IA41 - Delirium 2

Bastien CRAMILLET Stéphane GENET Thomas KRAUSE
Xavier MICHEL

5 janvier 2010

Sommaire

1 Introduction

Dans le cadre de notre UV IA41 à l'UTBM, nous devions réaliser un projet de fin de semestre. Le choix du sujet s'est porté sur la création d'une intelligence artificielle pour un jeu vidéo. Dans le cas présent, il s'agit du jeu `Delirium 2`. Dans ce jeu, le joueur dirige un mineur se déplaçant dans des galeries souterraines à la recherche de diamants. Sur son parcours le mineur doit éviter un certain nombre de pièges, comme des monstres ou des rochers pouvant lui tomber sur la tête. Pour chaque souterrain, le joueur doit ramasser un certain nombre de diamant sur la carte pour ensuite se rendre à la sortie du niveau et passer au souterrain suivant.

Le but du projet est donc de rendre le mineur complètement autonome, ce dernier devant trouver seul son chemin pour récupérer les diamants, éviter les pièges et les ennemies qui l'entour puis rejoindre la sortie.

Ainsi, toutes les situations devront être étudiées pour que le mineur réponde intelligemment à celles-ci et fasse les choix les plus judicieux pour son évolution.

Nous allons vous présenter, à travers ce rapport, les différents problèmes que peut rencontrer le mineur et présenter les solutions que nous avons implémentées. Mais dans un premier temps, il est nécéssaire de constituer une spécification détaillée.

2 Spécifications

La réalisation de la spécification s'est organisée par l'intermédiaire de réunions. Ces rencontres ont permis d'étudier les principes élémentaires requis pour une bonne évolution du mineur dans les souterrains. Ils nous on également permit de diviser notre travail en plusieurs partie. Chacune ayant pour rôle de répondre à un problème particulier.

Il est évident que pour réaliser ce projet il a fallu communiquer en dehors de ces séances. Un serveur de gestion de version à été mis en place pour y déposé notre travail et une chaine de courriel à été utilisé pour la communication sur les différents états du projet.

Comme précisé en introduction, le jeu Délirium 2 met en situation un mineur devant évoluer dans des souterrains et répondant à certaines situations. Après analyse des différents niveaux déja implémentés dans le jeu de base, voici ce qui ressort de notre phase de spécifications :

1. Le mineur peut se déplacer dans 4 directions primaires (vers le haut, vers le bas, vers la droite, vers la gauche)
2. Son déplacement d'une case A à une case B n'est possible que si la case B est de type "vide" ou "herbe"
3. Son déplacement d'une case A à une case B est également possible si la case B est un rocher pouvant être déplacé.
4. Une série de rochers (allant de 1 à N rochers) peut être déplacée si la case terminant la série de rocher et étant à l'opposée du mineur est de type "vide"
5. Une série de rochers peut être déplacée vers la droite, vers la gauche ou vers le haut si elle répond aux points précédents.
6. Son déplacement doit également prendre en compte des dangers pouvant provenir du haut.
7. Un rocher peut tomber de sa position initiale si le mineur déplace une série de rochers inférieure, récupère un diamant ou creuse une case "herbe" qui soutenait le rocher.
8. Le mineur peut récolter un diamant en position A, en arrivant des cases situées à gauche, à droite, en haut, ou en bas de la position du diamant.
9. Cette règle peut être éronnée si le diamant est en cours de chute libre tout comme pourrait le faire un rocher (Cf : puce numéro 7). Dans cette situation le mineur ne peut pas récupérer le diamant par une case inférieure sinon il meurt.
10. Le mineur peut rencontrer des ennemis le long de son parcours.
11. Il y a 2 types d'ennemis que nous devons prendre en compte. Les monstres bleus et monstres rouges.
12. Ces deux ennemis doivent être évités. C'est à dire que le mineur ne doit pas entrer dans un périmètre de deux cases autour du monstre.
13. Un monstre ne peut se déplacer que d'une case "vide" à une case "vide".
14. Certains monstre protègent des diamants. Le mineur doit dans certaines situations trouver une solution pour permettre de débloquer le diamant sans se faire attrapper par le monstre protégeant le diamant.
15. Les monstres peuvent être tués si un rocher leur tombe sur la tête.
16. Les souterrains peuvent contenir plusieurs types de "mur" infranchissable par les monstre et le mineur. Dans cet enssemble de mur existe des murs pouvant être détruits en faisant exploser un monstre à coté un certain nombre de fois (allant de 1 fois à 4 fois suivant le type de mur).

17. Certaines situations contraignent le mineur à devoir provoquer la destruction d'un mur pour pouvoir débloquer un passage vers une autre partie du souterrain.
18. Une carte de souterrain est définie par un fichier XML.
19. Afin de terminer un niveau (ou souterrain), le mineur doit récolter un nombre de diamant minimum. Ce paramètre est définit dans le code XML de la carte.
20. Le mineur est également contraint par un temps de jeu qu'il ne doit pas dépasser pour chaque niveau. Ce paramètre est également définit dans le code XML de la carte.
21. Le mineur doit pouvoir trouver le diamant le plus proche dans son périmètre de vue.
22. Le mineur doit pouvoir trouver le chemin le plus rapide d'un point A à un point B. En prenant en compte les élements cités précédement.
23. Le mineur doit pouvoir abandonner un diamant qui ne sera jamais accessible. (Diamant entouré de mur par exemple.).
24. Le mineur doit se concentrer sur la sortie dès que le nombre de diamants récoltés est égale au nombre de diamants nécéssaire pour terminer le niveau.

L'intelligence artificielle devra être développée grâce au langage Prolog. Il sera également préférable de séparer les différentes groupes de prédicats de même famille dans des modules prolog. Nous détaillons nos modules dans la partie réalisation de notre rapport.

3 Réalisation

Différentes stratégies et algorithmes ont été développés pour répondre aux attentes de la phase de spécifications. Qu'il s'agisse du déplacement, de la recherche d'un diamant, l'élimination d'un ennemi ou d'un passage rendu difficile par la chute de rocher ...

3.1 Recherche du plus court chemin : Algorithme A*

Une solution avait été mise en place en début de projet, qui consistait à développer l'`Algorithme de Dijkstra`. Elle fut remplacée par une méthode plus adaptée à notre problème.

Cet algorithme reconnu de recherche de plus court chemin dans des graphes a été implémenté pour gérer les déplacements du mineur à partir d'un point de départ donné, d'un point d'arrivé et d'une liste représentant la carte du jeu. Il a été implémenté dans plusCourtChemin.pl. A partir de ces éléments l'algorithme étudie les directions possibles que peut prendre le mineur. Considérons la carte de jeu comme un tableau, chaque direction possible ajoute un lien entre la case où se trouve le mineur et celle où il peut aller, définissant ainsi un chemin possible. Tous les chemins possibles sont retenus dans une liste dite *"ouverte"*, les chemins que le mineur ne peut pas prendre, comme par exemple un mur ou un rocher bloquant le passage, sont retenus dans une liste dite *"fermée"*. L'opération de recherche d'une direction possible est répétée à partir des chemins contenus dans la liste ouverte jusqu'à atteindre le point d'arrivée. Si un chemin s'avère être inefficace et n'abouti à rien, il sera à son tour inséré dans la liste fermée et retiré de la liste ouverte. Le chemin semblant être le plus court sera prioritaire et le mineur l'empruntera.

`Complément d'informations à cette adresse :`
`http://fr.wikipedia.org/wiki/Algorithme_A*`

3.2 Stratégie de recherche du plus proche diamant

3.2.1 Situation simple

Dans le cas le plus simple, le mineur va chercher à aller au diamant le plus proche de lui. Par exemple sur la figure ci-dessous, le mineur va tracer son chemin vers le diamant numéro 1 puis se déplacer. Après ce déplacement il recherchera le dimant qui sera le plus proche de lui (à nouveau le 1) ; il se déplacera donc à nouveau dans la direction de ce diamant, et ainsi de suite...

