IA41

INTELLIGENCE ARTIFICIELLE

RAPPORT PROJET

DELIRIUM 2

REALISATION D'UNE INTELLIGENCE ARTIFICIELLE DANS UN JEU VIDEO

ZIANE CLEMENT

KOUETE WILLY GAEL

DUONG TONY

PRINTEMPS 2015

SOMMAIRE

I. DESCRIPTION DU PROJET

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 réalisation d'une intelligence artificielle pour le jeu vidéo Delirium 2. Dans ce jeu, le mineur se déplace 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 diamants 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 nécessaires, éviter les pièges et les ennemis qui l'entourent 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 présenter, à travers ce rapport, l'étude menée préalablement et les différentes solutions que nous envisagions. Par la suite, nous détaillerons la solution que nous avons choisi d'implémenter. Ensuite nous illustreront cette solution avec des jeux d'essais. Enfin nous soulèverons les différents problèmes rencontrés et la façon dont nous les avons traités.

II. PHASE D'ETUDE

Dans cette partie, nous allons détailler la phase de réflexion qui a précédé le développement et en particulier quatre points : comment nous avons compris le sujet, la problématique qui en découle, et les deux solutions nécessaires.

1. Analyse de l'environnement

La première phase de notre projet fut de comprendre le fonctionnement de l'environnement qui nous a permis de développer notre agent d'intelligence artificielle. En effet, nous avons passé la première semaine à comprendre comment le fonctionnement de Ipseity. Nous avons ensuite appris comment générer de nouvelles cartes afin de pouvoir effectuer des jeux d'essais, comment modifier le périmètre de vision de notre agent pour optimiser notre système. Et enfin nous avons analysé comment les fonctions Prolog étaient appelées par l'environnement et avons constaté qu'elles sont appelées à chaque tour et non de façon récursive ce qui nous a permis d'adapter notre conception.

2. Problématique

Dans un deuxième temps, nous avons extrait de notre analyse du sujet et de l'environnement la problématique guidant la conception de notre intelligence artificielle. Nous avons découpé cette problématique en deux parties que nous pouvons synthétiser avec les deux questions suivantes : "Comment trouver le plus court chemin pour récolter les diamants et atteindre le niveau supérieur ?" et "Comment anticiper les obstacles afin d'assurer à notre agent un chemin sans danger ?".

3. Recherche du chemin le plus court

Nous avons réfléchi aux divers solutions possibles pour répondre à la première problématique et nous avons découpé un cheminement

en trois étapes, la première étant de trouver la position des diamants les plus proches de notre agent, ensuite il faut générer le plus court chemin jusqu'à ces différentes positions, et une fois que le nombre de diamant récolté est suffisant pour monter de niveau, nous devons trouver le plus court chemin jusqu'à la porte de fin de niveau.

Tout d'abord nous nous sommes accordés à dire que, pour la première étape, une simple analyse du périmètre de vue était suffisante. Puis une fois l'analyse terminée, nous stockons les différentes positions dans une liste à part pour un traitement ultérieur. Nous avons également anticipé dans notre réflexion que le périmètre de vue de notre agent pouvait être limité et qu'aucun diamant n'apparaissent dedans.

Deuxièmement, nous devions trouver une solution à la recherche du plus court chemin, nous avions analysé l'environnement comme un graphe fini et nous avons décidé d'utiliser un algorithme de recherche du plus court chemin, de plus lors de nos cours nous avions étudié l'algorithme A* qui nous semblait parfaitement adapté à ce genre de situation.

Enfin nous avions pensé à répéter ces deux premières étapes à pour trouver et acheminer notre agent jusqu'à la porte de fin de niveau une fois que le nombre de diamants récoltés était suffisant.

4. Éviter les obstacles

Dans un deuxième temps, notre réflexion s'est portée sur la sécurité de notre agent durant son parcours du niveau.

Tout d'abord, nous avons répertorié les différentes difficultés que pouvait rencontrer notre agent en deux catégories, les obstacles mobiles tels que les monstres, les chutes de pierres et de diamantes et les obstacles statiques comme les pierres.

Ensuite nous avons commencé par réfléchir aux différentes manières d'éviter à notre agent une rencontre avec un monstre. Nous avions pu extraire de cette réflexion deux solutions envisageable. D'une part nous pouvions anticiper le déplacement des monstres en analysant leur algorithme de déplacement et ainsi générer un chemin sécurisé pour notre IA.

Enfin nous avons pensé à une solution pour éviter les obstacles pseudo-mobile, en effet les pierres peuvent passer d'un état statique à un état mobile et vice-versa, nous avons pensé à analyser à différents itérations du jeu, la position des pierres autour pour déterminer si elles tombent ou non, et esquiver les pierres dans le cas où celles-ci seraient en mouvement et à proximité de notre agent.

