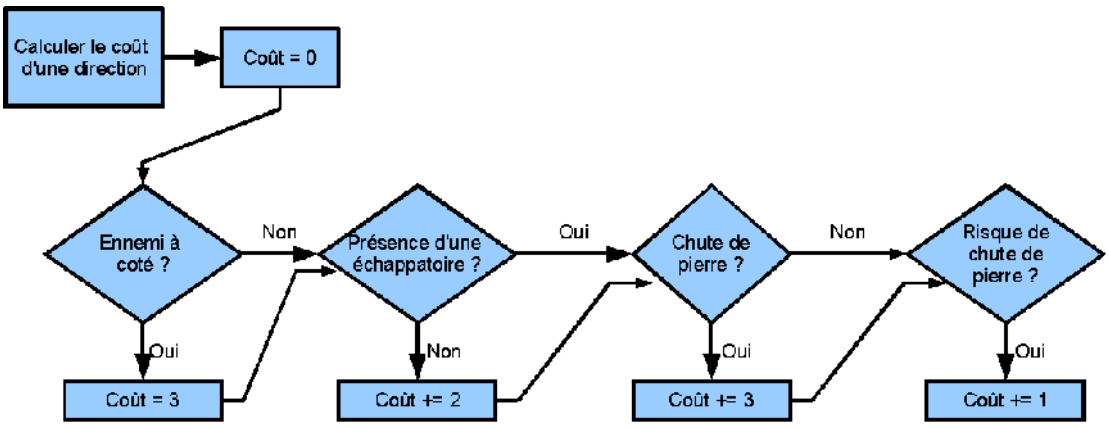


a) Analyse du danger de rester en place

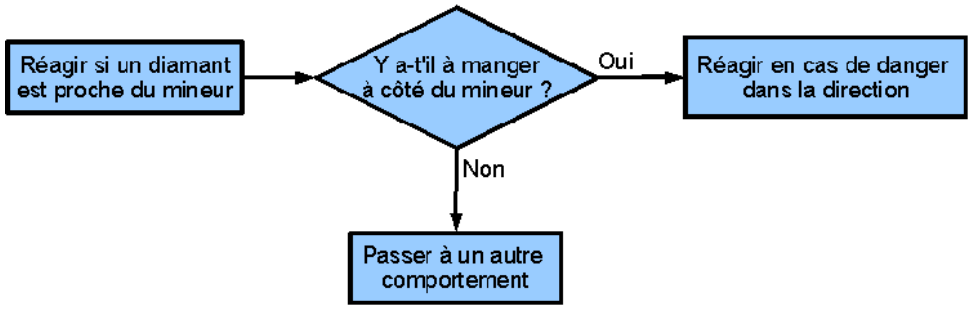
Cette analyse est faite dans le module safety_restrictions.pl grâce au prédicat gotoSafely en lui passant la valeur 4 pour direction ce qui correspond à l'envie de ne pas bouger. Ce prédicat est aussi utilisé dans la plupart des déplacements effectués par notre mineur. Ce prédicat prend tout sons sens étant donné que le niveau évolue, et que notre mineur ne peut donc pas se contenter de calculer un chemin et d'y aller les yeux fermés.



Pour calculer le coût d'une direction, nous utilisons le prédicat calculateDirCost :



b) Mineur attiré par un diamant (faim)

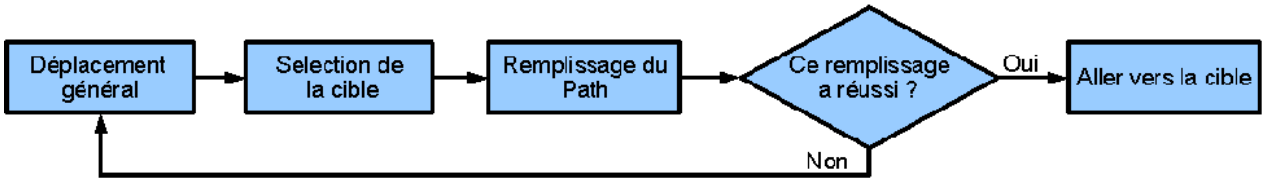


2.3. Les déplacements

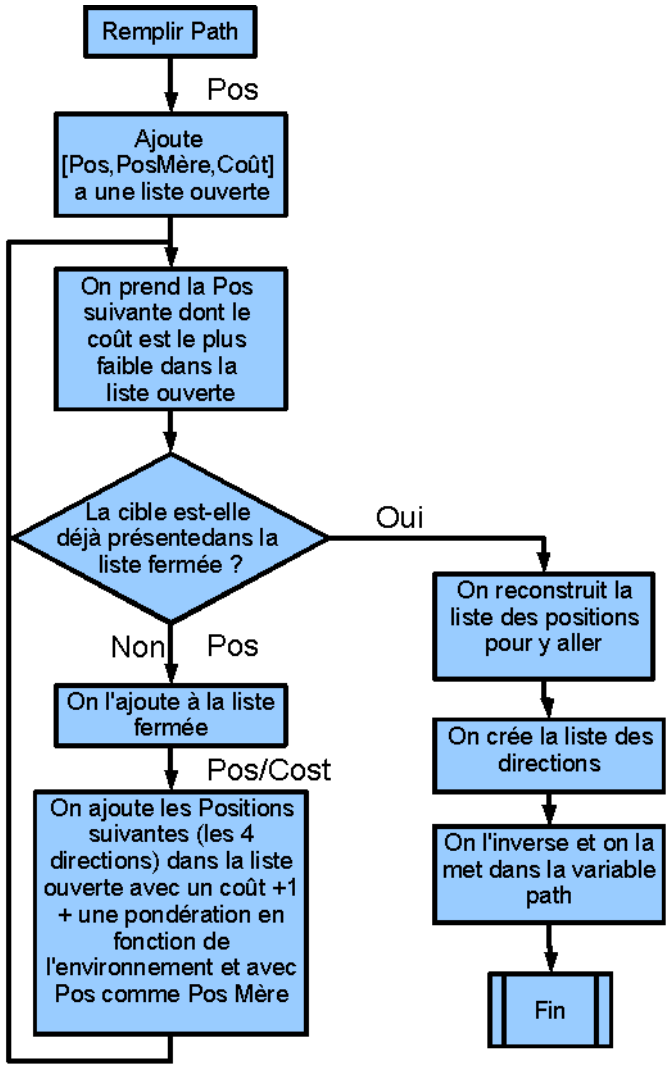
Les déplacements sont gérés de la même façon pour le comportement général « aller vers la sortie » ou pour « aller vers un diamant ».

a) Se déplacer en général

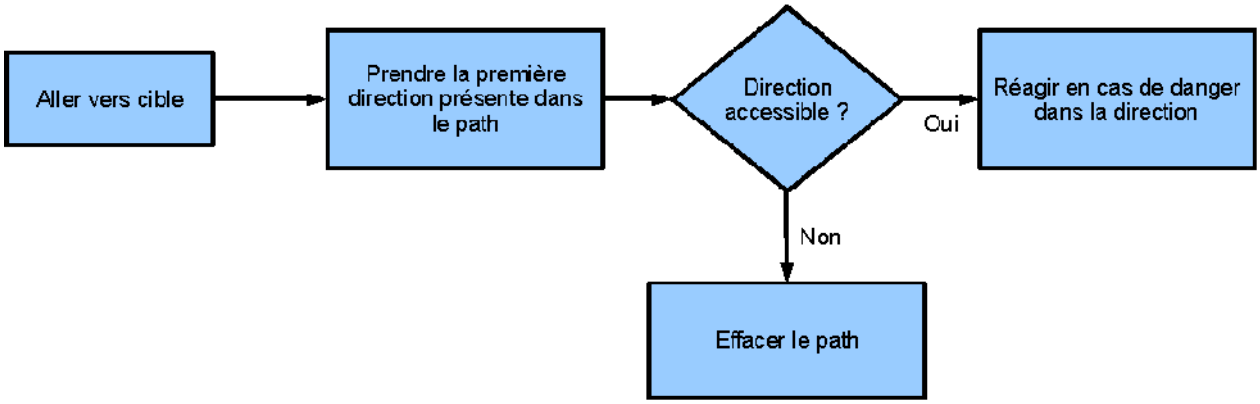
On recherche d'abord pour se déplacer où se trouve la cible. Puis on recherche le chemin pour y aller. Enfin on applique le chemin:



L'entrée « Remplissage du Path » correspond à un prédicat findPath se trouvant dans find_path.pl. Il peut être modélisé de cette manière :

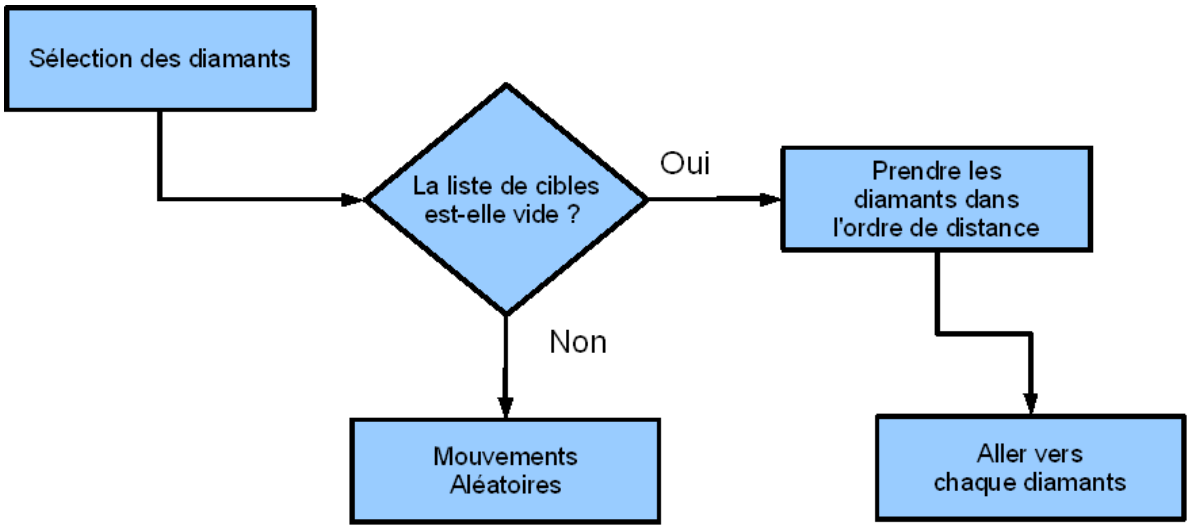


L'entrée « Aller vers cible » correspond à un prédicat goFromPath qui est implémenté dans le fichier principal ai-00.pl dont voilà la structure:



Quant à « Réagir en cas de danger dans la direction », il correspond au point développé dans la partie précédente qui s'appelle « Réagir en cas de danger ».

b) Sélection des diamants



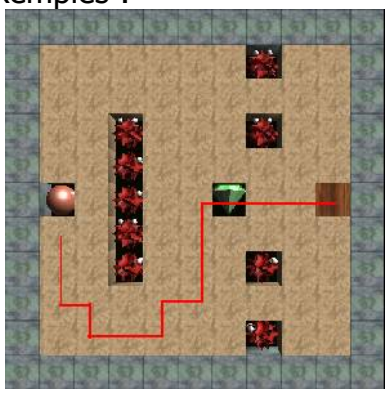
c) Sélection de la sortie

La sélection de la sortie se fait simplement avec le prédicat getExit se trouvant dans le module general_functions.pl. Ce prédicat boucle simplement sur la map pour trouver la sortie.

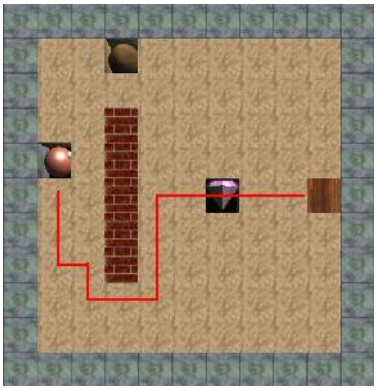
3. Le comportement de notre mineur

3.1. Les situations traitées

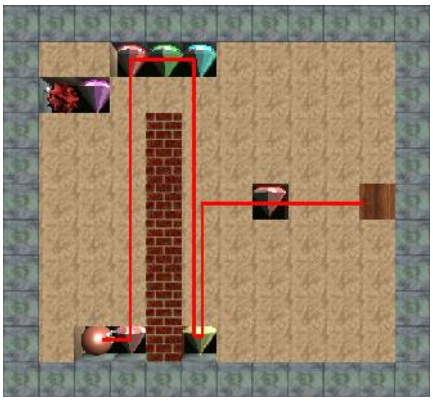
A défaut de pouvoir finir les niveaux proposés par le jeu, nous avons préféré créer des cartes afin de montrer ce que le mineur sait faire. En mettant le mineur dans des situations particulières, il arrive à la sortie. En voici quelques exemples :



Dans ce premier cas, le mineur évite autant que possible les ennemis. C'est l'utilisation de l'algorithme A* qui permet de trouver le chemin le plus sûr. (le canGoto empêche le mineur d'aller dans une case voisine d'un monstre)

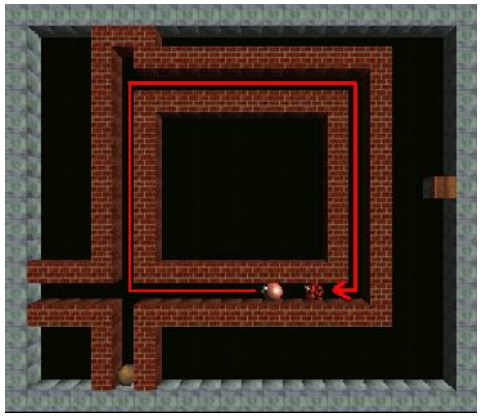


Pour illustrer le fait que le mineur ne prend pas toujours le chemin le plus court, mais le plus sûr, nous lui donnons la possibilité dans cette carte de passer sous une pierre (le coût de ce chemin va augmenter) ou bien un passage plus long mais sans risque de modifier la configuration des pierres.