3.2.2 Situation plus complexe

En réalité, il arrive très souvent que le diamant le plus proche du mineur soit inaccessible auquel cas on va préférer s'intéresser au diamant suivant. Par exemple sur la figure suivante, le diamant 1 est inaccessible, le mineur va alors rechercher au diamant numéro 2. Ce dernier étant également inaccessible, le mineur va alors se déplacer vers le diamant noté 3.

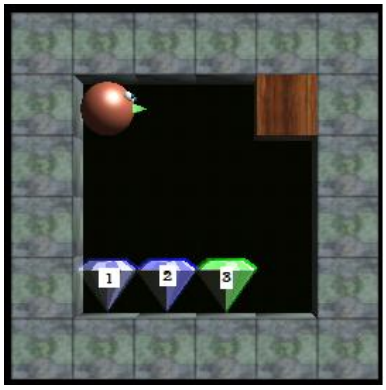

FIG. 1 – Représentation d'une situation simple de recherche du plus proche diamant

FIG. 2 – Représentation d'une situation complexe de recherche du plus proche diamant

3.3 Déplacement de rochers

La stratégie de déplacement de rochers est très simple.
Tout d'abord il faut savoir que le mineur évite le plus possible de déplacer des rochers. S'il peut effectuer le déplacement sans déplacer de rochers, il le fera. Cette technique permet au mineur de ne pas bloquer stupidement ses objectifs. Par exemple sur la situation ci-dessous le mineur est tenté de partir sur la gauche à son premier déplacement auquel cas il bloquera la sortie. Grâce à l'utilisation du chemin sans déplacement de rochers, la sortie reste disponible.

Cependant le mineur est apte à déplacer les rochers ci cela devenait nécessaire, sur une profondeur choisit arbitrairement qui est de 5 rochers. Par exemple il peut pousser 5 rochers sur sa gauche si la situation s'y prête (pas d'obstacle derrière les rochers...). Par exemple dans la situation ci-dessous le mineur va bien évidemment déplacer les rochers pour se sortir de cette situation.

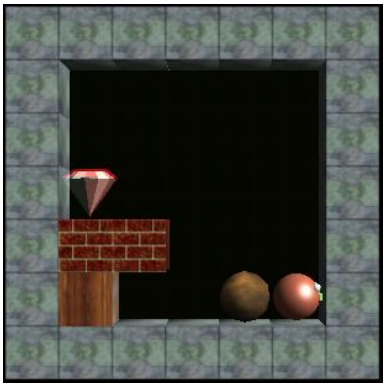

FIG. 3 – Situation où le déplacement d'un rocher est possible mais pas nécéssaire

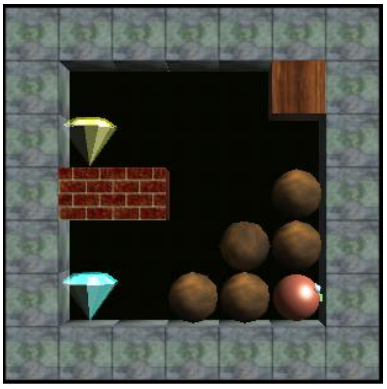

FIG. 4 – Situation où le déplacement d'un rocher est obligatoire

Les rochers se déplacent parfois seuls, comme dans le cas d'une chute sur le mineur. Dans ce cas le mineur va alors détecter qu'il est menacé et il esquivera les rochers tombant en se décalant sur la gauche ou sur la droite selon les possibilités et l'objectif à atteindre.

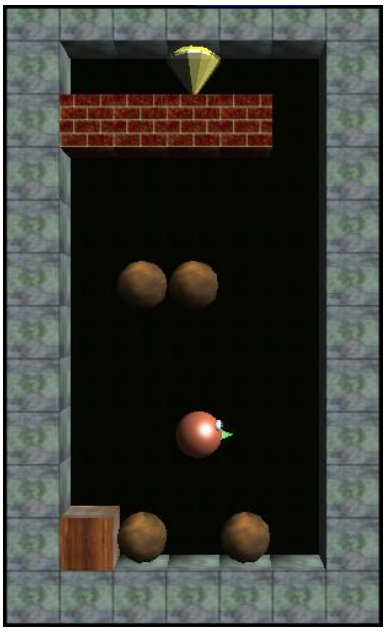

FIG. 5 – Chute libre de plusieurs rochers sur le mineur

3.4 Stratégie mise en place pour éviter un monstre

3.4.1 Cas général

Pour le cas général, dès que le mineur rencontre un monstre dans son champ de vision, la liste représentant le tableau de cases visibles par le mineur est modifiée avant de lui laisser chercher le plus court chemin vers le prochain diamant ou la sortie.
Chaque monstre présent dans le champ de vision du mineur est alors encerclé de murs à 2 cases aux alentours. L'algorithme A* recherchera donc un chemin lui permettant d'éviter le monstre. Bien entendu les murs ajoutés pour éviter le monstre ne sont pas visible pour le joueur.

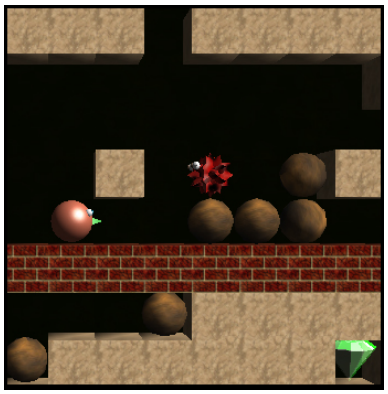

FIG. 6 – Monstre mettant en danger le mineur : vision du joueur

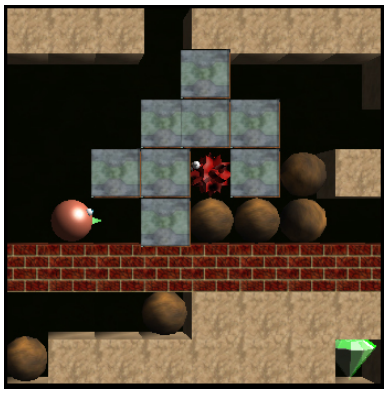

FIG. 7 – Monstre mettant en danger le mineur : vision technique

3.4.2 Cas plus complexe

Afin de ne pas perturber l'environnement de la carte. Seules les cases accessibles par le mineur (case vide et case d'herbe) sont remplacées par des murs. Mais nous aurions également pu avoir cette configuration :

Sur cette représentation seules les cases marquées d'un marqueur gris seront remplacées par des murs. Le diamant n'est pas remplacé, ni la case herbe du dessous. Nous ne remplaÃ§ons donc pas la case de niveau supérieur à une case non remplacée. Ici la case herbe est au niveau 2 du champ de vision du monstre, le diamant au niveau 1 est bloquant pour la case herbe. La case herbe n'est donc pas remplacée. Cette représentation nous permet de résoudre certaines situations.