III. SOLUTIONS RETENUES

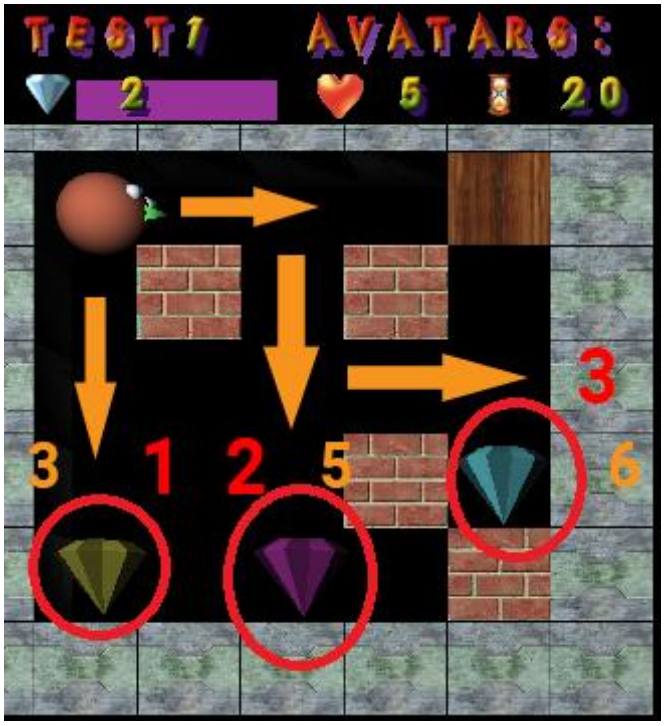
1. Recherche des diamants

Nous avons commencé à implémenter notre solution en développant une méthode permettant de retrouver les diamants les plus proches de notre agent (*diamant.pl*).

Tout d'abord nous vérifions qu'il y a bien des diamants dans le périmètre de vue de notre mineur, le périmètre de vue étant par défaut toute la carte du jeu. Si des diamants sont visibles par le mineur et accessibles, nous stockons leur position dans une liste. Cette liste sera analysée afin de trouver le diamant le plus proche. Pour chaque diamant, nous utilisons l'algorithme A* et gardons en mémoire la longueur du chemin. Cette longueur de chemin sera comparée entre tous les diamants et ainsi, nous pouvons trouver le diamant le plus proche.

La position de ce plus proche diamant sera utilisée par l'algorithme de recherche du plus court chemin (*astar.pl*) comme étant la prochaine destination de notre personnage.

Nous recommençons cette étape à chaque itération du jeu jusqu'à ce que notre mineur ait récolté assez de diamant pour pouvoir chercher la sortie et ainsi passer au niveau supérieur.



Dans cette figure, vous pouvez remarquer la longueur du chemin (*en orange*) ainsi que la priorité (*en rouge*) pour aller vers chaque diamant.

Dans ce cas particulier, il va prendre le chemin vers le diamant jaune (de longueur 3).

2. Recherche du chemin le plus court

A* est un algorithme reconnu de recherche du plus court chemin dans des graphes.

Il a été implémenté pour gérer les déplacements du mineur à partir d'un point de départ donné, d'un point d'arrivée et d'une liste représentant la carte du jeu (*astar.pl*).

A partir de ces éléments, l'algorithme étudie les directions possibles que peut prendre le mineur.

Voici l'algorithme détaillé de l'A* :

a. Ajouter la position de départ du mineur dans la liste ouverte avec un coût de 0.
b. Répétition :
    - Chercher parmi les quatre cases adjacentes à la case en cours celle dont le coût est le plus petit. Cette case devient la case actuelle.
    - On ajoute la position de la case en cours à la liste fermée (elle est considérée comme analysée).
    - Pour chacune des 4 cases adjacentes de la case en cours, nous faisons les tests suivants.
        i. Si c'est une case non traversable (rochers, murs) ou si c'est une case de la liste de la liste fermée, alors cette case est ignorée.
        ii. Sinon on l'ajoute dans la liste ouverte. Cette case fera partie d'une des cases fils de la case actuelle. Calculer le coût de cette case.
    - Nous trions la liste de manière croissante en fonction du coût.
c. Arrêt :
    - Si la destination finale a été trouvée, alors le chemin à partir de la case de départ vers cette case de destination est sauvegarde. Dans ce cas, le mineur prendra ce chemin pour aller vers la destination.

- Sinon si la liste ouverte est vide, cela signifie qu'un chemin n'a pas été trouvé. Dans ce cas, la liste définissant le chemin sera vide. Le mineur aura deux comportements possibles à cet état : soit un danger imminent est détecté (l'approche d'un monstre à proximité lors de la prochaine itération, ou un rocher en chute se situant juste au-dessus de celui-ci) auquel cas il cherchera à se placer en position de sécurité ; soit aucun danger n'est détecté auquel cas il ne bougera pas pendant cette itération du jeu.