Enfin, dans cette troisième carte, nous allons montrer le comportement du mineur lorsqu'il repère un bijou à coté de sa trajectoire. C'est le fonctionnement du beHungry qui est ici mis en avant. Dans ce cas, le mineur va repérer les bijoux par ordre de distance. Le premier bijoux qu'il va repérer est donc ici celui se trouvant de l'autre coté du mur, il va donc essayer d'aller le manger en contournant le mur, mais son comportement primitif va l'obliger à manger les diamants qui se trouvent juste à coté de lui sur son parcours. Cela va éviter qu'il passe à coté de diamants sans les prendre. Bien sur il ne prend pas un diamant se trouvant à coté d'un monstre. Le beHungry intègre cette restriction de sureté. Il reprend ensuite son chemin pour récupérer le diamant à coté du mur.

Notre mineur va bien sûr éviter les dangers lorsqu'il ne bouge pas à l'aide de son comportement primitif. Il sera capable de gérer des situations comme celles de ce schéma :



Ici, le mineur respecte bien ses règles de survie :

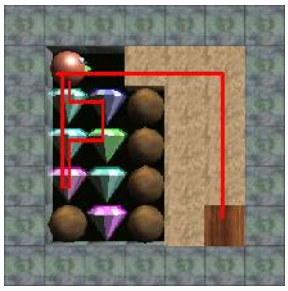
Premièrement il échappe au monstre, ensuite il évite la chute de pierre, et le plus important, il contrôle bien que la direction dans laquelle il se dirige comporte une échappatoire. C'est à dire que le mineur doit pouvoir s'écarter perpendiculairement à la direction choisie.

Sur ce schéma on remarque que ça n'est pas le cas dans les 2 voix en bas à gauche. C'est pour ça que le mineur n'y entre jamais, sauf s'il n'avait pas eu le choix.

On remarque que le mineur évalue les chemins sans issue (en bas a gauche par exemple).

3.2. Les cartes du jeu

Le mineur réussi le premier niveau qui est assez simple. Le mineur prends juste le nombre de diamant nécessaires pour aller à la sortie.



Les difficultés commencent dans les niveaux plus grands où le mineur fait faces à différentes difficultés. Il ramasse aisément les premiers diamants autour de lui mais s'il y a un diamant dans un périmètre d'une dizaine de cases qui n'est pas atteignable, il se bloque.

Un autre problème se pose quand il faut déplacer des pierres pour passer. Par exemple dans le niveau deux :

Dans cette situation il n'y a pas beaucoup de possibilité pour passer, et nous n'avons pas trouver de règle pour résoudre ce problème.

3.3. Améliorations à apporter

Lorsque le mineur ne trouve plus de diamants autour de lui, il adopte un comportement aléatoire dans le but de retrouver un diamant. Nous avons commencé à mettre en place un système de zone pour dire au mineur de se déplacer grossièrement à un endroit comportant encore des diamants, mais nous n'avons pas eu le temps de le finaliser.

Il serait bien entendu possible d'implémenter des prédicats permettant au mineur de débloquer les passages, ou encore de tuer les ennemis.

Conclusion

Ce projet nous a permis de mettre en application les notions que nous avons acquises pendant ce semestre. Il a nécessité beaucoup d'investissement personnel pour l'implémentation de notions que nous avions abordé en cours ce semestre (le A*), mais il nous a surtout permis de passer de la théorie à une application concrète tout en nous familiarisant avec la récursivité et avec un langage déclaratif : Prolog.

Nous avons aussi été amenés à réaliser un vrai travail d'équipe en gérant efficacement la répartition des tâches. En effet, ayant réalisé ce travail à quatre, il a été nécessaire d'organiser correctement le travail de chacun. Nous avons par exemple structuré notre programme Prolog en développant de petits prédicats répartis dans plusieurs fichiers afin de favoriser le travail de groupe.

ANNEXES : Code source commenté

ai-00.pl

```
:- consult( 'general_functions.pl' ).
:- consult( 'find_path.pl' ).
:- consult( 'safety_restrictions.pl' ).
:- consult( 'global_direction.pl' ).
:- consult( 'find_diamonds.pl' ).
:- consult( 'be_hungry.pl' ).

/*
  Définition de l'IA du mineur
  Les prédicats setViewPerimeter/2 et move/6 sont consultés dans le jeu.
*/
/*
  setViewPerimeter( -SX,-SY )
  Périmètre de vue du mineur.
  Si SX=0 ou SY=0, le mineur a une connaissance globale du sous-terrain.
  Si SX>0 et SY>0, le mineur perçoit l'ensemble des (2*SX+1)*(2*SY+1) cases autour de lui.
*/
setViewPerimeter( 0,0 ).

/*
  move( +L,+X,+Y,+Pos,+Size,+CanGotoExit,-Dx,-Dy )

  * L représente la liste des items perçus par le mineur
  * (X,Y) représente la position absolue du mineur dans la zone
  * Pos représente la position du mineur dans la liste L
  * Size représente le nombre d'items dans une ligne stockée dans la liste L.
    Soit P la position du mineur dans la liste. Alors :
    * P-1 indique la case à sa gauche,
    * P+1 indique la case à sa droite,
    * P-Size indique la case en haut,
    * P+Size indique la case en bas
  * CanGotoExit indique si le mineur peut atteindre la sortie ou non
  * (Dx,Dy) représente le mouvement que le mineur doit effectuer sur la base de ce qu'il a perçu
*/

% Tester si les variables globales ont été crées, et les créer si nécessaire
move( L, X, Y, Pos, Size, CanGotoExit, Dx, Dy ) :-
        (not(nb_current(path, Val));
         not(nb_current(diamondsTargets, Val2));
         not(nb_current(unreachableTargetFromPos, Val3)) )
       , nb_setval(path, [])
       , nb_setval(diamondsTargets, [])
       , nb_setval(unreachableTargetFromPos, [])
       , dir(4, Dx, Dy), !.

% Si on est en danger
move( L, X, Y, Pos, Size, CanGotoExit, Dx, Dy ) :-
       gotoSafely( L, X, Y, Pos, Size, CanGotoExit, Dx1, Dy1, 4)
       , (Dx1 \= 0; Dy1 \= 0)
       , !
          , gotoSafely( L, X, Y, Pos, Size, CanGotoExit, Dx, Dy, 4).
% Si on peut aller à la sortie
move( L, X, Y, Pos, Size, 1, Dx, Dy ) :-
       nb_getval(path, [])
       , !
       , getExit( L, Exit)
       , !, findPath(L, Pos, Size, Exit, -1, Directions)
          , nb_setval(path, Directions)
          , goFromPath(L, Pos, Size, Dx, Dy).

% Si on peut aller à la sortie
move( L, X, Y, Pos, Size, 1, Dx, Dy ) :-
         nb_getval(path, Valeur)
         , goFromPath(L, Pos, Size, Dx, Dy).

% Si le mineur a faim
move( L, X, Y, Pos, Size, 0, Dx, Dy ) :-
         beHungry( L, X, Y, Pos, Size, 0, Dx1, Dy1)
       , dir(Dir, Dx1, Dy1)
       , gotoSafely( L, X, Y, Pos, Size, CanGotoExit, Dx, Dy, Dir)
```

```
       , !, nb_setval(path, []).

% Si la variable global path existe, on lance findDiamonds()
move( L, X, Y, Pos, Size, 0, Dx, Dy ) :-
          lookForDiamonds( L, X, Y, Pos, Size, 0, Dx1, Dy1 )
          , dir(Dir, Dx1, Dy1)
          , gotoSafely( L, X, Y, Pos, Size, 0, Dx, Dy, Dir)
          , ( (Dx1 =:= Dx, Dy1 =:= Dy) ; nb_setval(path, []) ).

randomMove( L, X, Y, Pos, Size, CanGotoExit, Dx, Dy ) :-
          D is random( 4 )
          , gotoSafely( L, X, Y, Pos, Size, CanGotoExit, Dx, Dy, D)
          1, !.

randomMoveTo( L, X, Y, Pos, Size, CanGotoExit, Dx, Dy, DirList) :-
          length(DirList, DirListLen)
          , D is random( DirListLen )
          , nth0(D, DirList, Val)
          , gotoSafely( L, X, Y, Pos, Size, CanGotoExit, Dx, Dy, Val)
          , !.

ecrire( T ) :-
          open( 'trace.txt', append, L )
          , write( L, T )
          , nl( L )
          , close( L ).



/*
  goFromPath(L, Pos, Size, Dx, Dy) :
  regarder dans la variable globale path le chemin qu'il faut prendre :
  - on s'arrête si il est vide
  - On prend la première direction et s'y on peut y aller on l'enlève de la liste et on
    la retourne
  - Si on ne peut pas y aller on vide le path pour qu'il soit recalculé
*/

% Si le path est vide, on s'arrête
goFromPath(L, Pos, Size, Dx, Dy) :- nb_getval(path, []), !.

% Si on peut aller dans la direction indiquée dans le path et que le path n'est pas vide
goFromPath(L, Pos, Size, Dx, Dy) :-
              nb_getval(path, [Dir|Path])
            , canGoto(L, Pos, Size, Dir)
            , !, nb_setval(path, Path)
               , gotoSafely( L, X, Y, Pos, Size, CanGotoExit, Dx, Dy, Dir).

% Si on ne peut pas aller dans la direction indiquée dans le path, on le vide pour le recalculer
goFromPath(L, Pos, Size, Dx, Dy) :-
              nb_setval(path, []), !.


/***********************************
 * Définition des quatre directions *
 * 0 : gauche                       *
 * 1 : droite                       *
 * 2 : haut                         *
 * 3 : bas                          *
 ***********************************/
dir( 0,-1, 0 ):- !.
dir( 1, 1, 0 ):- !.
dir( 2, 0,-1 ):- !.
dir( 3, 0, 1 ):- !.
dir( 4, 0, 0 ):- !.


/*
  Affichage des coordonnées du mineur et d'un texte adéquat s'il a ramassé tous les diamants
nécessaires
*/
display( X,Y,0 ) :- write( X ), write( ',' ), write( Y ), nl.
display( X,Y,1 ) :- write( X ), write( ',' ), write( Y ), write( ' - Tous les diamants necessaires
ont ete recoltes !' ), nl.
```