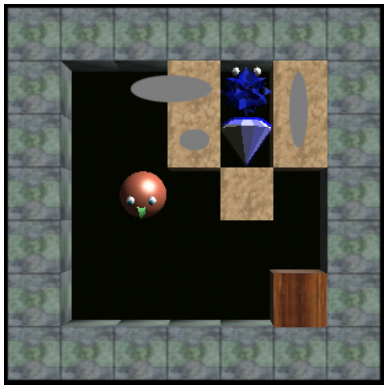

FIG. 8 – Limite de la construction de murs autour d'un monstre

3.5 Situation particulière 1 : Dimant piégé par un monstre

3.5.1 Aperçu de la situation

Sur certaines cartes, un monstre peut bloquer l'accès au mineur à un diamant. Voici un aperçu de la situation :

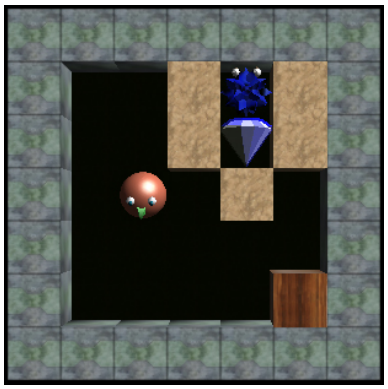

FIG. 9 – Aperçu de la situation particulière 1

3.5.2 Résolution de la situation

Afin de résoudre cette situation, le mineur devra creuser la case herbe située en dessous du diamant, puis s'écarter pour laisser tomber le diamant. L'algorithme standard de recherche du plus proche diamant et l'encerclement du monstre par des murs permettra ensuite au mineur de résoudre cette carte. Voici une image explicative de la solution :

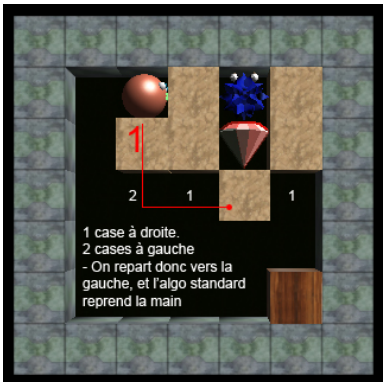

FIG. 10 – Résolution de la situation particulière 1

3.6 Situation particulière 2 : Attaque des montres

Cette stratégie est implémentée dans situation2.pl. Elle consiste à neutraliser un ennemis dans une situation donnée que nous allons détailler par la suite. Elle se décompose en trois phases.

3.6.1 Phase 1 : repérer la situation

La situation doit se présenter sous la forme suivante : un monstre se promène sur une ligne bloquée sur une courte distance. Au dessus de cette ligne se trouve une ligne d'herbe. Enfin, un rocher doit être présent au dessus de la ligne d'herbe en bout de course du monstre. La situation typique est résumée sur l'image ci-dessous.

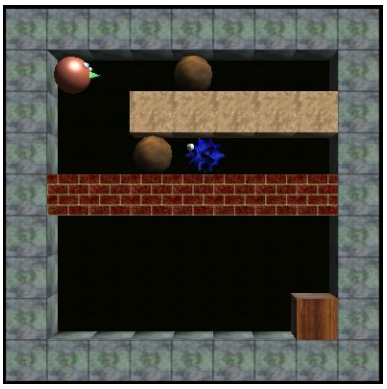

FIG. 11 – Phase 1 de résolution de la situation 2

Cette situation étant repéré, le mineur va alors se mettre en position en sécurité comme sur l'image ci-dessous.

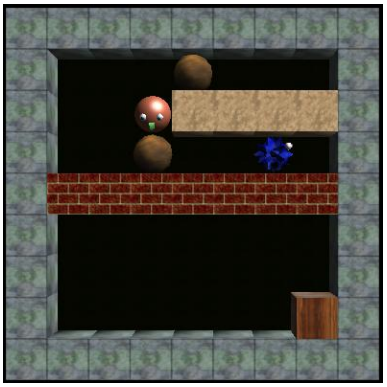

FIG. 12 – Phase 1 de résolution de la situation 2

3.6.2 Phase 2 : placement sous le rocher

Le mineur est à ce moment en position d'attente, il va alors attendre que le monstre se situe à quatre cases de distance à partir de sa butée à gauche pour se placer sous le rocher et se mettre en position d'attente, comme mis en image ci-dessous.

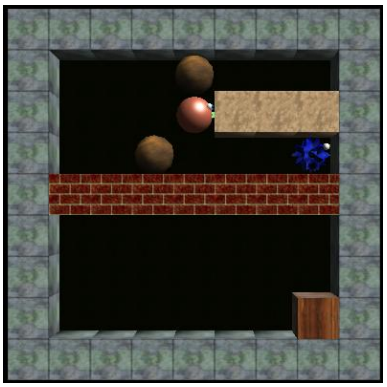

FIG. 13 – Phase 2 de résolution de la situation 2

3.6.3 Phase 3 : Déclenchement du piège

Enfin, lorsque le monstre s'approche, le mineur va s'écarter de sous le rocher de faÃ§on à ce que celui-ci tombe sur le monstre. Ainsi le monstre est neutralisé.

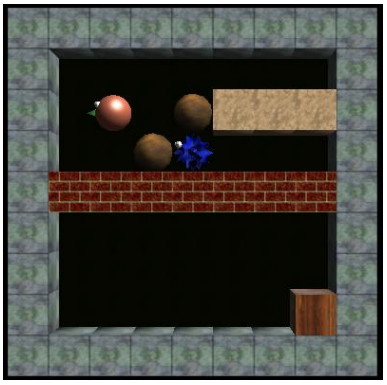

FIG. 14 – Phase 3 de résolution de la situation 2

3.7 EXTRA : Editeur de carte Html/JavaScript

Dès le début du projet, il nous était difficile de concevoir des cartes résumant une situation particulière. Nous n'arrivions pas à visualiser correctement ce que nous concevions comme carte dans le fichier XML. Nous avons donc pensé à concevoir un éditeur de carte Delirium 2.0 en HTML JavaScript.

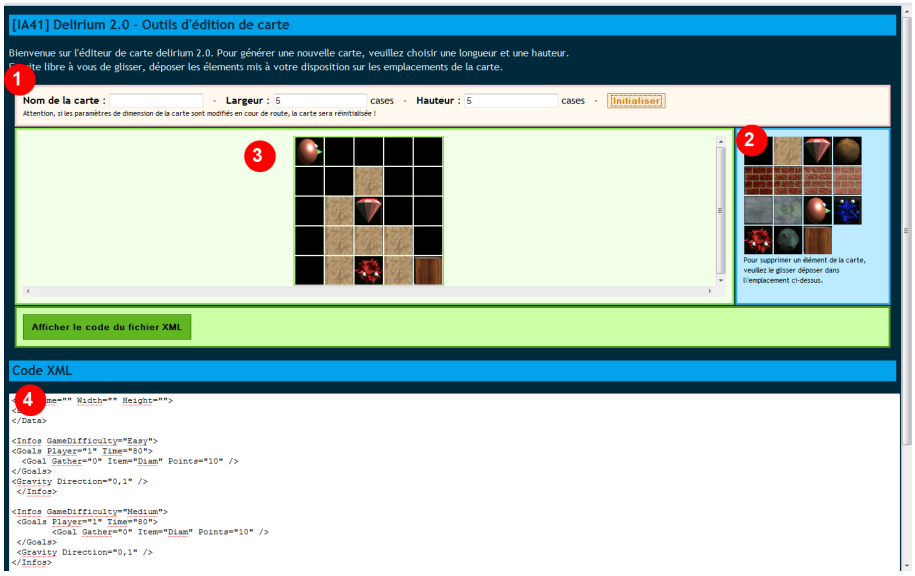

FIG. 15 – Editeur de carte Delirium 2

1. Nous renseignons les informations de la carte
2. Nous avons à disposition les éléments de la carte
3. Nous pouvons glisser depuis le « 2 » les éléments de la carte sur la carte.
4. Le code XML du fichier est généré automatiquement.

4 Conclusion

Lors de ce projet de plusieurs mois, nous devions réalisation l'intelligence artificielle du héro d'un jeu vidéo, le permettant de se déplacer, de récupérer ses objectifs de mission, à savoir, des diamants, ainsi qu'éviter les ennemis qui le pourchassent et les pièges tendus par le décor et ses rochers. Aujourd'hui, notre héro est tout à fait capable de se débrouiller dans ce genre d'environnement, en prenant la direction des diamants et en évitant ou éliminant les ennemis, se servant même de l'environnement pour s'en sortir indemne. Les objectifs que nous nous étions posés ont donc été remplis.

De nombreuses difficultés ont été rencontrées, comme pour l'implémentation de l'algorithme A* qui est quelque peu délicat à programmer ou les situations d'esquive et d'élimination des ennemis. Travailler avec un langage récursif n'est pas non plus chose aisé au début.