REMARQUE IMPORTANTE

Notre mineur ne prendra seulement que la première case du chemin trouvé (donc premier élément de la liste). Cette première case déterminera l'action du mineur (Droite = 1 ; Haut = 2 ; Gauche = 3 ; Bas = 4). Ensuite, l'algorithme de recherche est lancé à la prochaine itération du jeu.

Cette manière de procéder possède un avantage certain : à chaque itération de jeu, nous pouvons prendre en compte les modifications de l'environnement (monstres, rochers). Si nous n'avions qu'à suivre un seul chemin trouvé jusqu'au bout, il aurait été difficile de prévoir si un rocher tombera sur notre mineur ou si notre mineur croisera un monstre lors de son parcours.

3. Gestion des monstres

Durant son évolution, le mineur devra éviter différents monstres qui cherchent à le manger. Notre mineur meurt s'il se situe sur l'une des quatre cases adjacentes d'un monstre. Nous avons dû trouver une astuce intelligente afin d'assurer à notre personnage la possibilité de récolter les diamants nécessaires et d'atteindre la sortie sans se faire manger par les monstres.

Tout d'abord, nous stockons la carte à un instant *t-1* pour la comparer à un instant *t* afin de prévoir le sens du déplacement des monstres. Une fois le sens du déplacement des monstres prévus, nous créons une liste temporaire qui serait la carte à l'instant *t+1* avec les futures positions des monstres. Nous anticipons ainsi leur déplacement.

Les cases autour d'un monstre dans cette carte seront considérées comme étant une case impossible à atteindre et ainsi notre mineur ne la prend pas en compte dans son chemin et s'assure un chemin sans danger pour atteindre sa cible *(monster.pl)*.

Voici l'algorithme des monstres :

- Si c'est la première itération du jeu, la direction prise sera dans l'ordre {Haut, Droite, Bas, Gauche, NeBougePas}. *(s'il ne peut pas aller en haut, il cherchera à aller à droite, s'il ne peut pas aller à droite, il cherchera à aller en bas, etc…)*
- Sinon la direction prise sera en fonction de sa direction précédente. Le monstre cherchera à aller à sa gauche en premier lieu, puis *en cas d'obstacle* à continuer dans sa direction actuelle, puis à aller à sa droite, puis à faire demi-tour, puis *dans le cas où il y a des obstacles partout autour de lui* il ne bouge pas).

Cet algorithme nous a permis d'améliorer l'efficacité de notre mineur.

Voici un exemple de déplacement d'un monstre lorsqu'il n'y a pas d'obstacles.

t	*t+1*	*t+2*	*t+3*	*t+4*
				

Voici un exemple de carte prédictive. A partir de la carte au temps *t-1* et celle au temps, nous pouvons déterminer la prochaine direction du monstre, et par la même occasion, sa prochaine case de destination. Les carrés rouges représentent des obstacles virtuels sur la carte modifiée. Le mineur considérera, lors de la recherche de chemin, ces cases comme des obstacles. Il n'y a donc aucune chance qu'il meurt à cause des monstres.

t-1	*t*	*t+1* (carte prédite)
		

4. Gestion des chutes de rochers

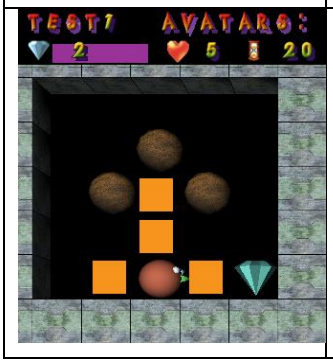
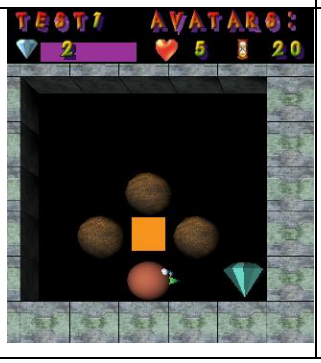
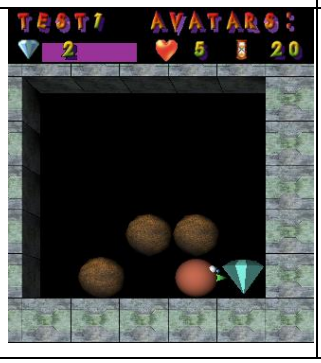
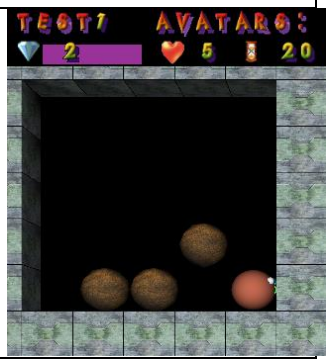
Le deuxième type d'obstacles rencontrés par le mineur est la chute de pierres. En effet, lorsque le mineur creuse, il peut laisser un vide sous certaines pierres et faire chuter celles-ci.