find_diamonds.pl

```
findDiamonds(L, X, Y, Pos, Size, MaxDistanceDiamonds, CostLimit):-
  subFindDiamonds(L, X, Y, Pos, Size, 1, MaxDistanceDiamonds, CostLimit)
  , !
  , nb_getval(diamondsTargets, DT).

subFindDiamonds(_, _, _, _, _, DistanceDiamonds, MaxDistanceDiamonds, _):-
  DistanceDiamonds>MaxDistanceDiamonds, !.
subFindDiamonds(L, X, Y, Pos, Size, DistanceDiamonds, MaxDistanceDiamonds, CostLimit):-
  fetchAndAddDiamonds(L, X, Y, Pos, Size, DistanceDiamonds, CostLimit)
  , NextDistanceDiamonds is DistanceDiamonds + 1
  , subFindDiamonds(L, X, Y, Pos, Size, NextDistanceDiamonds, MaxDistanceDiamonds, CostLimit).

/*
 * fetchAndAddDiamonds : Rechercher et ajoute à targets les diamants se trouvant à une
 * certaine distance et dont le coût ne dépasse pas CostLimit
 * N'échoue jamais !
 * Paramètres :
 *   + standards : L, X, Y, Pos, Size
 *   + DistanceDiamonds : Distance à laquelle doivent se trouver les diamants du mineur
 *   + CostLimit : Coût maximum d'un diamant pour qu'il soit ajouté à Targets
 */
fetchAndAddDiamonds(L, X, Y, Pos, Size, DistanceDiamonds, CostLimit):-
  findDiamondsAtDistance(L, X, Y, Pos, Size, DistanceDiamonds, DiamondsList)
  , addAllToTargetsIfCostOk(L, Pos, Size, DiamondsList, CostLimit)
  , !.
fetchAndAddDiamonds(_, _, _, _, _, _, _).

/*
 * findDiamondsAtDistance : Retourne tous les diamants se trouvant à une
 * certaine distance (égale strictement) du mineur
 * Si aucun diamants : retourne une liste vide
 * Paramètres :
 *   + standards : L, X, Y, Pos, Size
 *   + DistanceDiamonds : Distance à laquelle doivent se trouver les diamants du mineur
 *   - DiamondsList : Liste de la position des diamants
 */
findDiamondsAtDistance(L, X, Y, Pos, Size, DistanceDiamonds, DiamondsList):-
  getDistanceIdList(L, X, Y, Pos, Size, DistanceDiamonds, IdList)
  , getInterestPointsInAList(L, IdList, DiamondsList).

% Une boucle pour une liste de diamants à ajouter
addAllToTargetsIfCostOk(L, Pos, Size, [], CostLimit) :- !.
addAllToTargetsIfCostOk(L, Pos, Size, [E|R], CostLimit) :- addToTargets(E),
addAllToTargetsIfCostOk(L, Pos, Size, R, CostLimit), !.

/*
 * addToTargets : Ajoute la position du diamant à la var globale diamondsTargets si elle n'est pas
déjà dedans
 * Si targets n'existe pas, on la crée
 * N'échoue jamais !
 * Paramètres :
 *   + DiamondPos : Position du diamant */
% Si la liste des cibles existe et que le diamant n'y est pas déjà, on l'ajoute
addToTargets(DiamondPos):- nb_current(diamondsTargets, Targets)
  , not(member(DiamondPos, Targets))
  , !
  , append(Targets, [DiamondPos], NewDiamondsTargets)
  , nb_setval(diamondsTargets, NewDiamondsTargets).

% Si le diamant existe déjà dans la liste des cibles, on l'ignore
addToTargets(DiamondPos):- nb_current(diamondsTargets, Targets)
  , member(DiamondPos, Targets)
  , !.

% Si la liste des cibles n'existe pas encore, on la crée
addToTargets(DiamondPos):-
  nb_setval(diamondsTargets, [DiamondPos])
  , !.
```