Le travail que nous avons réalisé peut tout à faire être amélioré, notamment au niveau de la rapidité d'exécution qui est parfois assez lente, principalement lors de passages complexes ou de nombreux chemins sont possibles pour notre héro.

5 Codes sources du programme

5.1 ai-00.pl

```
:-use_module('outilsCarte').
:-use_module('plusCourtChemin').
:-use_module('eviterMonstre').
:-use_module('situations').
:-use_module('situations2').

/*
  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( 3,3 ).


/*
        trouve la position du diamant le plus proche à partir du coin
           haut-gauche, enfin en ligne par ligne bof bof
        L : la liste des cases
        Code : code de l'élément à trouver , par exemple 2 pour les
           diamants
        Indice : indice en cours d'inspection (envoyer 0 lors de l'appel)
        I : Contiendra l'indice de la case contenant le diamant : valeur de
            retour
*/
trouverPositionElement([T|_], Code, Indice, I) :- T=Code, I is Indice, !.
trouverPositionElement([_|R], Code, Indice, I) :- Indice2 is Indice + 1,
    trouverPositionElement(R, Code, Indice2, I), !.


/*
        plus court chemin pour atteindre une destination Xd, Yd a partir
           des coordonnées du joueur Xp, Yp
        Pas de gestion des obstacles
*/
pccDestination(Start, Finish, L, Size, D) :-
                solve(Start, Soln, L, Size, Finish),
                not(Soln = []),
                reverse(Soln, Oo),
                nth0(1, Oo, ToGo),
                quelleSuivante(Start, ToGo, D, Size).

/*
  quelle est la direction pour la case suivante ? : C : courant, N : next,
     D : direction
```

```
  0 : gauche
  1 : droite
  2 : haut
  3 : bas

  quelleSuivante(+Current, +Next, -Direction, +Size)
*/
quelleSuivante(C, N, 1, _) :-
  C2 is C + 1,
  N = C2,
  !.
quelleSuivante(C, N, 0, _) :-
    C2 is C - 1,
  N = C2,
  !.
quelleSuivante(C, N, 3, Size) :-
    C2 is C + Size,
  N = C2,
  !.
quelleSuivante(C, N, 2, Size) :-
    C2 is C - Size,
  N = C2,
  !.
%quelleSuivante(_, _, D, _) :- ecrire('ALEATOIRE !'), D is random(5).
quelleSuivante(_, _, 4, _) :-
  ecrire('Bloque !').

/*
trouverPlusProcheDiamant(+positionJoueur, +Map, +Size, -Destination)
*/

trouverPlusProcheDiamant(PosPlayer, Ldepart, Size, D) :-
    %ecrire(Size),

    suppressionDiamantsInnaccessible(Ldepart, PosPlayer, Size, L),

      XPlayer is PosPlayer mod Size,
      YPlayer is PosPlayer // Size,
      findall(
            Couple,
            (
                  nth0(Indice, L, 2),
                  XDiamant is Indice mod Size,
                  YDiamant is Indice // Size,
                  Distance is ( (XDiamant-XPlayer)*(XDiamant-XPlayer) +
                      (YDiamant-YPlayer)*(YDiamant-YPlayer) ),
                  Couple = [Distance, Indice]
            ),
            Succs
      ),
      not(Succs = []),
      sort(Succs, S),
    tenterDeplacement(PosPlayer, S, L, Size, D),
    !.
      %S = [ [_|[Indice]] | _ ].
```

```
% un monstre nous bloque le passage
trouverPlusProcheDiamant(PosPlayer, L, Size, D) :-
    nth0(Indice, L, 12),
    distance(PosPlayer, Indice, Size, Dist),
    trouverCompromis(PosPlayer, Indice, Dist, Size, L, D).

/*
distance euclidienne entre deux points A et B
distance(+A, +B, +Size, -Distance)
*/
distance(A, B, Size, Distance) :-
    Xa is A mod Size,
    Ya is A // Size,
    Xb is B mod Size,
    Yb is B // Size,
    Distance is ( (Xb-Xa)*(Xb-Xa) + (Yb-Ya)*(Yb-Ya) ).

/*
compromis pour fuir un monstre, on cherche à s'en éloigner a tout prit
    sachant qu'on a pas de passage vers le diamant :
  trouverCompromis(+PosPlayer, +PositionMonstre, +DistanceMonstre, +Size,
      +L, -DirectionAPrendre)
*/
% en bas
trouverCompromis(P, PM, DM, Size, L, 3) :-
    numCaseBas(P, Size, B),
    nth0(B, L, El),
    (El = 0 ; El = 1 ),
    distance(B, PM, Size, DistanceMonstre),
    DistanceMonstre >= DM,
    !.
% a gauche
trouverCompromis(P, PM, DM, Size, L, 0) :-
    numCaseGauche(P, Size, B),
    nth0(B, L, El),
    (El = 0 ; El = 1 ),
    distance(B, PM, Size, DistanceMonstre),
    DistanceMonstre >= DM,
    !.
% a droite
trouverCompromis(P, PM, DM, Size, L, 1) :-
    numCaseDroite(P, Size, B),
    nth0(B, L, El),
    (El = 0 ; El = 1 ),
    distance(B, PM, Size, DistanceMonstre),
    DistanceMonstre >= DM,
    !.
% en haut
trouverCompromis(P, PM, DM, Size, L, 2) :-
    numCaseHaut(P, Size, B),
    nth0(B, L, El),
    (El = 0 ; El = 1 ),
    distance(B, PM, Size, DistanceMonstre),
    DistanceMonstre >= DM,
    !.
```

```
tenterDeplacement(_, [], _, _, _) :- !, fail.

tenterDeplacement(From, [[_|[To]]|_], L, Size, D) :-
  pccDestination(From, To, L, Size, D),
  !.

tenterDeplacement(From, [_|Others], L, Size, D) :-
  tenterDeplacement(From, Others, L, Size, D).

/*
        renvoie la direction à prendre pour atteindre le diamant
*/
trouverPpcDiamant(L, Pos, Size, D) :-
    trouverPlusProcheDiamant(Pos, L, Size, D).

/*
        renvoie la direction à prendre pour atteindre le diamant
*/
trouverPpcSortie(L, Pos, Size, D) :-
    nth0(I, L, 17),
    pccDestination(Pos, I, L, Size ,D).



/*
        trouve ou aller s il reste des diamants
        Xplayer, Yplayer : position du joueur
        L : Map du jeux
        Size : longeur d'une ligne du jeux
        Direction : direction à prendre
        CanGotoExit : peux aller a la sortie
*/
trouverOuAller(L, Pos, Size, Direction, CanGotoExit) :-
    CanGotoExit = 0,
    trouverPpcDiamant(L, Pos, Size, Direction),!.

% aller a la sortie
trouverOuAller(L, Pos, Size, Direction, CanGotoExit) :-
    CanGotoExit = 1,
    trouverPpcSortie(L, Pos, Size, Direction),!.


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

  * 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
  * VPx et VPy représente le périmètre de vue actuel utilisé pour générer L
  * NewVPx et NewVPy représente le nouveau périmètre de vue du mineur
*/


/* Situation 2 */

move( L, _, _, _, Size, _, Dx, Dy, _, _, -1, _ ) :-
    nth0(P, L, 10),
    situations2(L, P, Size, Z),
    P = Z,
    dir( 4, Dx, Dy ), !.

move( L, _, _, _, Size, _, Dx, Dy, _, _, -1, _ ) :-
    nth0(P, L, 10),
    situations2(L, P, Size, Goal),
    pccDestination(P, Goal, L, Size, D),
    dir( D, Dx, Dy ), !.
/* Fin situation 2 */


/*
  Si size devient trop importante tentative
  de mouvement aléatoire
*/
move( _, _, _, _, Size, _, Dx, Dy, _, _, -1, _ ) :-
    Size > 18,
    %D is random(4),
    dir( 1, Dx, Dy ), !.


move( L, _, _, Pos, Size, _, Dx, Dy, _, _, -1, _ ) :-
    situations(L, Pos, Size, Direction),
    dir( Direction, Dx, Dy ), !.

move( L, _, _, Pos, Size, CanGotoExit, Dx, Dy, _, _, -1, _ ) :-
    CanGotoExit = 1,
    eviterMonstre(L, Size, L2),
    trouverOuAller(L2, Pos, Size, Direction, CanGotoExit),
    dir( Direction, Dx, Dy ),
    !.

move( L, _, _, Pos, Size, CanGotoExit, Dx, Dy, _, _, -1, _) :-
    member( 2, L ),
    eviterMonstre(L, Size, L2),
    trouverOuAller(L2, Pos, Size, Direction, CanGotoExit),
    dir( Direction, Dx, Dy ),
    !.
```

```
move( _, _, _, _, _, _, _, _, Vx, Vy, Vx1, Vy1 ) :- Vx1 is Vx+1, Vy1 is
    Vy+1.


suppressionDiamantsInnaccessible(L, Pos, Size, L3) :-

  findall(Indice,(nth0(Indice, L, 2)),ListeIndicesDiamants),
  not(ListeIndicesDiamants = []),

  supprimerDiamantsEncercles(L, Size, ListeIndicesDiamants, LI),
  replaceAll(L, LI, L2, 9),

  supprimerDiamantsBloques(L2, Pos, Size, L3),
  !.

suppressionDiamantsInnaccessible(L, _, _, L).

isQueMur([]) :- !.
isQueMur([T|R]) :- T =< 9 , T >= 4, isQueMur(R),!.

rechercheIndiceLigneMur([], _, _) :- fail, !.
rechercheIndiceLigneMur([T|_], CPT, CPT) :-
        isQueMur(T),!.
rechercheIndiceLigneMur([_|R], I, CPT) :-
        CPT2 is CPT+1, rechercheIndiceLigneMur(R, I, CPT2),!.

listeInt(Debut, Debut, [Debut]):-!.
listeInt(Debut, Fin, [Debut|R]) :- Debut2 is Debut+1, listeInt(Debut2, Fin,
    R),!.

supprimerDiamantsBloques(L, Pos, Size, LFin) :-
  getLignes(L, Size, L2),
  rechercheIndiceLigneMur(L2, I, 0),
  I > 0, length(L2, LongL2), LongL21 is LongL2 - 1, I < LongL21,
  IDebMur is (Size * I),
  IFinMur is (IDebMur + Size),
  nettoyerLMurBloque(Pos, IDebMur, IFinMur, L, LFin),!.

supprimerDiamantsBloques(L, _, _, L) :- !.

nettoyerLMurBloque(Pos, ID, _, L, L2) :-
  length(L, Long),
  Pos < ID,
  Long2 is Long-1,
  listeInt(ID, Long2, LInt),
  replaceAll(L, LInt, L3, 9),L2=L3,!.
nettoyerLMurBloque(Pos, ID, IF, L, L2) :-
  Pos > ID,
  listeInt(0, IF, LInt),
  replaceAll(L, LInt, L3, 9),L2=L3,!.


% Betement on regarde si le diamant est encerclé
supprimerDiamantsEncercles(_, _, [], []):-!.
supprimerDiamantsEncercles(L, Size, [T|R], [T|R2]) :-
        C1t is T - Size, getElement(L, C1t, C1),
```

```
        C2t is T - 1, getElement(L, C2t, C2),
        C3t is T + 1, getElement(L, C3t, C3),
        C4t is T + Size,write(C4t), getElement(L, C4t, C4),
        (C1 =< 9 , C1 >= 4, C2 =< 9 , C2 >= 4, C3 =< 9 , C3 >= 4, C4 =< 9 ,
            C4 >= 4),
        supprimerDiamantsEncercles(L, Size, R, R2),
!.
supprimerDiamantsEncercles(L, Size, [_|R], R2) :-
        supprimerDiamantsEncercles(L, Size, R, R2),!.

/**
Ecrit dans un fichier indiqué en paramètre
@profil : ecrireFile( +T , +NomFichier )
**/
ecrireFile( T , File ) :- open( File , append, L ), write( L, T ), nl( L ),
    close( L ).

/**
Ecrit dans le fichier trace.txt
**/
ecrire( T ) :- open( 'trace.txt', append, L ), write( L, T ), nl( L ),
   close( L ).


/*
  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.
```