find_path.pl

```
/*
 ###########################################
 #                  A Star                 #
 ###########################################
 # Cet algorithme est appelé par le prédicat findPath(L, Pos, Size,
 #   DestinationPos, PathDirections) et renvoie
 # une liste des directions à suivre (forme 0, 1, 2, 3) dans la variable
 #   PathDirections.
 # Cette liste représente le plus court chemin vers la position DestinationPos,
 #   en partant de Pos.
 #
 # Des prédicats intermédiaires sont utilisés comme:
 #  - loop_findPath qui boucle sur l'algo A* (findPath servant juste à
 #  initialiser loop_findPath)
 #  - addChildsNodes qui ajoute les noeuds voisin d'un noeud passé en
 #  paramètre, en controlant leur
 #    validité sur la carte du mineur, et renvoie la nouvelle liste de noeuds
 #  - buildDirectionFromClosedList qui transforme la Liste Fermée du A star en
 #    liste de Direction
 #    (utilise loop_build... pour boucler et getDirId pour transformer 2
 #     positions en 1 direction)
 ###########################################
 #                                         #
 # prédicats utilisables (explications):   #
 #      - getNode                   |   Renvoie le noeud correspondant ([Pos,
 #                                      Father, Cost]) provenant
 #      - getNodeFromFather         |-> de la liste de noeuds (Nodes) passée
 #                                      en paramètre.
 #      - getNodeFromCost           |-> Selon le cas, on peut spécifier Pos,
 #                                      Father ou encore Cost.
 #      - getLowerCostNode          |   getLowerCostNode renvoie automatique
 #                            le noeud dont le cout global est le plus faible.
 #                                      #
 #      - takeNode                  |
 #      - takeNodeFromFather        |-> Comme les 'get' avec en plus une
 #                            suppression du noeud dans la liste de noeuds.
 #      - takeNodeFromCost          |-> La nouvelle liste est donc renvoyée
 #      - takeNodeFromSmallerCost   |
 #                                      #
 #      - addNode                    -> Ajoute un noeud dans une liste en
 #                                 vérifiant s'il est déjà présent. Dans ce cas,
 #                                      s'il est 'meilleur', on remplace le
 #                                 précédent.
 #      - existsNode                 -> Vrai si le noeud existe dans la liste
 #                                 de noeuds.
 #                                      #
 #      - rmNode                    |-> Supprime un noeud d'une liste de
 #                                 noeuds.
 #      - rmLowerCostNode           |   Renvoie la nouvelle liste (privée de
 #                                 ce noeud).
 #                                      #
 ###########################################
 # prédicats utilisables (example):        #
 #      - getNode(Nodes, Pos, Node)        avec Nodes = [Node1, Node2 ...] et
 #                                         Node = [Pos, Father, Cost]
 #      - getNodeFromFather(Nodes, Father, Node)
 #      - getNodeFromCost(Nodes, Cost, Node)
 #      - getLowerCostNode(Nodes, Size, DestinationPos, Node)
 #                                      #
 #      - takeNode(Nodes, Pos, Node, NewNodes)                avec NewNodes
 #                                          la liste Nodes privée de Node
 #      - takeNodeFromFather(Nodes, Father, Node, NewNodes)
 #      - takeNodeFromCost(Nodes, Cost, Node, NewNodes)
 #      - takeNodeFromSmallerCost(Nodes, Size, DestinationPos, Node, NewNodes)
 #                                      #
 #      - addNode(Nodes, Node, NewNodes)
 #      - existsNode(Nodes, Node)
 #                                      #
 #      - rmNode(Nodes, Pos, NewNodes)
 #      - rmLowerCostNode(Nodes, NewNodes)
 #                                      #
 ###########################################
 */
```

```
/************
 * findPath *
 ************
 * findPath(L, Pos, Size, DestinationPos, CostLimit, PathDirections)
 *
 * Trouve le chemin d'une position à une autre.
 * Pos est la position de départ et DestinationPos celle d'arrivée
 *
 * CostLimit arrête la recherche si le cout du déplacement est supérieur à ce
 * nombre (CostLimit)
 * Si ce nombre vaut -1, la recherche est illimitée.
 * PathDirections est une liste de directions (0, 1, 2, 3) que findPath
 * renvoie.
 */
findPath(L, Pos, Size, DestinationPos, CostLimit, PathDirections) :-
  loop_findPath(L, Size, DestinationPos, CostLimit, [[Pos,-1,0]], [], PathDirections,
NewOpenedList, NewClosedList)
  , !.

loop_findPath(L, Size, DestinationPos, CostLimit, _, ClosedList, PathDirections, _, _) :-
  existsNode(ClosedList, DestinationPos)
  , !
  , buildDirectionFromClosedList(Size, ClosedList, DestinationPos, PathDirections).
loop_findPath(L, Size, DestinationPos, CostLimit, OpenedList, ClosedList, PathDirections,
NewOpenedList, NewClosedList) :-
  takeLowerCostNode(OpenedList, Size, DestinationPos, [Pos,Father,LowerCost], NewOpenedList2)
  , TopCase is Pos-Size
  , LeftCase is Pos-1
  , RightCase is Pos+1
  , BottomCase is Pos+Size
  , addNode(ClosedList, [Pos,Father,LowerCost], NewClosedList2)
  , addChildsNodes(L, Size, DestinationPos, CostLimit, NewOpenedList2, NewClosedList2,
[Pos,Father,LowerCost], [TopCase, LeftCase, RightCase, BottomCase], NewOpenedList3, NewClosedList3)
  , !
  , loop_findPath(L, Size, DestinationPos, CostLimit, NewOpenedList3, NewClosedList3,
PathDirections, NewOpenedList, NewClosedList).

/************
 * getNode *
 ************/
getNode([], _, []) :- !.
getNode([[PosNum,Father,Cost] | _], PosNum, [PosNum,Father,Cost]) :- !.
getNode([_ | OtherNodes], PosNum, Node) :- getNode(OtherNodes, PosNum, Node), !.

/*********************
 * getNodeFromFather *
 *********************/
getNodeFromFather([], _, []) :- !.
getNodeFromFather([[PosNum,Father,Cost] | _], Father, [PosNum,Father,Cost]) :- !.
getNodeFromFather([_ | OtherNodes], Father, Node) :- getNode(OtherNodes, Father, Node), !.

/*******************
 * getNodeFromCost *
 *******************/
getNodeFromCost([], _, []) :- !.
getNodeFromCost([[PosNum,Father,Cost] | _], Cost, [PosNum,Father,Cost]) :- !.
getNodeFromCost([_ | OtherNodes], Cost, Node) :- getNodeFromCost(OtherNodes, Cost, Node), !.

/********************
 * getLowerCostNode *
 ********************/
getLowerCostNode(Nodes, Size, DestinationPos, Node) :-
  findLowerCost(Nodes, Size, DestinationPos, LowerCost)
  , getNodeFromCost(Nodes, LowerCost, Node)
  , !.

findLowerCost([], _, _, -1) :- !.
findLowerCost(Nodes, Size, DestinationPos, LowerCost) :-   loop_findLowerCost(Nodes, [_,_,-1],
LowerCost, Size, DestinationPos)
  , !.

loop_findLowerCost([], [_,_,TmpLowerCost], TmpLowerCost, Size, DestinationPos) :- !.
loop_findLowerCost([[Pos,_,Cost] | OtherNodes], [_,_,-1], LowerCost, Size, DestinationPos) :-
  loop_findLowerCost(OtherNodes, [Pos,_,Cost], LowerCost, Size, DestinationPos), !.

loop_findLowerCost([[Pos,_,Cost] | OtherNodes], [LowerCostPos,_,TmpLowerCost], LowerCost, Size,
```

```
DestinationPos) :-
   distanceS(Size, Pos, DestinationPos, ActualDistance)
   , ActualTotal is ActualDistance + Cost
   , distanceS(Size, LowerCostPos, DestinationPos, LowerCostDistance)
   , LowerCostTotal is LowerCostDistance + TmpLowerCost
   , ActualTotal < LowerCostTotal, Cost >= 0
   , !
   , loop_findLowerCost(OtherNodes, [Pos,_,Cost], LowerCost, Size, DestinationPos).

loop_findLowerCost([[_,_,Cost] | OtherNodes], [LowerCostPos,_,TmpLowerCost], LowerCost, Size,
DestinationPos) :-
   loop_findLowerCost(OtherNodes, [LowerCostPos,_,TmpLowerCost], LowerCost, Size, DestinationPos).

/************
 * takeNode *
 ************/
takeNode([], _, [], []) :- !.
takeNode([[PosNum,Father,Cost] | OtherNodes], PosNum, [PosNum,Father,Cost], OtherNodes) :- !.
takeNode([HeadNode | OtherNodes], PosNum, Node, [HeadNode | OtherNewNodes]) :-
   takeNode(OtherNodes, PosNum, Node, OtherNewNodes)
   , !.

/**********************
 * takeNodeFromFather *
 **********************/
takeNodeFromFather([], _, [], []) :- !.
takeNodeFromFather([[PosNum,Father,Cost] | OtherNodes], Father, [PosNum,Father,Cost],
OtherNodes) :- !.
takeNodeFromFather([HeadNode | OtherNodes], Father, Node, [HeadNode | OtherNewNodes]) :-
   takeNodeFromFather(OtherNodes, Father, Node, OtherNewNodes), !.

/**********************
 * takeNodeFromCost *
 **********************/
takeNodeFromCost([], _, [], []) :-
   !.
takeNodeFromCost([[PosNum,Father,Cost] | OtherNodes], Cost, [PosNum,Father,Cost], OtherNodes) :-
   !.
takeNodeFromCost([HeadNode | OtherNodes], Cost, Node, [HeadNode | OtherNewNodes]) :-
   takeNodeFromCost(OtherNodes, Cost, Node, OtherNewNodes), !.

/********************
 * takeLowerCostNode *
 ********************/
takeLowerCostNode(Nodes, Size, DestinationPos, Node, NewNodes) :-  findLowerCost(Nodes, Size,
DestinationPos, LowerCost), takeNodeFromCost(Nodes, LowerCost, Node, NewNodes), !.

/**************
 * existsNode *
 **************/
existsNode([], _) :- fail, !.
existsNode([[PosNum,Father,Cost] | _], PosNum) :- !.
existsNode([_ | OtherNodes], PosNum) :-
   existsNode(OtherNodes, PosNum), !.

/************
 * addNode *
 ************/
addNode([], Node, [Node]) :- !.
addNode([[LP,LF,LC] | LO], [P, F, C], NewNodes) :-
   existsNode([[LP,LF,LC] | LO], P)
   , !
   , replaceNode([[LP,LF,LC] | LO], [P, F, C], NewNodes).
addNode(Nodes, Node, [Node | Nodes]) :-
   !.

replaceNode([], _, []), !.
replaceNode([[P, LF, LC] | O], [P, F, C], [[P, F, C] | O]) :- LC > C, !.
replaceNode([[P, LF, LC] | O], [P, F, C], [[P, LF, LC] | O]) :- !.
replaceNode([HeadNode | LO], Node, [HeadNode | O]) :-
   replaceNode(LO, Node, O), !.

/***********
 * rmNode *
 ***********/
rmNode([], _, []) :- !.
```

```
rmNode([[PosNum,Father,Cost] | OtherNodes], PosNum, OtherNodes) :- !.
rmNode([HeadNode | OtherNodes], PosNum, [HeadNode | OtherNewNodes]) :-
   rmNode(OtherNodes, PosNum, OtherNewNodes), !.

/*******************
 * rmLowerCostNode *
 *******************/
rmLowerCostNode(Nodes, NewNodes) :-
   takeLowerCostNode(Nodes, Size, DestinationPos, _, NewNodes)
   , !.

/******************
 * addChildsNodes *
 ******************
 * addChildsNodes(L[], Size, DestinationPos, CostLimit, OpenedList[][], ClosedList[][],
FatherNode[], ChildsNodesPos[], NewOpenedList[][], NewClosedList[][])
 *
 * Permet à partir d'une LISTE DE POSITIONS DE NOEUDS fils, et d'un NOEUD père,
 * de renvoyer une nouvelle LISTE OUVERTE contenant les noeuds fils de la liste
 * fournie.
 *
 * Leur père est le père fourni et leur cout est celui du père + 1
 *
 * Si le cout des noeud est plus grand que CostLimit, ils ne sont pas ajoutés
 */
addChildsNodes(L, Size, DestinationPos, CostLimit, OpenedList, ClosedList, [Pos,Father,LowerCost],
[], OpenedList, ClosedList) :- !.

% Si le child existe dans la liste fermée :
addChildsNodes(L, Size, DestinationPos, CostLimit, OpenedList, ClosedList, [Pos,Father,LowerCost],
[ChildNodePos|OtherChildsNodes], NewOpenedList, NewClosedList) :-
   ChildCost is LowerCost + 1
   , (ChildCost =< CostLimit ; CostLimit =:= -1)
   , existsNode(ClosedList, ChildNodePos)
   , addChildsNodes(L, Size, DestinationPos, CostLimit, OpenedList, ClosedList,
[Pos,Father,LowerCost], OtherChildsNodes, NewOpenedList, NewClosedList2)
   , !
   , addNode(NewClosedList2, [ChildNodePos, Pos, ChildCost], NewClosedList).

% Si le child n'est pas un obstacle
addChildsNodes(L, Size, DestinationPos, CostLimit, OpenedList, ClosedList, [Pos,Father,LowerCost],
[ChildNodePos|OtherChildsNodes], NewOpenedList, NewClosedList) :-
   getDirId(Size, Pos, ChildNodePos, Dir)
   , canGoto(L, Pos, Size, Dir, DestinationPos, HasPushDesti)
   , ( ( ChildCost is (LowerCost + 1 + HasPushDesti*5 + 3) , fallDanger(L, Pos, Size)) % Ajout d'un
cout au danger de chute
      ; ChildCost is (LowerCost + 1 + HasPushDesti*5) )
                                                          % Sans danger de chute
   , (ChildCost =< CostLimit ; CostLimit =:= -1)
   , addChildsNodes(L, Size, DestinationPos, CostLimit, OpenedList, ClosedList,
[Pos,Father,LowerCost], OtherChildsNodes, NewOpenedList2, NewClosedList)
   , !
   , addNode(NewOpenedList2, [ChildNodePos, Pos, ChildCost], NewOpenedList).

% Sinon
addChildsNodes(L, Size, DestinationPos, CostLimit, OpenedList, ClosedList, [Pos,Father,LowerCost],
[ChildNodePos|OtherChildsNodes], NewOpenedList, NewClosedList) :-
   addChildsNodes(L, Size, DestinationPos, CostLimit, OpenedList, ClosedList,
[Pos,Father,LowerCost], OtherChildsNodes, NewOpenedList, NewClosedList).


/*******************************
 * buildDirectionFromClosedList *
 *******************************
 * buildDirectionFromClosedList(Size, ClosedList[][], Pos, PathDirections[])
 *
 * Permet de construire une liste de directions à partir de la liste
 * fernmée de noeuds
 *
 * (appelé lorsque la liste fermée contient la position de destination)
 */
buildDirectionFromClosedList(Size, ClosedList, Pos, PathDirections) :-
   loop_buildDirectionFromClosedList(Size, ClosedList, Pos, PathDirections2)
   , reverse(PathDirections2, PathDirections), !.

loop_buildDirectionFromClosedList(Size, ClosedList, Pos, []) :-
```

```
   getNode(ClosedList, Pos, [Pos,-1,Cost]), !.

loop_buildDirectionFromClosedList(Size, ClosedList, Pos, [Dir|OtherDirections]) :-
   getNode(ClosedList, Pos, [Pos,PreviousPos,Cost])
   , !
   , getDirId(Size, PreviousPos, Pos, Dir)
   , loop_buildDirectionFromClosedList(Size, ClosedList, PreviousPos, OtherDirections).


/*#########################################################*
 #              .:: canGoto du Astar ::.                   #
 # Version dérivée de canGoto, (cf. general_functions.pl) #
 #                                                         #
 # Idem mais on fourni une position au prédicat qui        #
 # renverra une variable nous disant si la position a été #
 # poussée ou pas (0 ou 1)                                 #
 *#########################################################*/

/********************
 * canGoto du Astar * Peut-on aller dans la direction donnée ?
 ********************/
canGoto(_, _, _, 4, _, 0) :- !.

%% Si la position à gauche est remplacable directement (pas poussable)
% Test si la position à droite est la Desti spécifié, on renvoie 1
canGoto(L, Pos, Size, 0, Desti, 1) :-
   PosI is Pos-1
   , ILine is PosI // Size, PlayerLine is Pos // Size
      % est-ce bien sur la même ligne ?
   , not( ennemiDanger(L, Pos, Size, 0) )
   , ILine =:= PlayerLine, Desti =:= PosI, nth0(PosI, L, Elem), isReplacable(Elem)
   , !.

% Sinon renvoie 0
canGoto(L, Pos, Size, 0, Desti, 0) :-
   PosI is Pos-1
  , ILine is PosI // Size, PlayerLine is Pos // Size
                                                % est-ce bien sur la même ligne ?
   , not( ennemiDanger(L, Pos, Size, 0) )
   , ILine =:= PlayerLine, nth0(PosI, L, Elem), isReplacable(Elem), !.

%% Si la position suivante est poussable on appelle canPush
% Idem pour l'actuel mais avec une boucle sur la suite
canGoto(L, Pos, Size, 0, Desti, 1) :-
   PosI is Pos-1
   , ILine is PosI // Size, PlayerLine is Pos // Size % est-ce bien sur la même ligne ?
   , not( ennemiDanger(L, Pos, Size, 0) )
   , ILine =:= PlayerLine
   , Desti =:= PosI
   , nth0(PosI, L, Elem)
   , isDirectlyPushable(Elem)
   , canPush(L, Pos, Size, PosI, 0, Desti, HasPushDesti)
   , !. % la var HasPush n'est pas intéressante ici

% Idem pour l'actuel mais avec une boucle sur la suite
canGoto(L, Pos, Size, 0, Desti, HasPushDesti) :-
   PosI is Pos-1
   , ILine is PosI // Size
   , PlayerLine is Pos // Size     % est-ce bien sur la même ligne ?
   , not( ennemiDanger(L, Pos, Size, 0) )
   , ILine =:= PlayerLine
   , nth0(PosI, L, Elem)
   , isDirectlyPushable(Elem)
   , canPush(L, Pos, Size, PosI, 0, Desti, HasPushDesti)
   , !. % ici la var HasPush est importante

%% On refais pareil pour chaque direction
canGoto(L, Pos, Size, 1, Desti, 1) :-
   PosI is Pos+1
   , ILine is PosI // Size
   , PlayerLine is Pos // Size
   , not( ennemiDanger(L, Pos, Size, 1) )
   , ILine =:= PlayerLine
   , Desti =:= PosI
   , nth0(PosI, L, Elem)
   , isReplacable(Elem), !.
```

```
canGoto(L, Pos, Size, 1, Desti, 0) :-
   PosI is Pos+1
   , ILine is PosI // Size
   , PlayerLine is Pos // Size
   , not( ennemiDanger(L, Pos, Size, 1) )
   , ILine =:= PlayerLine, nth0(PosI, L, Elem), isReplacable(Elem), !.
canGoto(L, Pos, Size, 1, Desti, 1) :- PosI is Pos+1
   , ILine is PosI // Size, PlayerLine is Pos // Size
   , not( ennemiDanger(L, Pos, Size, 1) )
   , ILine =:= PlayerLine, Desti =:= PosI, nth0(PosI, L, Elem), isDirectlyPushable(Elem), canPush(L,
Pos, Size, PosI, 1, Desti, HasPushDesti), !.
canGoto(L, Pos, Size, 1, Desti, HasPushDesti) :- PosI is Pos+1
   , ILine is PosI // Size, PlayerLine is Pos // Size
   , not( ennemiDanger(L, Pos, Size, 1) )
   , ILine =:= PlayerLine
   , nth0(PosI, L, Elem), isDirectlyPushable(Elem), canPush(L, Pos, Size, PosI, 1, Desti,
HasPushDesti), !.

canGoto(L, Pos, Size, 2, Desti, 1) :- PosI is Pos-Size
   , ICol is PosI mod Size, PlayerCol is Pos mod Size
   , not( ennemiDanger(L, Pos, Size, 2) )
   , ICol =:= PlayerCol, Desti =:= PosI, nth0(PosI, L, Elem)
   , isReplacable(Elem), !.
canGoto(L, Pos, Size, 2, Desti, 0) :- PosI is Pos-Size
   , ICol is PosI mod Size, PlayerCol is Pos mod Size
   , not( ennemiDanger(L, Pos, Size, 2) )
   , ICol =:= PlayerCol, nth0(PosI, L, Elem), isReplacable(Elem), !.
canGoto(L, Pos, Size, 2, Desti, 1) :- PosI is Pos-Size
   , ICol is PosI mod Size, PlayerCol is Pos mod Size
   , not( ennemiDanger(L, Pos, Size, 2) )
   , ICol =:= PlayerCol, Desti =:= PosI, nth0(PosI, L, Elem), isDirectlyPushable(Elem), canPush(L,
Pos, Size, PosI, 2, Desti, HasPushDesti), !.
canGoto(L, Pos, Size, 2, Desti, HasPushDesti) :- PosI is Pos-Size
   , ICol is PosI mod Size, PlayerCol is Pos mod Size
   , not( ennemiDanger(L, Pos, Size, 2) )
   , ICol =:= PlayerCol, nth0(PosI, L, Elem), isDirectlyPushable(Elem), canPush(L, Pos, Size, PosI,
2, Desti, HasPushDesti), !.

canGoto(L, Pos, Size, 3, Desti, 1) :- PosI is Pos+Size
   , ICol is PosI mod Size, PlayerCol is Pos mod Size
   , not( ennemiDanger(L, Pos, Size, 3) )
   , ICol =:= PlayerCol, Desti =:= PosI, nth0(PosI, L, Elem), isReplacable(Elem), !.
canGoto(L, Pos, Size, 3, Desti, 0) :- PosI is Pos+Size
   , ICol is PosI mod Size, PlayerCol is Pos mod Size
   , not( ennemiDanger(L, Pos, Size, 3) )
   , ICol =:= PlayerCol, nth0(PosI, L, Elem), isReplacable(Elem), !.
canGoto(L, Pos, Size, 3, Desti, 1) :- PosI is Pos+Size
   , ICol is PosI mod Size, PlayerCol is Pos mod Size
   , not( ennemiDanger(L, Pos, Size, 3) )
   , ICol =:= PlayerCol, Desti =:= PosI, nth0(PosI, L, Elem), isDirectlyPushable(Elem), canPush(L,
Pos, Size, PosI, 3, Desti, HasPushDesti), !.
canGoto(L, Pos, Size, 3, Desti, HasPushDesti) :- PosI is Pos+Size
   , ICol is PosI mod Size, PlayerCol is Pos mod Size
   , not( ennemiDanger(L, Pos, Size, 3) )
   , ICol =:= PlayerCol, nth0(PosI, L, Elem)
   , isDirectlyPushable(Elem)
   , canPush(L, Pos, Size, PosI, 3, Desti, HasPushDesti), !.

/***********
 * canPush * Quelle que soit la direction :
 ***********/
% Si l'élément est bloquant à distance => Fail
canPush(L, Pos, Size, PosI, _, _, 0) :-
           nth0(PosI, L, Elem), not(isIndirectlyPushable(Elem))
           , !
           , fail.

/***********
 * canPush * Spécifique à chaque direction :
 ***********/
canPush(L, Pos, Size, PosI, 0, Desti, 1) :-
           ILine is PosI // Size, PlayerLine is Pos // Size
         , ILine =:= PlayerLine
           , Desti =:= PosI, nth0(PosI, L, Elem)
           , isIndirectlyPushable(Elem), not(isEmpty(Elem))
         , ! , NewPosI is PosI-1
```

```
          , canPush(L, Pos, Size, NewPosI, 0, Desti, HasPushDesti).
canPush(L, Pos, Size, PosI, 0, Desti, HasPushDesti) :-
          ILine is PosI // Size, PlayerLine is Pos // Size
        , ILine =:= PlayerLine, nth0(PosI, L, Elem)
          , isIndirectlyPushable(Elem), not(isEmpty(Elem))
        , ! , NewPosI is PosI-1
          , canPush(L, Pos, Size, NewPosI, 0, Desti, HasPushDesti).
canPush(L, Pos, Size, PosI, 0, Desti, 1) :-
          ILine is PosI // Size, PlayerLine is Pos // Size
        , ILine =:= PlayerLine, Desti =:= PosI
          , nth0(PosI, L, Elem), isEmpty(Elem), !.
canPush(L, Pos, Size, PosI, 0, Desti, 0) :-
          ILine is PosI // Size, PlayerLine is Pos // Size
        , ILine =:= PlayerLine, nth0(PosI, L, Elem), isEmpty(Elem), !.

canPush(L, Pos, Size, PosI, 1, Desti, 1) :-
          ILine is PosI // Size, PlayerLine is Pos // Size
        , ILine =:= PlayerLine, Desti =:= PosI,
             nth0(PosI, L, Elem), isIndirectlyPushable(Elem), not(isEmpty(Elem))
        , ! , NewPosI is PosI+1,
             canPush(L, Pos, Size, NewPosI, 1, Desti, HasPushDesti).
canPush(L, Pos, Size, PosI, 1, Desti, HasPushDesti) :-
          ILine is PosI // Size, PlayerLine is Pos // Size
        , ILine =:= PlayerLine, nth0(PosI, L, Elem)
          , isIndirectlyPushable(Elem), not(isEmpty(Elem))
        , ! , NewPosI is PosI+1
          , canPush(L, Pos, Size, NewPosI, 1, Desti, HasPushDesti).
canPush(L, Pos, Size, PosI, 1, Desti, 1) :-
          ILine is PosI // Size, PlayerLine is Pos // Size
        , ILine =:= PlayerLine, Desti =:= PosI
          , nth0(PosI, L, Elem), isEmpty(Elem), !.
canPush(L, Pos, Size, PosI, 1, Desti, 0) :-
          ILine is PosI // Size, PlayerLine is Pos // Size
        , ILine =:= PlayerLine, nth0(PosI, L, Elem), isEmpty(Elem), !.

canPush(L, Pos, Size, PosI, 2, Desti, 1) :-
          ICol is PosI mod Size, PlayerCol is Pos mod Size
        , ICol =:= PlayerCol, Desti =:= PosI
          , nth0(PosI, L, Elem), isIndirectlyPushable(Elem), not(isEmpty(Elem))
        , ! , NewPosI is PosI-Size
          , canPush(L, Pos, Size, NewPosI, 2, Desti, HasPushDesti).
canPush(L, Pos, Size, PosI, 2, Desti, HasPushDesti) :-
          ICol is PosI mod Size, PlayerCol is Pos mod Size
        , ICol =:= PlayerCol, nth0(PosI, L, Elem)
          , isIndirectlyPushable(Elem), not(isEmpty(Elem))
        , ! , NewPosI is PosI-Size
          , canPush(L, Pos, Size, NewPosI, 2, Desti, HasPushDesti).
canPush(L, Pos, Size, PosI, 2, Desti, 1) :-
          ICol is PosI mod Size, PlayerCol is Pos mod Size
        , ICol =:= PlayerCol, Desti =:= PosI
          , nth0(PosI, L, Elem), isEmpty(Elem), !.
canPush(L, Pos, Size, PosI, 2, Desti, 0) :-
          ICol is PosI mod Size, PlayerCol is Pos mod Size
        , ICol =:= PlayerCol, nth0(PosI, L, Elem), isEmpty(Elem), !.

canPush(L, Pos, Size, PosI, 3, Desti, 1) :-
          ICol is PosI mod Size, PlayerCol is Pos mod Size
        , ICol =:= PlayerCol, Desti =:= PosI, nth0(PosI, L, Elem)
          , isIndirectlyPushable(Elem), not(isEmpty(Elem))
        , ! , NewPosI is PosI+Size
          , canPush(L, Pos, Size, NewPosI, 3, Desti, HasPushDesti).
canPush(L, Pos, Size, PosI, 3, Desti, HasPushDesti) :-
          ICol is PosI mod Size, PlayerCol is Pos mod Size
        , ICol =:= PlayerCol, nth0(PosI, L, Elem), isIndirectlyPushable(Elem), not(isEmpty(Elem))
        , ! , NewPosI is PosI+Size
          , canPush(L, Pos, Size, NewPosI, 3, Desti, HasPushDesti).
canPush(L, Pos, Size, PosI, 3, Desti, 1) :-
          ICol is PosI mod Size, PlayerCol is Pos mod Size
        , ICol =:= PlayerCol, Desti =:= PosI
          , nth0(PosI, L, Elem), isEmpty(Elem), !.
canPush(L, Pos, Size, PosI, 3, Desti, 0) :-
          ICol is PosI mod Size, PlayerCol is Pos mod Size
        , ICol =:= PlayerCol, nth0(PosI, L, Elem), isEmpty(Elem), !.
```