5.2 eviterMonstre.pl

```
:- module(eviterMonstre, [eviterMonstre/3]).

/**
* Retourne l'indice du monstre présent dans le champs de vision
* @profil : getIndiceMonstre(+L, +Size, -Indice)
**/
getIndiceMonstre(L, _, Indice) :-
  nth0(Indice, L, 11),!.
getIndiceMonstre(L, _, Indice) :-
  nth0(Indice, L, 12),!.

/**
* Entoure le monstre de bloc
* et retourne le nouveau champs de vision du mineur
* @profil : entourerMonstres(+L, +Size, -L2).
**/
entourerMonstres(L, Size, L2) :-

  getIndiceMonstre(L, Size, I),

  % Indices à 2 cases
  H1 is I - Size * 2,
  H2 is I - Size - 1,
  H3 is I - Size + 1,
  H4 is I - 2,
  H5 is I + 2,
  H6 is I + Size - 1,
  H7 is I + Size + 1,
  H8 is I + Size * 2,

  % Indices à 1 case
  C1 is I - Size,
  C2 is I - 1,
  C3 is I + 1,
  C4 is I + Size,

  verifierIndices(L, [H1,H2,H3,H4,H5,H6,H7,H8,C1,C2,C3,C4], Indices),

  verifierIndicesC1(L, Indices, Indices2, Size),
  verifierIndicesC2(L, Indices2, Indices3, Size),
  verifierIndicesC3(L, Indices3, Indices4, Size),
  verifierIndicesC4(L, Indices4, Indices5, Size),
  replaceAll(L, Indices5, L2, 9),

  !.

/**
* Empeche de remplacer les élements du décors (autour du monstre) non
    accecibles par un mur
**/
verifierIndices(_, [], []).
verifierIndices(L, [T|R], [T|R2]) :-
  caseAccessible(L, T), verifierIndices(L, R, R2),!.
verifierIndices(L, [_|R], [-1|R2]) :-
```

```
  verifierIndices(L, R, R2), !.

/**
* Empèche de placer un bloc sur la liste de manière incohérente.
* Exemple : Si le mineur est tout à droite de la carte, un Pos + 2 ramenera
    sur la ligne suivante.
* @profil : verifierIndicesCN(+L, +Cases, -R, + Size).
* C1 => PosMineur - Size
* C2 => PosMineur - 1
* C3 => PosMineur + 1
* C4 => PosMineur + Size
**/
verifierIndicesC1(L, [_,H2,H3,H4,H5,H6,H7,H8,C1,C2,C3,C4], R, _) :-
  not(caseAccessible(L, C1)),
  R = [-1,H2,H3,H4,H5,H6,H7,H8,C1,C2,C3,C4],!.

verifierIndicesC1(_, L2, L2, _).

verifierIndicesC2(L, [H1,H2,H3,_,H5,H6,H7,H8,C1,C2,C3,C4], R, _) :-
  not(caseAccessible(L, C2)),
  R = [H1,H2,H3,-1,H5,H6,H7,H8,C1,C2,C3,C4],!.

verifierIndicesC2(_, L2, L2, _).

verifierIndicesC3(L, [H1,H2,H3,H4,_,H6,H7,H8,C1,C2,C3,C4], R, _) :-
  not(caseAccessible(L, C3)),
  R = [H1,H2,H3,H4,-1,H6,H7,H8,C1,C2,C3,C4],!.

verifierIndicesC3(_, L2, L2, _).

verifierIndicesC4(L, [H1,H2,H3,H4,H5,H6,H7,_,C1,C2,C3,C4], R, _) :-
  not(caseAccessible(L, C4)),
  R = [H1,H2,H3,H4,H5,H6,H7,-1,C1,C2,C3,C4],!.

verifierIndicesC4(_, L2, L2, _).


/**
* Prédicat principal
* @profil : eviterMonstre(+L, +Size, -L2)
**/
eviterMonstre(L, Size, L2) :-
  entourerMonstres(L, Size, L2),!.
eviterMonstre(L, _, L) :- !.
```