safety_restrictions.pl

```
/***********************************
 * Module de Restrictions de Sûreté *
 ***********************************
 *
 * D'abord on teste si la direction est sans danger:
 *
 * -> Si c'est le cas on la valide
 *
 * -> Sinon, on étudie toutes les directions en leur donnant un cout (un peu
 * comme pour le A*)
 *    On crée donc une liste[][] contenant des listes[] composées par: [la
 * direction, le cout]
 *      Ex: [[0, 2], [1, 5], [2, 1], [3, 9]]
 *    Cette liste ne contient que les directions ok avec canGoto
 *    Une fois que c'est fait, on choisis le cout le plus faible et on valide
 * cette direction.
 *
 * Les variables spéciales impliquées seront:
 *   - DirectionsList, une liste[][] de listes contenants une direction + un
 * cout associé
 *
 * Les prédicats appelés seront:
 *   - gotoSafely pour le traitement général des restrictions
 *   - createDirList pour la création de la liste et le calcul des coûts de
 *     chaque direction
 *   - chooseLowerCostDir pour choisir la direction de plus faible coût parmis
 *     la liste des directions
 */
gotoSafely( L, X, Y, Pos, Size, CanGotoExit, Dx, Dy, Dir) :-
              calculateDirCost(L, X, Y, Pos, Size, Dir, Cost)
            , Cost =< 2
            , !, dir(Dir, Dx, Dy).

gotoSafely( L, X, Y, Pos, Size, CanGotoExit, Dx, Dy, Dir) :-
              createDirList(L, Pos, Size, DirectionsList)
            , chooseLowerCostDir(DirectionsList, LowerCostDir)
            , nb_setval(path, [])
            , dir(LowerCostDir, Dx, Dy), !.


/*****************
 * createDirList *
 *****************
 * createDirList(L, Pos, Size, DirectionsList)
 *
 * Renvoie une liste de directions possibles avec le coût calculé en fonction
 * de l'environnement
 */
createDirList(L, Pos, Size, DirectionsList) :- loop_createDirList(L, Pos, Size, 0, DirectionsList),
!.
% Si on peut y aller, on ajoute (liste non terminée)
loop_createDirList(L, Pos, Size, Dir, [[Dir,Cost]|OtherDir]) :-
  Dir < 4
  , canGoto(L, Pos, Size, Dir)
  , calculateDirCost(L, X, Y, Pos, Size, Dir, Cost)
  , !, NextDir is Dir+1
     , loop_createDirList(L, Pos, Size, NextDir, OtherDir).
% Sinon on ignore (liste non terminée)
loop_createDirList(L, Pos, Size, Dir, OtherDir) :-
  Dir < 4
  , !
    , NextDir is Dir+1
    , loop_createDirList(L, Pos, Size, NextDir, OtherDir).

% Sinon, Liste terminée
loop_createDirList(L, Pos, Size, _, []) :- !.


/********************
 * calculateDirCost *
 ********************
 * calculateDirCost(L, X, Y, Pos, Size, Dir, Cost)
 * Renvoie un cout en fonction d'une direction donnée
 */
```

```
/* TOUTES DIRECTIONS */
% S'il y a un ennemi (coût = 3)
calculateDirCost(L, X, Y, Pos, Size, Dir, Cost) :-
  ennemiDanger(L, Pos, Size, Dir)
  , !
    , calculateDirCost3(L, X, Y, Pos, Size, Dir, 3, Cost).
calculateDirCost(L, X, Y, Pos, Size, Dir, Cost) :-
  calculateDirCost3(L, X, Y, Pos, Size, Dir, 0, Cost)
  , !.

% S'il n'y a pas d'échapatoire (coût = 2)
calculateDirCost3(L, X, Y, Pos, Size, Dir, StartCost, NewCost) :-
  not(isThereAnEscapeWay(L, X, Y, Pos, Size, Dir))
  , NewStartCost is StartCost+2
  , !
  , calculateDirCost2(L, X, Y, Pos, Size, Dir, NewStartCost, NewCost).
calculateDirCost3(L, X, Y, Pos, Size, Dir, StartCost, NewCost) :- calculateDirCost2(L, X, Y, Pos,
Size, Dir, StartCost, NewCost)
  , !.

/* ALLER A GAUCHE */
% S'il y a une chute de pierre à gauche (coût = 3)
calculateDirCost2(L, X, Y, Pos, Size, 0, StartCost, NewCost) :-   underFallAtLeft(L, Pos, Size)
  , NewCost is StartCost+3
  , !.

/* ALLER A DROITE */
% S'il y a une chute de pierre à droite  (coût += 3)
calculateDirCost2(L, X, Y, Pos, Size, 1, StartCost, NewCost) :-   underFallAtRight(L, Pos, Size),
NewCost is StartCost+3, !.

/* ALLER EN HAUT */
% S'il y a une chute au-dessus (coût += 3)
calculateDirCost2(L, X, Y, Pos, Size, 2, StartCost, NewCost) :-
  underFall(L, Pos, Size)
  , NewCost is StartCost+3
  , !.

/* ALLER EN BAS */
% Risque de chute de pierres (coût += 1)
calculateDirCost2(L, X, Y, Pos, Size, 3, StartCost, NewCost) :-
  fallDanger(L, Pos, Size)
  , NewCost is StartCost+1, !.
/* NE PAS BOUGER */
% Si une pierre nous tombe dessus (coût += 4)
calculateDirCost2(L, X, Y, Pos, Size, 4, StartCost, NewCost) :-
  underFall(L, Pos, Size), NewCost is StartCost+4
  , !
    , ecrire(['UnderFall, cout: ', NewCost,' (calculateDirCost2) ', Pos]).

calculateDirCost2(L, X, Y, Pos, Size, _, StartCost, StartCost) :- !.


/**********************
 * chooseLowerCostDir *
 **********************
 * chooseLowerCostDir(DirectionsList, LowerCostDir)
 *
 * Renvoie la direction ayant le plus petit coût de DirectionsList dans
 * LowerCostDir
 */
chooseLowerCostDir(DirectionsList, LowerCostDir) :-
  loop_getLowerCostDir(DirectionsList, [4,-1], LowerCostDir)
  , !.

loop_getLowerCostDir([[Dir,Cost]|OtherDir], [4,-1], LowerCostDir) :-
  loop_getLowerCostDir(OtherDir, [Dir,Cost], LowerCostDir)
  , !.

loop_getLowerCostDir([[Dir,Cost]|OtherDir], [TmpLowerCostDir,TmpLowerCost], LowerCostDir) :-
  TmpLowerCost >= Cost
  , !
  , loop_getLowerCostDir(OtherDir, [Dir,Cost], LowerCostDir).
loop_getLowerCostDir([[_,_]|OtherDir], Tmp, LowerCostDir) :-
  loop_getLowerCostDir(OtherDir, Tmp, LowerCostDir)
  , !.
```

```
loop_getLowerCostDir([], [4,-1], _) :-
   fail, !.
loop_getLowerCostDir([], [TmpLowerCostDir,TmpLowerCost], TmpLowerCostDir) :- !.


/************************************************
 * Indique un danger de chute à gauche ou à droite *
 ************************************************/
% Danger de chute
fallDanger(L, Pos, Size) :-
   fallDangerFromLeft(L, Pos, Size),!;
   fallDangerFromRight(L, Pos, Size),!;
   fallDangerFromTop(L, Pos, Size),!.

fallDangerFromSide(L, Pos, Size) :-
   fallDangerFromLeft(L, Pos, Size),!;
   fallDangerFromRight(L, Pos, Size),!.

fallDangerFromLeft(L, Pos, Size):-
   Pos1 is Pos-Size-1
   , nth0(Pos1, L, TopLeft)
      , isObject(TopLeft)
   ,  Pos2 is Pos-1, nth0(Pos2, L, Left)
      , isObject(Left)
   , Pos3 is Pos-Size, nth0(Pos3, L, Top),
      isEmpty(Top),!.
fallDangerFromRight(L, Pos, Size):-
   Pos1 is Pos-Size+1
   , nth0(Pos1, L, TopRight)
      , isObject(TopRight)
   , Pos2 is Pos+1, nth0(Pos2, L, Right)
      , isObject(Right)
   , Pos3 is Pos-Size, nth0(Pos3, L, Top)
      , isEmpty(Top),!.

fallDangerFromTop(L, Pos, Size):-
   Pos1 is Pos-Size, nth0(Pos1, L, Top)
      , isObject(Top),!.

% Chute en cours (au dessus)
underFall(L, Pos, Size):-
   Pos1 is Pos-Size, nth0(Pos1, L, FirstTop)
      , isEmpty(FirstTop),
   Pos2 is Pos-2*Size, nth0(Pos2, L, SecondTop)
      , isObject(SecondTop),!.

underFall(L, Pos, Size) :-
   Pos1 is Pos-Size, nth0(Pos1, L, FirstTop)
      , isEmpty(FirstTop)
   , Pos2 is Pos-2*Size, nth0(Pos2, L, SecondTop)
      , isEmpty(SecondTop)
   , Pos3 is Pos-3*Size, nth0(Pos3, L, ThirdTop)
      , isObject(ThirdTop),!.

% Chute en cours à droite ou à gauche
%
% Risque imminant seulement, çàd pierre dont la position Y est 1 case au dessus
% du mineur
% Si le mineur va à gauche/droite et que la pierre frappe le mineur directement
% après ça
underFallOnSide(L, Pos, Size) :-
   underFallAtLeft(L, Pos, Size),!;
   underFallAtRight(L, Pos, Size),!.

underFallAtLeft(L, Pos, Size) :-
   Pos1 is Pos-1, nth0(Pos1, L, LeftCase),
     isEmpty(LeftCase),
   Pos2 is Pos-Size-1, nth0(Pos2, L, LeftTopCase)
      , isObject(LeftTopCase),!.

underFallAtLeft(L, Pos, Size) :-
   canGoto(L, Pos, Size, 0)
   , Pos2 is Pos-Size-1, nth0(Pos2, L, LeftTopCase)
      , isEmpty(LeftTopCase)
   , Pos3 is Pos-Size-Size-1, nth0(Pos3, L, LeftTopTopCase)
```

```
     , isObject(LeftTopTopCase),!.

underFallAtRight(L, Pos, Size) :-
   Pos1 is Pos+1, nth0(Pos1, L, RightCase),
   isEmpty(RightCase),
   Pos2 is Pos-Size+1, nth0(Pos2, L, RightTopCase),
   isObject(RightTopCase),!.

underFallAtRight(L, Pos, Size) :-
   canGoto(L, Pos, Size, 1),
   Pos2 is Pos-Size+1, nth0(Pos2, L, RightTopCase),
     isEmpty(RightTopCase),
   Pos3 is Pos-Size-Size+1, nth0(Pos3, L, RightTopTopCase),
     isObject(RightTopTopCase),!.

/**********************
 * Danger des ennemis *
 **********************/
ennemiDanger(L, Pos, Size, 0) :-
   X is (Pos mod Size)
   , Y is (Pos // Size)
   , ennemiDangerFromLeft(L, X, Y, Pos, Size)
   , !.
ennemiDanger(L, Pos, Size, 1) :-
   X is (Pos mod Size)
   , Y is (Pos // Size)
   , ennemiDangerFromRight(L, X, Y, Pos, Size)
   , !.
ennemiDanger(L, Pos, Size, 2) :-
   X is (Pos mod Size)
   , Y is (Pos // Size)
   , ennemiDangerFromTop(L, X, Y, Pos, Size)
   , !.
ennemiDanger(L, Pos, Size, 3) :-
   X is (Pos mod Size)
   , Y is (Pos // Size)
   , ennemiDangerFromBottom(L, X, Y, Pos, Size)
   , !.

ennemiDanger(L, Pos, Size, 4) :-
   X is (Pos mod Size), Y is (Pos // Size)
   , LeftPos is Pos-1, nth0(LeftPos, L, LeftElem)
   , isEmpty(LeftElem), ennemiDangerFromLeft(L, X, Y, Pos, Size), !.
ennemiDanger(L, Pos, Size, 4) :-
   X is (Pos mod Size), Y is (Pos // Size)
   , RightPos is Pos+1, nth0(RightPos, L, RightElem)
   , isEmpty(RightElem), ennemiDangerFromRight(L, X, Y, Pos, Size), !.
ennemiDanger(L, Pos, Size, 4) :-
   X is (Pos mod Size), Y is (Pos // Size)
   , TopPos is Pos-Size, nth0(TopPos, L, TopElem)
   , isEmpty(TopElem), ennemiDangerFromTop(L, X, Y, Pos, Size), !.
ennemiDanger(L, Pos, Size, 4) :-
   X is (Pos mod Size), Y is (Pos // Size)
   , BottomPos is Pos+Size, nth0(BottomPos, L, BottomElem)
   , isEmpty(BottomElem), ennemiDangerFromBottom(L, X, Y, Pos, Size), !.

ennemiDangerFromTop(L, X, Y, Pos, Size) :-
   ennemiAtLeftUpper(L, X, Y, Pos, Size), !.
ennemiDangerFromTop(L, X, Y, Pos, Size) :-
   ennemiAtTopTop(L, X, Y, Pos, Size), !.
ennemiDangerFromTop(L, X, Y, Pos, Size) :-
   ennemiAtRightUpper(L, X, Y, Pos, Size), !.

ennemiDangerFromRight(L, X, Y, Pos, Size) :-
   ennemiAtRightUpper(L, X, Y, Pos, Size), !.
ennemiDangerFromRight(L, X, Y, Pos, Size) :-
   ennemiAtRightRight(L, X, Y, Pos, Size), !.
ennemiDangerFromRight(L, X, Y, Pos, Size) :-
   ennemiAtRightDown(L, X, Y, Pos, Size), !.

ennemiDangerFromBottom(L, X, Y, Pos, Size) :-
   ennemiAtLeftDown(L, X, Y, Pos, Size), !.
ennemiDangerFromBottom(L, X, Y, Pos, Size) :-
   ennemiAtBottomBottom(L, X, Y, Pos, Size), !.
ennemiDangerFromBottom(L, X, Y, Pos, Size) :-
   ennemiAtRightDown(L, X, Y, Pos, Size), !.
```

```
ennemiDangerFromLeft(L, X, Y, Pos, Size) :-
  ennemiAtLeftUpper(L, X, Y, Pos, Size), !.
ennemiDangerFromLeft(L, X, Y, Pos, Size) :-
  ennemiAtLeftLeft(L, X, Y, Pos, Size), !.
ennemiDangerFromLeft(L, X, Y, Pos, Size) :-
  ennemiAtLeftDown(L, X, Y, Pos, Size), !.


ennemiAtLeftUpper(L, X, Y, Pos, Size) :-
  LeftUpper is Pos - Size - 1
  , LeftUpperX is (LeftUpper mod Size)
  , LeftUpperY is (LeftUpper // Size)
  , LeftUpperX =:= X-1, LeftUpperY =:= Y-1
  , nth0(LeftUpper, L, LeftUpperElement)
  , isMonster(LeftUpperElement)
   , !.

ennemiAtTopTop(L, X, Y, Pos, Size) :-
  TopTop is Pos - 2*Size
  , TopTopX is (TopTop mod Size)
  , TopTopY is (TopTop // Size)
  , TopTopX =:= X, TopTopY =:= Y-2
  , nth0(TopTop, L, TopTopElement), isMonster(TopTopElement)
  , !.

ennemiAtRightUpper(L, X, Y, Pos, Size) :-
  RightUpper is Pos - Size + 1, RightUpperX is (RightUpper mod Size), RightUpperY is
(RightUpper // Size)
  , RightUpperX =:= X+1, RightUpperY =:= Y-1
  , nth0(RightUpper, L, RightUpperElement)
  , isMonster(RightUpperElement)
  , !.

ennemiAtRightRight(L, X, Y, Pos, Size) :-
  RightRight is Pos + 2
  , RightRightX is (RightRight mod Size)
  , RightRightY is (RightRight // Size)
  , RightRightX =:= X+2, RightRightY =:= Y
  , nth0(RightRight, L, RightRightElement)
  , isMonster(RightRightElement)
  , !.

ennemiAtRightDown(L, X, Y, Pos, Size) :-
  RightDown is Pos + Size + 1
  , RightDownX is (RightDown mod Size)
  , RightDownY is (RightDown // Size)
  , RightDownX =:= X+1, RightDownY =:= Y+1
  , nth0(RightDown, L, RightDownElement)
  , isMonster(RightDownElement)
  , !.

ennemiAtBottomBottom(L, X, Y, Pos, Size) :-
  BottomBottom is Pos + 2*Size
  , BottomBottomX is (BottomBottom mod Size)
  , BottomBottomY is (BottomBottom // Size)
  , BottomBottomX =:= X, BottomBottomY =:= Y+2
  , nth0(BottomBottom, L, BottomBottomElement)
  , isMonster(BottomBottomElement)
    , !.

ennemiAtLeftDown(L, X, Y, Pos, Size) :-
  LeftDown is Pos + Size - 1
  , LeftDownX is (LeftDown mod Size)
  , LeftDownY is (LeftDown // Size)
  , LeftDownX =:= X-1, LeftDownY =:= Y+1
  , nth0(LeftDown, L, LeftDownElement), isMonster(LeftDownElement)
  , !.

ennemiAtLeftLeft(L, X, Y, Pos, Size) :-
  LeftLeft is Pos - 2
  , LeftLeftX is (LeftLeft mod Size)
  , LeftLeftY is (LeftLeft // Size)
  , LeftLeftX =:= X-2, LeftLeftY =:= Y
  , nth0(LeftLeft, L, LeftLeftElement)
  , isMonster(LeftLeftElement)
```

```
  , !.


/**********************
 * isThereAnEscapeWay *
 **********************
 * isThereAnEscapeWay(L, X, Y, Pos, Size, Dir)
 *
 * Calcul s'il y a une échappatoire dans la direction donnée
 *
 * Le sens de échappatoire ici est simplement un écartement par rapport à la
 * direction donnée
 * Par exemple pour une chute, une échappatoire serait une possibilité d'aller
 * à gauche ou à droite à un moment donné
 */
/* A GAUCHE */
isThereAnEscapeWay(L, X, Y, Pos, Size, 0) :-
  canGoto(L, Pos, Size, 0)
  , LeftPos is Pos-1
  , canGoto(L, LeftPos, Size, 2)
  , !. % On peut aller à gauche puis en haut
isThereAnEscapeWay(L, X, Y, Pos, Size, 0) :-
  canGoto(L, Pos, Size, 0)
  , LeftPos is Pos-1
  , canGoto(L, LeftPos, Size, 3)
  , !. % On peut aller à gauche puis en bas
isThereAnEscapeWay(L, X, Y, Pos, Size, 0) :-
  canGoto(L, Pos, Size, 0)
  , LeftPos is Pos-1
  , isThereAnEscapeWay(L, X, Y, LeftPos, Size, 0)
  , !. % On peut aller à gauche et retenter ça plus loin

/* A DROITE */
isThereAnEscapeWay(L, X, Y, Pos, Size, 1) :-
  canGoto(L, Pos, Size, 1)
  , RightPos is Pos+1
  , canGoto(L, RightPos, Size, 2)
  , !.
isThereAnEscapeWay(L, X, Y, Pos, Size, 1) :-
  canGoto(L, Pos, Size, 1)
  , RightPos is Pos+1
  , canGoto(L, RightPos, Size, 3)
  , !.
isThereAnEscapeWay(L, X, Y, Pos, Size, 1) :-
  canGoto(L, Pos, Size, 1)
  , RightPos is Pos+1
  , isThereAnEscapeWay(L, X, Y, RightPos, Size, 1)
  , !.

/* EN HAUT */
isThereAnEscapeWay(L, X, Y, Pos, Size, 2) :-
  canGoto(L, Pos, Size, 2)
  , TopPos is Pos-Size
  , canGoto(L, TopPos, Size, 0)
  , !.
isThereAnEscapeWay(L, X, Y, Pos, Size, 2) :-
  canGoto(L, Pos, Size, 2)
  , TopPos is Pos-Size
  , canGoto(L, TopPos, Size, 1)
  , !.
isThereAnEscapeWay(L, X, Y, Pos, Size, 2) :-
  canGoto(L, Pos, Size, 2)
  , TopPos is Pos-Size
  , isThereAnEscapeWay(L, X, Y, TopPos, Size, 2)
  , !.

/* EN BAS */
isThereAnEscapeWay(L, X, Y, Pos, Size, 3) :-
  canGoto(L, Pos, Size, 3)
  , BottomPos is Pos+Size
  , canGoto(L, BottomPos, Size, 0)
  , !.
isThereAnEscapeWay(L, X, Y, Pos, Size, 3) :-
  canGoto(L, Pos, Size, 3)
  , BottomPos is Pos+Size
  , canGoto(L, BottomPos, Size, 1)
```

```
  , !.
isThereAnEscapeWay(L, X, Y, Pos, Size, 3) :-
  canGoto(L, Pos, Size, 3)
  , BottomPos is Pos+Size
  , isThereAnEscapeWay(L, X, Y, BottomPos, Size, 3)
  , !.

isThereAnEscapeWay(L, X, Y, Pos, Size, 4) :-
  isThereAnEscapeWay(L, X, Y, Pos, Size, 0), !.
isThereAnEscapeWay(L, X, Y, Pos, Size, 4) :-
  isThereAnEscapeWay(L, X, Y, Pos, Size, 1), !.
isThereAnEscapeWay(L, X, Y, Pos, Size, 4) :-
  isThereAnEscapeWay(L, X, Y, Pos, Size, 2), !.
isThereAnEscapeWay(L, X, Y, Pos, Size, 4) :-
  isThereAnEscapeWay(L, X, Y, Pos, Size, 3), !.
```

```
/********************
 * Prédicats utiles *
 ********************/

% Renvoie l'écart entre le Mineur (pos X et Y) et l'Id spécifié (id dans la liste L)
%
% L'écart sur X et sur Y est renvoyé dans les variables respectives DiffX et DiffY
%
% En gros c'est la distance projetée sur X et Y

distance(X, Y, Size, Id, DiffX, DiffY) :-
     TX is abs((Id mod Size) - X)
     , TX =:= DiffX
     , TY is abs((Id // Size) - Y)
     , TY =:= DiffY.

distanceS(Size, Pos1, Pos2, D) :-
     TX is abs((Pos2 mod Size) - (Pos1 mod Size))
     , TY is abs((Pos2 // Size) - (Pos1 // Size))
     , D is TX + TY.

equals(L, L).




% getIdFromDistance permet de récupérer une liste contenant tous les Index dans la liste L des
% cases situées à une certaine distance du mineur.
%
% ex: On veut les positions des cases qui sont à 3 déplacement du mineur
%

getDistanceIdList(L, X, Y, Pos, Size, Distance, IdList) :-
     listHeight(L, Pos, Size, Height)
     , Id is Pos-Distance, IdX is X-Distance
 % IdX peut être négatif, IdY vaut Y
     , getIdFromDistance(X, Y, Pos, Size, Height, Distance, IdX, Y, Id, IdList).

% Premier quart (en haut à gauche, gauche inclue mais pas le haut):
getIdFromDistance(X, Y, Pos, Size, Height, Distance, IdX, IdY, Id, [Id|R]) :-            IdX < X,
IdY =< Y,
% le quartier + ajout de l'id
     dX >= 0, IdY >= 0,
% n'est pas hors map
     NextIdX is (IdX + 1), NextIdY is (IdY - 1),
% calcul des prochains IdX et IdY
     NextId is (Size*NextIdY + NextIdX),
     !,
% On passe le prochain Id pour l'ajouter à la liste si jamais.
     getIdFromDistance(X, Y, Pos, Size, Height, Distance, NextIdX, NextIdY, NextId, R).

getIdFromDistance(X, Y, Pos, Size, Height, Distance, IdX, IdY, _, ListId) :-             IdX < X,
IdY =< Y,
% le quartier est hors map
     NextIdX is (IdX + 1), NextIdY is (IdY - 1),
% calcul des prochains IdX et IdY
     NextId is (Size*NextIdY + NextIdX),
     !,
% On passe le prochain Id pour l'ajouter à la liste si jamais.
     getIdFromDistance(X, Y, Pos, Size, Height, Distance, NextIdX, NextIdY, NextId, ListId).


% Deuxième quart (en haut à droite, haut inclue mais pas la droite):
getIdFromDistance(X, Y, Pos, Size, Height, Distance, IdX, IdY, Id, [Id|R]) :-            IdX >= X,
IdY < Y,
% le quartier + ajout de l'id à la liste
     IdX < Size, IdY >=0,
% n'est pas hors map
     NextIdX is (IdX + 1),
     NextIdY is (IdY + 1),
% calcul des prochains IdX et IdY
     NextId is (Size*NextIdY + NextIdX),
     !,
% On passe le prochain Id pour l'ajouter à la liste si jamais.
```

```
     getIdFromDistance(X, Y, Pos, Size, Height, Distance, NextIdX, NextIdY, NextId, R).

getIdFromDistance(X, Y, Pos, Size, Height, Distance, IdX, IdY, _, ListId) :-            IdX >= X,
IdY < Y,
% le quartier est hors map
     NextIdX is (IdX + 1),
     NextIdY is (IdY + 1),
% calcul des prochains IdX et IdY
     NextId is (Size*NextIdY + NextIdX),
     !,
% On passe le prochain Id pour l'ajouter à la liste si jamais.
     getIdFromDistance(X, Y, Pos, Size, Height, Distance, NextIdX, NextIdY, NextId, ListId).


% Troisième quart (en bas à droite, droite inclue mais pas le bas):
getIdFromDistance(X, Y, Pos, Size, Height, Distance, IdX, IdY, Id, [Id|R]) :-           IdX > X,
IdY >= Y,
% le quartier + ajout de l'id à la liste
     IdX < Size, IdY < Height,
% n'est pas hors map
     NextIdX is (IdX - 1),
     NextIdY is (IdY + 1),
% calcul des prochains IdX et IdY
     NextId is (Size*NextIdY + NextIdX),
     !,
% On passe le prochain Id pour l'ajouter à la liste si jamais.
     getIdFromDistance(X, Y, Pos, Size, Height, Distance, NextIdX, NextIdY, NextId, R).

getIdFromDistance(X, Y, Pos, Size, Height, Distance, IdX, IdY, _, ListId) :-            IdX > X,
     IdY >= Y,
% le quartier est hors map
     NextIdX is (IdX - 1),
     NextIdY is (IdY + 1),
% calcul des prochains IdX et IdY
     NextId is (Size*NextIdY + NextIdX),
     !,
% On passe le prochain Id pour l'ajouter à la liste si jamais.
     getIdFromDistance(X, Y, Pos, Size, Height, Distance, NextIdX, NextIdY, NextId, ListId).


% Quatrième quart (en bas à gauche, bas inclu mais pas la gauche) PRIVE du dernier élement:
getIdFromDistance(X, Y, Pos, Size, Height, Distance, IdX, IdY, Id, [Id|R]) :-           LastX is
(X-Distance+1),
     LimitY is Y+1,
     IdX =< X,
     IdX > LastX,
     IdY > LimitY,
% le quartier privé du dernier élément + ajout de l'id à la liste
     IdX >= 0, IdY < Height,
% n'est pas hors map
     NextIdX is (IdX - 1),
     NextIdY is (IdY - 1),
% calcul des prochains IdX et IdY
     NextId is (Size*NextIdY + NextIdX),
     !,
% On passe le prochain Id pour l'ajouter à la liste si jamais.
     getIdFromDistance(X, Y, Pos, Size, Height, Distance, NextIdX, NextIdY, NextId, R).

getIdFromDistance(X, Y, Pos, Size, Height, Distance, IdX, IdY, _, ListId) :-            LastX is
(X-Distance+1),
     LimitY is Y+1,
    X,
     IdX > LastX,
     IdY > LimitY,
% le quartier est hors map
     NextIdX is (IdX - 1), NextIdY is (IdY - 1),
% calcul des prochains IdX et IdY
     NextId is (Size*NextIdY + NextIdX),
     !,
% On passe le prochain Id pour l'ajouter à la liste si jamais.
     getIdFromDistance(X, Y, Pos, Size, Height, Distance, NextIdX, NextIdY, NextId, ListId).

% Dernière case (dans le 4ème quart) pour condition d'arrêt
getIdFromDistance(X, Y, Pos, Size, Height, Distance, IdX, IdY, T, [T]) :-
     LastX is (X-Distance+1),
     LastY is Y+1,
```

```
     IdX =:= LastX,
% le dernier + ajout de l'id à la liste
     IdY =:= LastY,
     IdX >= 0, IdY < Height,
     !.
% n'est pas hors map

getIdFromDistance(X, Y, Pos, Size, Height, Distance, IdX, IdY, _, []) :-
     LastX is (X-Distance+1),
     LastY is Y+1,
     IdX =:= LastX,
% le dernier
     IdY =:= LastY,!.
% pas d'ajout de l'id à la liste

% est hors map => pas de check vu qu'on ajoute pas à la liste ici



% Pour savoir si la case représente du vide
isEmpty(0).

% Pour savoir si c'est un diamant ou encore une pierre
isObject(2).
isObject(3).

isDiamond(2).

isInteresting(2).

isExit(17).

% Pour savoir si c'est un monstre
isMonster(11).
isMonster(12).
isMonster(13).
isMonster(14).
isMonster(15).

% Tous les objets que l'on peut pousser
isDirectlyPushable(0).
isDirectlyPushable(1).
isDirectlyPushable(2).
isDirectlyPushable(3).
isDirectlyPushable(16).

% Tous les objets qui ne nous retiennent pas à distance
isIndirectlyPushable(0).
isIndirectlyPushable(2).
isIndirectlyPushable(3).
isIndirectlyPushable(11).
isIndirectlyPushable(12).
isIndirectlyPushable(13).
isIndirectlyPushable(14).
isIndirectlyPushable(15).
isIndirectlyPushable(16).

% Ce que le Mineur peut remplacer sans pousser
isReplacable(0).
isReplacable(1).
isReplacable(2).
isReplacable(17).



/*
 * concat_list(?L, ?L1, ?L2)
 *
 * Renvoie Yes si L est la concaténation des listes L1 et L2
 */
concat_list(L, [], L) :- !.
concat_list([X|R], [X|R1], L2) :- concat_list(R, R1, L2).



% Donne la hauteur de L
```

```
listHeight(L, Pos, Size, Height):-length(L, Len), Height is abs(Len/Size).



/*#########################################################*
 #                  .:: canGoto ::.                        #
 # Détermine si on peut aller dans une certaine direction  #
 #                                                         #
 # Utilise les prédicats canGoto et canPush                #
 # Prédicats alternatifs: canGotoOneSide                   #
 #                        canGotoBothSides                 #
 *#########################################################*/

/***********
 * canGoto * Peut-on aller dans la direction donnée ?
 ***********/

canGoto(_, _, _, 4) :- !.

canGoto(L, Pos, Size, 0) :- PosI is Pos-1
         , ILine is PosI // Size, PlayerLine is Pos // Size
         , ILine =:= PlayerLine, nth0(PosI, L, Elem), isReplacable(Elem), !.
canGoto(L, Pos, Size, 0) :- PosI is Pos-1
         , ILine is PosI // Size, PlayerLine is Pos // Size
         , ILine =:= PlayerLine, nth0(PosI, L, Elem), isDirectlyPushable(Elem), canPush(L, Pos,
Size, PosI, 0), !.

canGoto(L, Pos, Size, 1) :- PosI is Pos+1
         , ILine is PosI // Size, PlayerLine is Pos // Size
         , ILine =:= PlayerLine, nth0(PosI, L, Elem), isReplacable(Elem), !.
canGoto(L, Pos, Size, 1) :- PosI is Pos+1
         , ILine is PosI // Size, PlayerLine is Pos // Size
         , ILine =:= PlayerLine, nth0(PosI, L, Elem), isDirectlyPushable(Elem), canPush(L, Pos,
Size, PosI, 1), !.

canGoto(L, Pos, Size, 2) :- PosI is Pos-Size
         , ICol is PosI mod Size, PlayerCol is Pos mod Size
         , ICol =:= PlayerCol, nth0(PosI, L, Elem), isReplacable(Elem), !.
canGoto(L, Pos, Size, 2) :- PosI is Pos-Size
         , ICol is PosI mod Size, PlayerCol is Pos mod Size
         , ICol =:= PlayerCol, nth0(PosI, L, Elem), isDirectlyPushable(Elem), canPush(L, Pos, Size,
PosI, 2), !.

canGoto(L, Pos, Size, 3) :- PosI is Pos+Size
         , ICol is PosI mod Size, PlayerCol is Pos mod Size
         , ICol =:= PlayerCol, nth0(PosI, L, Elem), isReplacable(Elem), !.
canGoto(L, Pos, Size, 3) :- PosI is Pos+Size
         , ICol is PosI mod Size, PlayerCol is Pos mod Size
         , ICol =:= PlayerCol, nth0(PosI, L, Elem), isDirectlyPushable(Elem), canPush(L, Pos, Size,
PosI, 3), !.



/***********
 * canPush * Quelle que soit la direction :
 ***********/

% Si l'élément est bloquant à distance => Fail
canPush(L, Pos, Size, PosI, _) :-
           nth0(PosI, L, Elem), not(isIndirectlyPushable(Elem))
           , !
           , fail.


/***********
 * canPush * Spécifique à chaque direction :
 ***********/

canPush(L, Pos, Size, PosI, 0) :-
           ILine is PosI // Size, PlayerLine is Pos // Size
         , ILine =:= PlayerLine, nth0(PosI, L, Elem), isIndirectlyPushable(Elem),
not(isEmpty(Elem))
         , ! , NewPosI is PosI-1, canPush(L, Pos, Size, NewPosI, 0).

canPush(L, Pos, Size, PosI, 0) :-
           ILine is PosI // Size, PlayerLine is Pos // Size
```

```
        , ILine =:= PlayerLine, nth0(PosI, L, Elem), isEmpty(Elem), !.


canPush(L, Pos, Size, PosI, 1) :-
          ILine is PosI // Size, PlayerLine is Pos // Size
        , ILine =:= PlayerLine, nth0(PosI, L, Elem), isIndirectlyPushable(Elem),
not(isEmpty(Elem))
        , ! , NewPosI is PosI+1, canPush(L, Pos, Size, NewPosI, 1).

canPush(L, Pos, Size, PosI, 1) :-
          ILine is PosI // Size, PlayerLine is Pos // Size
        , ILine =:= PlayerLine, nth0(PosI, L, Elem), isEmpty(Elem), !.


canPush(L, Pos, Size, PosI, 2) :-
          ICol is PosI mod Size, PlayerCol is Pos mod Size
        , ICol =:= PlayerCol, nth0(PosI, L, Elem), isIndirectlyPushable(Elem), not(isEmpty(Elem))
        , ! , NewPosI is PosI-Size, canPush(L, Pos, Size, NewPosI, 2).

canPush(L, Pos, Size, PosI, 2) :-
          ICol is PosI mod Size, PlayerCol is Pos mod Size
        , ICol =:= PlayerCol, nth0(PosI, L, Elem), isEmpty(Elem), !.


canPush(L, Pos, Size, PosI, 3) :-
          ICol is PosI mod Size, PlayerCol is Pos mod Size
        , ICol =:= PlayerCol, nth0(PosI, L, Elem), isIndirectlyPushable(Elem), not(isEmpty(Elem))
        , ! , NewPosI is PosI+Size, canPush(L, Pos, Size, NewPosI, 3).

canPush(L, Pos, Size, PosI, 3) :-
          ICol is PosI mod Size, PlayerCol is Pos mod Size
        , ICol =:= PlayerCol, nth0(PosI, L, Elem), isEmpty(Elem), !.

/******************
 * canGotoOneSide * Est-ce qu'on peut aller d'un côté au moins ?
 ******************/

canGotoOneSide(L, Pos, Size) :- canGoto(L, Pos, Size, 0), !.
canGotoOneSide(L, Pos, Size) :- canGoto(L, Pos, Size, 1), !.

 /*******************
 * canGotoBothSides * Est-ce qu'on peut aller des deux côtés ?
 ********************/

canGotoOneSide(L, Pos, Size) :- canGoto(L, Pos, Size, 0), canGoto(L, Pos, Size, 1), !.


/*
 * getExit
 *
 * Retourne la position de la sortie
 */

getExit(L, Sortie) :- loop_getExit(L, 0, Sortie), !.
getExit([], _) :- fail, !.

loop_getExit([X|R], CurrentPos, CurrentPos) :- isExit(X), !.
loop_getExit([X|R], CurrentPos, Sortie) :- NextPos is CurrentPos + 1, loop_getExit(R, NextPos,
Sortie), !.

loop_getExit([], _, _) :- fail, !.


/*#####################################################*
 #                 .:: getDirId ::.                    #
 # Renvoie l'Id de Direction (0,1,2,3) à partir des    #
 #  positions de départ et d'arrivée                   #
 *#####################################################*/

getDirId(Size, Pos1, Pos2, 2) :- Diff is Pos1-Pos2, Diff =:= Size, !.
getDirId(Size, Pos1, Pos2, 3) :- Diff is Pos1-Pos2, Diff =:= -Size, !.
getDirId(Size, Pos1, Pos2, 0) :- Diff is Pos1-Pos2, Diff =:= 1, !.
getDirId(Size, Pos1, Pos2, 1) :- Diff is Pos1-Pos2, Diff =:= -1, !.
```