5.3 outilsCarte.pl

```
:- module(outilsCarte, [
                caseExist/2,
                            getElement/3,
                dernierElement/2,
                            caseAccessible/2,
                            caseDanger/2,
                            numCaseHaut/3,
                            numCaseBas/3,
                            numCaseGauche/3,
                            numCaseDroite/3,
                            replaceElement/4,
                            replaceAll/4,
                retourneElementFromTo/4,
                getLignes/3
                          ]
).

/**
* Dernier élement d'une liste
**/
dernierElement(L, E) :- reverse(L, [E|_]).

/**
* Permet de vérifier si une case est existante
* @profil : caseExist(+L, +Indice)
**/
caseExist([_|_], 0).
caseExist([_|R], Indice) :-
  Indice > 0,
  I2 is Indice - 1,
  caseExist(R, I2).

/**
* Permet de récupérer la valeur de l'élément d'indice Indice dans la liste
   L
* @profil : getElement(+L, +Indice, ?Element)

getElement([], _, _) :- fail.
getElement([T|_], 0, Element) :- Element is T.
getElement([T|R], Indice, Element) :- Indice > 0, I2 = Indice - 1,
   getElement(R, I2, Element).
**/
getElement(L, Indice, Element) :- length(L, Long), Indice > Long, Element =
    9,!.
getElement(_, Indice, Element) :- Indice < 0, Element = 9,!.
getElement(L, Indice, Element) :- nth0(Indice, L, E), Element is E.

/**
* Permet de savoir si une case est accecible (case vide ou herbe)
* @profil : caseAccessible(+L, +Indice)
**/
caseAccessible(L, Indice) :- getElement(L, Indice, Element), Element =< 1,
   !.
```

```
caseAccessibleOuDiamant(L, Indice) :- getElement(L, Indice, Element),
    Element =< 2, !.

/**
* Permet de savoir si une case est un danger
* @profil : caseDanger(+L, +Indice)
**/
caseDanger(L, Indice) :- getElement(L, Indice, Element), Element = 11, !.
caseDanger(L, Indice) :- getElement(L, Indice, Element), Element = 12, !.


/**
* Permet de connaitre le numéro de la case du dessus
* @profil : numCaseHaut(+N, +Size, -Ret)
**/
numCaseHaut(N, Size, Ret) :-
        Ret is N - Size.

/**
* Permet de connaitre le numéro de la case du dessus
* @profil : numCaseBas(+N, +Size, -Ret)
**/
numCaseBas(N, Size, Ret) :-
        Ret is N + Size.

/**
* Permet de connaitre le numéro de la case de gauche
* @profil : numCaseGauche(+N, +Size, -Ret)
**/
numCaseGauche(N, Size, Ret) :-
        Col is N mod Size,
        Col \= 0,
        Ret is N - 1.

/**
* Permet de connaitre le numéro de la case de droite
* @profil : numCaseDroite(+N, +Size, -Ret)
**/
numCaseDroite(N, Size, Ret) :-
        Col is ( N mod Size ) + 1,
        Col \= Size,
        Ret is N + 1.


/**
* Remplace la case d'incide Indice (commence à 0) dans L par la valeur
    Valeur
* retourne la nouvelle liste dans NewL
**/
replaceElement([_|R], 1, Valeur, [Valeur|R]).
replaceElement([T|R], Indice, Valeur, [T|R2]) :-
                I2 is Indice - 1,
                replaceElement(R, I2, Valeur, R2),!.

/**
```

```
* Remplace dans L tous les élements d'indice compris dans la liste I
   exemple : [indice0, indice5, ...]
* par la valeur V, ne remplace pas la case contenant le joueur
* Puis renvoi la liste modifié sous L2
* @profil : replaceAll(+L, +I, -L2, +V)
**/
replaceAll(L, I, L2, V) :-
            sublist( <(-1), I, I2),
             sort(I2, I3),
             replaceAll2(L, I3, L2, V),!.
replaceAll2(L, [], L, _):-!.
replaceAll2([], _, [], _):-!.
replaceAll2([_|R], [0|RI], [V|R2], V) :-
                maplist(plus(-1), RI, I2),
                replaceAll2(R, I2, R2, V),!.
replaceAll2([T|R], I, [T|R2], V) :-
             maplist(plus(-1), I, I2),
             replaceAll2(R, I2, R2, V).


/**
* Retourne les élements d'une liste de l'indice D à l'indice A
* Indice commencant à 1
* retourne la liste des élements dans L2
* @profil : retourneElementFromTo(L, D, A, L2)
**/
retourneElementFromTo(L, D, A, L2) :-
        D < 0, retourneElementFromTo(L, 1, A, L2),!.
retourneElementFromTo(L, D, A, L2) :-
        A < 0, retourneElementFromTo(L, D, 1, L2),!.
retourneElementFromTo(L, D, D, L2) :-
        nth1(D, L, E),
        L2 = [E],!.
retourneElementFromTo(L, D, A, L2) :-
        retourneElementFromTo2(L, D, A, L2),!.

retourneElementFromTo2([T|_], 0, 0, [T]) :- !.
retourneElementFromTo2([T|R], 0, A, [T|R2]) :-
                            A2 is A - 1,
                            retourneElementFromTo2(R, 0, A2, R2),!.
retourneElementFromTo2([_|R], D, A, L2) :-
                    D2 is D-1,
                    A2 is A-1,
                    retourneElementFromTo2(R, D2, A2, L2),!.

/**
Retourne L Sous forme de liste de lignes
**/
nPremiersTermes(_, 0, []) :- !.
nPremiersTermes([T|R], N, [T|R2]) :- N2 is N-1, nPremiersTermes(R, N2, R2).
supprimerNPremiers(L, 0, L) :- !.
supprimerNPremiers([_|R], N, R2) :- N2 is N-1, supprimerNPremiers(R, N2,
   R2).

getLignes([], _, []) :- !.
getLignes(L, Size, [T|R2]) :-
```

```
        nPremiersTermes(L, Size, T),
        supprimerNPremiers(L, Size, L2),
        getLignes(L2, Size, R2).
```