global-direction.pl

```
/*********************
 * Direction Globale *
 *********************/

getInterestPoints( L, X, Y, Pos, Size, MaxDistance, InterestPoints) :-
     getInterest( L, X, Y, Pos, Size, MaxDistance, 1, InterestPoints).

getInterest( L, X, Y, Pos, Size, MaxDistance, Distance, InterestPoints) :-
     Distance < MaxDistance
     , !
     , getDistanceIdList(L, X, Y, Pos, Size, Distance, IdList1)
     , getInterestPointsInAList(L, IdList1, InterestIdList1)
     , concat_list(InterestPoints, InterestIdList1, IdList2)
     , NextDistance is Distance + 1
     , getInterest( L, X, Y, Pos, Size, MaxDistance, NextDistance, IdList2).

getInterest( L, X, Y, Pos, Size, MaxDistance, Distance, InterestPoints) :-
     getDistanceIdList(L, X, Y, Pos, Size, Distance, InterestPoints).



/*
 * getInterestPointsInAList
 *
 * getInterestPointsInAList (L, ListePosition, ListeFinale)
 *
 * Prend la liste L en paramètre, ainsi qu'une liste de positions dans L
 * Si les élément L[position] sont "intéressants" (diamand, ...), leur position est ajoutée
 * à la liste retournée.
 *
 * Ex: si L = [0,2,2,4,5,2,0,1] et ListePosition = [1,4,7],
 *               |     |     |
 *               1     4     7
 * ListeFinale = [1] (car seul la position 1 est intéressante)
 */


getInterestPointsInAList(L, [X | P], [X | F]) :-
     nth0(X, L, E), isInteresting(E), getInterestPointsInAList(L, P, F), !.

getInterestPointsInAList(L, [_ | P], F) :-
     getInterestPointsInAList(L, P, F), !.

getInterestPointsInAList(L, [], []) :- !.







/*******************
 * lookForDiamonds *
 *******************
 * lookForDiamonds( L, X, Y, Pos, Size, CanGotoExit, Dx, Dy )
 *
 * Lance la recherche de diamants, la construction des chemins pour y aller ...
 * On a deux listes:
 *    - path: qui indique les directions à suivre pour le diamant actuel
 *    - diamondsTargets: qui indique les cibles actuelles, le premier élément étant le diamant en
cour de recherche
 *
 * Si diamondsTargets est vide, ça signifie qu'on a pas de cible, il faut donc en trouver ou passer
à une autre zone de la map
 *
 * Sinon on regarde path
 *       Si path est vide, ça veut dire qu'on a pas encore de chemin pour aller vers notre cible
 *       Comme toutes les cibles de diamondsTargets sont atteignable (testées), on recherche le
chemin
 *
 *       Sinon on parcourt le chemin de path
 */
```

```
% Si on est à la position du prochain diamand de la liste, on le supprime (on a pas besoin du
Inutile retourné)
lookForDiamonds( L, X, Y, Pos, Size, CanGotoExit, Dx, Dy ) :-
     getActualDiamond(Pos), !, nextDiamondStage(Inutile).

% Si la liste diamondsTargets est vide, on la remplit et on trouve le premier chemin
lookForDiamonds( L, X, Y, Pos, Size, CanGotoExit, Dx, Dy ) :-
     nb_getval(diamondsTargets, [])
     , MaxDistanceDiamonds is 10, CostLimit is 30
     , findDiamonds(L, X, Y, Pos, Size, MaxDistanceDiamonds, CostLimit)
     , getActualDiamond(ActualDiamondPos)
     , !
     , findPath(L, Pos, Size, ActualDiamondPos, CostLimit, PathDirections)
     , nb_setval(path, PathDirections)
     , dir(4, Dx, Dy).

% Si la liste diamondsTargets est vide et qu'on ne peut pas la remplir, on passe à une autre ZONE
lookForDiamonds( L, X, Y, Pos, Size, CanGotoExit, Dx, Dy ) :-
     nb_getval(diamondsTargets, [])
     , !
     , randomMove( L, X, Y, Pos, Size, CanGotoExit, Dx, Dy ).

% Si le path est vide et que la cible est un diamant, on le rempli (si on arrive ici,
diamondsTargets n'est pas vide)
lookForDiamonds( L, X, Y, Pos, Size, CanGotoExit, Dx, Dy ) :-
     nb_getval(path, [])
     , getActualDiamond(ActualDiamondPos)
     , nth0(ActualDiamondPos, L, Elem)
     , isDiamond(Elem)
     , !
     , nextDiamondStage(NextDiamondPos)
% > On passe à l'étape suivante (si le findPath échoue)
     , findPath(L, Pos, Size, ActualDiamondPos, 30, PathDirections)
% -> On tente un findPath
     , nb_getval(diamondsTargets, OtherDiamondsTargets)
% --> Si on en sort, on récupère le reste des cibles
     , nb_setval(diamondsTargets, [ActualDiamondPos|OtherDiamondsTargets])
% et on remet l'actuel au début
                                                                                % C'est une
sorte de protection si le findPath bloque, la cible aura été supprimée
     , nb_setval(path, PathDirections)
     , dir(4, Dx, Dy).

% Si la destination n'est plus un diamant et qu'il existe une cible après l'actuelle, on rempli le
path à nouveau pour la cible suivante
lookForDiamonds( L, X, Y, Pos, Size, CanGotoExit, Dx, Dy ) :-
     getActualDiamond(ActualDiamondPos)
     , nth0(ActualDiamondPos, L, Elem)
     , not(isDiamond(Elem))
     , nextDiamondStage(NextDiamondPos)
     , !
     , findPath(L, Pos, Size, NextDiamondPos, 30, PathDirections)
     , nb_setval(path, PathDirections)
     , dir(4, Dx, Dy).

% Si la destination n'est plus un diamant et qu'il n'existe pas de cible après l'actuelle, on ne
bouge pas
% La suite se fera au tour suivant
lookForDiamonds( L, X, Y, Pos, Size, CanGotoExit, Dx, Dy ) :-
     getActualDiamond(ActualDiamondPos)
     , nth0(ActualDiamondPos, L, Elem)
     , not(isDiamond(Elem))
     , !
     , dir(4, Dx, Dy).

% Sinon, on le parcourt le path
lookForDiamonds( L, X, Y, Pos, Size, CanGotoExit, Dx, Dy ) :-
     goFromPath(L, Pos, Size, Dx, Dy)
     , !.
```

```
/*
 * getActualDiamond(ActualDiamond)
 * Renvoie le premier diamant de la liste des diamants cibles
 *
 * ? ActualDiamond
 */

getActualDiamond(_) :-
     nb_current(diamondsTargets, [])
     , fail
     , !.

getActualDiamond(ActualDiamond) :-
     nb_current(diamondsTargets, [ActualDiamond|_])
     , !.


/*
 * nextDiamondStage(NextDiamond)
 * Renvoie le prochain diamant de la liste des diamants cibles après
 * avoir supprimé le diamant actuel
 *
 * ? NextDiamond
 */

nextDiamondStage(_) :-
     nb_current(diamondsTargets, [])
     , fail
     , !.

nextDiamondStage(NextDiamond) :-
     nb_current(diamondsTargets, [_|Others])
     , nb_setval(diamondsTargets, Others)
     , getActualDiamond(NextDiamond)
     , !.
```