5.4 plusCourtChemin.pl

```
:-module(plusCourtChemin, [solve/5]).
:-use_module('outilsCarte').
:-use_module('rochers').


/*
        passageX(+A, -B, +Size, +L)
*/

/*
 en haut
*/
% si ya rien en haut
passageH(A, B, Size, L, 1000) :-
    rocherMeTombeDessus(L, Size, A),
    !,
    %ecrire(A), ecrire(L), ecrire(Size),
    numCaseHaut(A, Size, B),
        getElement(L, B, El),
      ( El =< 3 ; El = 17 ).


passageH(A, B, Size, L, 1) :-
    numCaseHaut(A, Size, B),
        getElement(L, B, El),
      ( El < 3 ; El = 17 ).

/*
en bas
*/
passageB(A, B, Size, L, 1) :-
    numCaseBas(A, Size, B),
    getElement(L, B, El),
    ( El < 3 ; El = 17 ).

/*
a gauche
*/
passageG(A, B, Size, L, 500) :-
  pousserRocherGauche(5, A, L, Size),
  numCaseGauche(A, Size, B).

passageG(A, B, Size, L, 1) :-
    numCaseGauche(A, Size, B),
      getElement(L, B, El),
      ( El < 3 ; El = 17 ).

/*
a droite
*/
passageD(A, B, Size, L, 500) :-
    pousserRocherDroite(5, A, L, Size),
    numCaseDroite(A, Size, B).
```

```
passageD(A, B, Size, L, 1) :-
    numCaseDroite(A, Size, B),
      getElement(L, B, El),
      ( El < 3 ; El = 17 ).

s(A, B, Poids, L, Size) :-
    passageD(A, B, Size, L, Poids).

s(A, B, Poids, L, Size) :-
    passageH(A, B, Size, L, Poids).

s(A, B, Poids, L, Size) :-
    passageG(A, B, Size, L, Poids).

s(A, B, Poids, L, Size) :-
    passageB(A, B, Size, L, Poids).


h(A, Distance, B, Size) :-
    Xa is A mod Size,
    Ya is A // Size,
    Xb is B mod Size,
    Yb is B // Size,
    Distance is ( (Xb-Xa)*(Xb-Xa) + (Yb-Ya)*(Yb-Ya) ).


solve(Start, Solution, L, Size, Finish) :-
        nb_setval(openList, [[0, [Start]]]),
        nb_setval(closedList, []),
        astar(Solution, L, Size, Finish).
        %astar([[[Start], 0]], SolPath, L, Size, Finish).

 % plus de noeuds dans open list, pas de solutions !
 astar([], _, _, _) :-
      nb_getval(openList, []),
      !.

% noeud courant = finish -> solution trouvée
astar(Solution, _, _, Finish) :-
      nb_getval(openList, [[_, Solution]|_]),
      Solution = [Finish|_],
      !.

% recherhe de tous les fils de la meilleur sol
astar(Solution, L, Size, Finish) :-
      nb_getval(openList, [[Cout, MeilleurChemin]|AutresCheminsOuverts]),
      nb_getval(closedList, ClosedList),
      % meilleur chemin en tete de liste
      MeilleurChemin = [DernierNoeud|_],
      % mettre à jour l'open liste = faire sauter la tete
      nb_setval(openList, AutresCheminsOuverts),
      %write(AutresCheminsOuverts),
      % ajouter le noeud courant a la liste close
      append(ClosedList, [DernierNoeud], ListeFerme),
      nb_setval(closedList, ListeFerme),
      % trouver tous les enfants
```

```
        findall(
                Couple,
                (
                        s(DernierNoeud, S, P, L, Size) ,
                        not(member(S, ListeFerme)),
          Couple = [S, P] % noeud et poids
                ),
                Succs
        ),
        %write(Succs),
        % les ajouter a l'open list
        ajouterChemin(MeilleurChemin, Cout, Succs, Size, Finish),
        % on continue a tracer les chemins
        astar(Solution, L, Size, Finish).


ajouterChemin(_, _, [], _, _) :- !. % plus rien à ajouter

ajouterChemin(MeilleurChemin, Cout, [Couple|NoeudsRestants], Size, Finish)
    :-
    % récup noeud et poids
     Couple = [S|CoutDeplacement],
       % calcul du nouveau cout
       h(S, Distance, Finish, Size),
     %ecrire(CoutDeplacement),
       NouveauCout is Cout + Distance + CoutDeplacement,
       %write(S), write('  '), write(NouveauCout), write('\n'),
       % ajout a la liste ouverte
       append([S], MeilleurChemin, NouveauChemin),
       %write(NouveauChemin), write('\n'),
       ajouterListeOuverte(NouveauChemin, NouveauCout),
       % on ajoute les autres noeuds possibles
       ajouterChemin(MeilleurChemin, Cout, NoeudsRestants, Size, Finish).


ajouterListeOuverte(NouveauChemin, NouveauCout) :-
        nb_getval(openList, OpenList),
        %write('liste ouverte actuelle : '), write(OpenList), write('\n'),
        %write('Nouveau chemin et cout : '), write(NouveauChemin), write('
            '), write(NouveauCout), write('\n'),
        append(OpenList, [[NouveauCout, NouveauChemin]],
            NouvelleListeOuverte),
        sort(NouvelleListeOuverte, NouvelleListeOuverteTriee),
        %write('Liste apres ajout : '), write(NouvelleListeOuverteTriee),
            write('\n'),
        nb_setval(openList, NouvelleListeOuverteTriee).
```

5.5 rochers.pl

```
:- module(rochers, [
                              pousserRocherDroite/4,
                              pousserRocherGauche/4,
                rocherMeTombeDessus/3
                             ]
).
:-use_module('outilsCarte').


/**
Pousser les rochers
Methode complète.
jusqu'a "Profondeur" rochers
**/

/*
  A gauche
*/

% profondeur max atteinte, a-t-on un vide ?
pousserRocherGauche(0, C, L, Size) :-
    numCaseGauche(C, Size, G),
    getElement(L, G, 0),
    !.
% un vide derriere les rochers
pousserRocherGauche(_, C, L, Size) :-
    numCaseGauche(C, Size, G),
    getElement(L, G, 0),
    !.
% un rocher, regarder encore derrière !
pousserRocherGauche(Profondeur, C, L, Size) :-
    numCaseGauche(C, Size, G),
    getElement(L, G, 3),
    Profondeur2 is Profondeur - 1,
    pousserRocherGauche(Profondeur2, G, L, Size).


/*
  A droite
*/

pousserRocherDroite(0, C, L, Size) :-
    numCaseDroite(C, Size, D),
    getElement(L, D, 0),
    !.

pousserRocherDroite(_, C, L, Size) :-
    numCaseDroite(C, Size, D),
    getElement(L, D, 0),
    !.

pousserRocherDroite(Profondeur, C, L, Size) :-
    numCaseDroite(C, Size, D),
    getElement(L, D, 3),
```

```
    Profondeur2 is Profondeur - 1,
    pousserRocherDroite(Profondeur2, D, L, Size).


/*
  En haut
*/


rocherMeTombeDessus(L, Size, N1) :-
  numCaseHaut(N1, Size, N2),
  numCaseHaut(N2, Size, N3),
  numCaseHaut(N3, Size, N4),
  getElement(L, N2, El1),
  getElement(L, N3, El2),
  getElement(L, N4, El3),
  (El1 = 3 ; El2 = 3 ; El3 = 3 ; El1 = 2 , El2 = 2 ; El3 = 2).
```

5.6 situations.pl

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

/**
* Repertories les différentes situations
* que le mineur peut rencontrer, et essaye, si une situation est trouvée de
* la résoudre.
* (cf : Fichier Situations.
**/

situations(L, Pos, Size, Direction) :-
        situation1(L, Pos, Size, Direction),!.
situations(L, Pos, Size, Direction) :-
        situation1_2(L, Pos, Size, Direction),!.
situations(L, Pos, Size, Direction) :-
        situation_rocher(L, Pos, Size, Direction),!.

/*
*************************************************************************
Situation 1
Verifie si il y a un monstre en Mineur - Size * 2 et un diamant en Mineur -
    Size
*************************************************************************
*/
situation1(L, Pos, Size, Direction) :-

        Diamant is Pos - Size,
        getElement(L, Diamant, EDiamant),
        EDiamant = 2,

        Monstre is Pos - Size * 2,
        caseDanger(L, Monstre),

        trouverDirectionSituation1(L, Pos, Size, Direction),!.

trouverDirectionSituation1(L, Pos, _, Direction) :-
        Droite1 is Pos + 1,
        Droite2 is Pos + 2,
        Gauche1 is Pos - 1,
        Gauche2 is Pos - 2,
        trouverDirectionSituation1E2(L, Droite1, Droite2, Gauche1, Gauche2,
            Direction),!.
trouverDirectionSituation1E2(L, D1, D2, _, _, Direction) :-
        caseAccessible(L, D1),
        caseAccessible(L, D2),
        Direction = 1,!.
trouverDirectionSituation1E2(L, _, _, G1, G2, Direction) :-
        caseAccessible(L, G1),
        caseAccessible(L, G2),
        Direction = 0,!.

/**
```

```
Situation 1, permet de déplacement de deux cases permettant de faire tomber
    le diamant
**/
situation1_2(L, Pos, Size, Direction) :-

        Diamant is Pos - Size + 1,
        getElement(L, Diamant, EDiamant),
        EDiamant = 2,

        Monstre is Pos - Size * 2 + 1,
        caseDanger(L, Monstre),

        Vide is Pos + 1,
        getElement(L, Vide, EVide),
        EVide = 0,

        Direction = 0,!.
situation1_2(L, Pos, Size, Direction) :-

        Diamant is Pos - Size - 1,
        getElement(L, Diamant, EDiamant),
        EDiamant = 2,

        Monstre is Pos - Size * 2 - 1,
        caseDanger(L, Monstre),

        Vide is Pos - 1,
        getElement(L, Vide, EVide),
        EVide = 0,

        Direction = 1,!.


/**
* SITUATION 2 Rochers :
* Si pos - Size est accecible et Pos - 2*Size est un rocher alors :
* - Le mineur choisit une direction dans un ordre de possibilité :
* - A Droite, En bas, A gauche
* La possiblité de déplacement est accepté si les 2 cases dans la direction
    sont accecible.
**/
situation_rocher(L, Pos, Size, Direction) :-

  Rocher is Pos - 2*Size + 1,
  CaseVide is Pos + 1, CaseVide2 is Pos - Size + 1, getElement(L,
      CaseVide2, EV2), EV2 = 0,
  getElement(L, Rocher, ER), ER = 3,
  getElement(L, CaseVide, EV), EV = 0,
  H is Pos - Size,
  G is Pos - 1,
  choisirDirRocher(L, H, G, 0, Direction),!.

situation_rocher(L, Pos, Size, Direction) :-

  Rocher is Pos - 2*Size - 1,
```

```
  CaseVide is Pos - 1, CaseVide2 is Pos - Size - 1, getElement(L,
      CaseVide2, EV2), EV2 = 0,
  getElement(L, Rocher, ER), ER = 3,
  getElement(L, CaseVide, EV), EV = 0,
  H is Pos - Size,
  G is Pos - 1,
  choisirDirRocher(L, H, G, 0, Direction),!.

choisirDirectionRocher(L, _, _, D, 1) :-
  caseAccessibleOuDiamant(L, D),!.
choisirDirectionRocher(L, _, _, _, 3) :-
  caseAccessibleOuDiamant(L, _),!.
choisirDirectionRocher(L, _, G, _, 0) :-
  caseAccessibleOuDiamant(L, G),!.
choisirDirectionRocher(L, H, _, _, 2) :-
  caseAccessibleOuDiamant(L, H),!.
```

5.7 situations2.pl

```
:-module(situations2, [situations2/4]).
:-use_module('outilsCarte').


presenceMonstre(L, I, P) :-
    P > 0,
    I2 is I + P,
    ( nth0(I2, L, 11) ; nth0(I2, L, 12) ),
    !.

presenceMonstre(L, I, P) :-
    P > 0,
    P2 is P - 1,
    presenceMonstre(L, I, P2).



/*
      Phase 1 : le monstre se ballade tranquille
*/
situations2(L, _, Size, Objectif) :-

    %ecrire('---'), ecrire(L), ecrire(Size),

    findall(
      Case,
      (
          (
            nth0(Case, L, 1) ;
            nth0(Case, L, 0) ;
            nth0(Case, L, 10)
          ),

          numCaseDroite(Case, Size, CDroite),
          nth0(CDroite, L, 1),

          numCaseHaut(CDroite, Size, CHautDroite),
          nth0(CHautDroite, L, 3),

          numCaseBas(Case, Size, CBas),
          nth0(CBas, L, 3),

          numCaseBas(CDroite, Size, CBasDroite),
          (
            nth0(CBasDroite, L, 0)  ;
            nth0(CBasDroite, L, 11) ;
            nth0(CBasDroite, L, 12)
          ),

          % monstre sur la même ligne a 1..7
          (
            presenceMonstre(L, CBas, 7) ;
            presenceMonstre(L, Case, 7)
          )
```

```
      ),
      ListeCase
    ),
    not( ListeCase = [] ),
    !,
    ListeCase = [O|_],
    prendreDecision(O, L, Size, Objectif).

/*
      Phase 2 : le piège est tendus
      cas gauche
*/

situations2(L, Pos, Size, Objectif) :-

    findall(
      Case,
      (
          (
            nth0(Case, L, 1) ;
            nth0(Case, L, 0) ;
            nth0(Case, L, 10)
          ),

          numCaseDroite(Case, Size, CDroite),
          (
            nth0(CDroite, L, 10) ;
            nth0(CDroite, L, 0)
          ),

          numCaseHaut(CDroite, Size, CHautDroite),
          nth0(CHautDroite, L, 3),

          numCaseBas(Case, Size, CBas),
          nth0(CBas, L, 3),

          % monstre sur la même ligne a 2..3
          Indice is CBas + 3,
          Indice2 is CBas + 2,
          ( nth0(Indice, L, 11) ; nth0(Indice, L, 12) ;  nth0(Indice2, L,
              11) ; nth0(Indice2, L, 12) )

      ),
      ListeCase
    ),
    not( ListeCase = [] ),
    Objectif is Pos - 1,
    nth0(Objectif, L, 0),
    !.


/*
      Phase 2 : le piège est tendus
      cas haut
*/
```

```
situations2(L, Pos, Size, Objectif) :-

    findall(
      Case,
      (
          (
            nth0(Case, L, 1) ;
            nth0(Case, L, 0) ;
            nth0(Case, L, 10)
          ),

          numCaseDroite(Case, Size, CDroite),
          (
            nth0(CDroite, L, 10) ;
            nth0(CDroite, L, 0)
          ),

          numCaseHaut(CDroite, Size, CHautDroite),
          nth0(CHautDroite, L, 3),

          numCaseBas(Case, Size, CBas),
          nth0(CBas, L, 3),

          % monstre sur la même ligne a 2..3
          Indice is CBas + 3,
          Indice2 is CBas + 2,
          ( nth0(Indice, L, 11) ; nth0(Indice, L, 12) ;  nth0(Indice2, L,
              11) ; nth0(Indice2, L, 12) )

      ),
      ListeCase
    ),
    not( ListeCase = [] ),
    Objectif is Pos - Size,
    ( nth0(Objectif, L, 0) ; nth0(Objectif, L, 0) ),
    !.



/*
Attendre sous le rocher
*/

situations2(L, Pos, Size, Objectif) :-

    findall(
      Case,
      (
          (
            nth0(Case, L, 1) ;
            nth0(Case, L, 0) ;
            nth0(Case, L, 10)
          ),

          numCaseDroite(Case, Size, CDroite),
```

```
            (
              nth0(CDroite, L, 10) ;
              nth0(CDroite, L, 0)
            ),

            numCaseHaut(CDroite, Size, CHautDroite),
            nth0(CHautDroite, L, 3),

            numCaseBas(Case, Size, CBas),
            nth0(CBas, L, 3),

            % monstre sur la même ligne a 1..7
            (
              presenceMonstre(L, CBas, 7) ;
              presenceMonstre(L, Case, 7)
            )

        ),
        ListeCase
    ),
    not( ListeCase = [] ),
    Objectif is Pos.

/*
Prise de décision pour la phase 1
*/

% se placer sur la case Objectif
prendreDecision(Objectif, L, _, Objectif) :-
    ( nth0(Objectif, L, 0) ; nth0(Objectif, L, 1) ),
    !.


% patienter avant que le monstre soit en position : aller retour G/D
prendreDecision(Objectif, L, Size, Objectif) :-
    numCaseBas(Objectif, Size, CBas),
    CBas2 is CBas + 4,
    not(nth0(CBas2, L, 11)),
    not(nth0(CBas2, L, 12)),
    !.


% Monstre en position : on se met sous le cailloux
prendreDecision(Objectif, _, _, Goal) :-
    Goal is Objectif + 1,
    !.
```

Table des matières