be_hungry.pl

```
beHungry(L, X, Y, Pos, Size, CanGotoExit, Dx, Dy) :-
     loop_beHungry(L, X, Y, Pos, Size, CanGotoExit, 0, Dx, Dy)
     , !.

% Si on n'a pas trouvé a manger
loop_beHungry(L, X, Y, Pos, Size, CanGotoExit, 4, _, _) :-
     fail
     , !.

% Si on ne peut pas manger dans cette direction
loop_beHungry(L, X, Y, Pos, Size, CanGotoExit, Dir, Dx, Dy) :-
     Dir < 4
     , not(canEat(L, Pos, Size, Dir))
     , !
     , NextDir is Dir + 1
     , loop_beHungry(L, X, Y, Pos, Size, CanGotoExit, NextDir, Dx, Dy).

% Si on peut
loop_beHungry(L, X, Y, Pos, Size, CanGotoExit, Dir, Dx, Dy) :-
     Dir < 4
     , !
     , dir(Dir, Dx, Dy).


canEat(L, Pos, Size, 0) :- PosI is Pos-1, PosI >= 0
        , ILine is PosI // Size, PlayerLine is Pos // Size
        , ILine =:= PlayerLine, nth0(PosI, L, Elem), isDiamond(Elem), !.

canEat(L, Pos, Size, 1) :- PosI is Pos+1, length(L, Len), PosI < Len
        , ILine is PosI // Size, PlayerLine is Pos // Size
        , ILine =:= PlayerLine, nth0(PosI, L, Elem), isDiamond(Elem), !.

canEat(L, Pos, Size, 2) :- PosI is Pos-Size, PosI >= 0
         , ICol is PosI mod Size, PlayerCol is Pos mod Size
         , ICol =:= PlayerCol, nth0(PosI, L, Elem), isDiamond(Elem), !.

canEat(L, Pos, Size, 3) :- PosI is Pos+Size, length(L, Len), PosI < Len
         , ICol is PosI mod Size, PlayerCol is Pos mod Size
         , ICol =:= PlayerCol, nth0(PosI, L, Elem), isDiamond(Elem), !.
```