Afin d'éviter la mort du mineur, nous avons implémenté une solution relativement proche de celle qui permet d'esquiver les monstres.

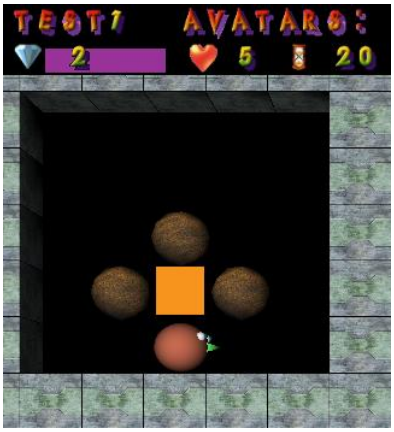
Tout d'abord nous comparons la position des pierres à un instant *t-1* et *t* pour déterminer si celles- ci sont en mouvement, si c'est le cas nous bloquons les cases susceptibles d'être atteintes par un rocher et de tuer notre mineur. Ces cases seront considérées comme des obstacles dans la carte de prédiction et l'algorithme A* les considèrera comme des cases non atteignables pour la recherche du plus court chemin.

Afin de déterminer le danger que représente un rocher, nous avons sauvegardé la liste répertoriant tous les rochers ainsi que leur état respectif (en cours de chute ou état stable).

Voici les cases qui seraient bloquées à l'instant *t+i* dans cette configuration. Les cases en orange représentent les obstacles virtuels sur la nouvelle carte modifiée.

t	*t+1*	*t+2*	*t+3*
			

Cas particulier



Sur cette figure, il n'y a pas d'obstacles virtuels à gauche et à droite du mineur (même si le rocher est en cours de chute).

Supposons que le mineur soit en position *(x,y)* alors les rochers qui sont en position *(x+1,y+1)* et *(x-1,y+1)* ne représente pas de danger pour lui. Comme sur l'illustration précédente, il pourra choisir de passer à gauche ou à droite.

5. Détection de danger imminent

En plus de la gestion des monstres et des chutes de rocher qui nous ont permis de créer une carte prédictive, nous avons anticipé le fait que des dangers puissent survenir lorsque le échoue à trouver un chemin vers sa destination (*auquel cas le choix par défaut serait de ne pas bouger et de recommencer toute la procédure à l'itération suivante pour trouver un diamant ou une sortie tout en gérant les dangers et obstacles*) avec le prédicat *haveToEscape*.

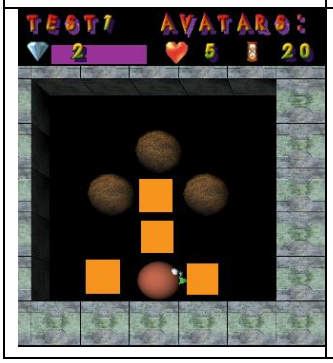
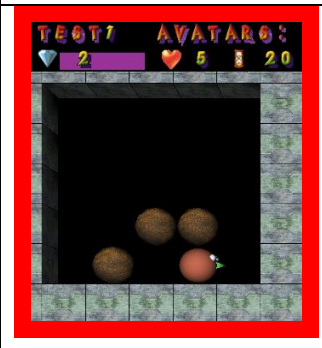
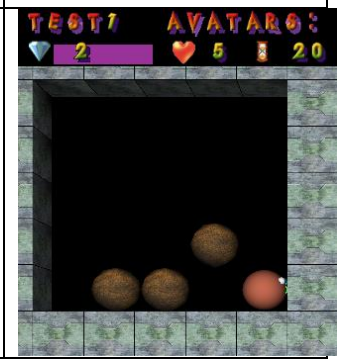
Dans ce cas-là nous testons deux types de danger potentiel :

- La chute des rochers sur les deux cases au-dessus du mineur (*fromFallingThings*)
- les monstres qui pourraient arriver sur l'une des cases adjacentes au mineur (*fromMonsters*)

Le prédicat *haveToEscape* renvoie vrai si l'un des deux prédicats *fromFallingThings ou fromMonsters* est vrai. Dans ce cas, le mineur cherchera la première case adjacente disponible (*toutes les cases disponibles de la carte modifiée par monster.pl et rock.pl sont sécurisées et aucun danger ne peut survenir à ces positions*).

Le mineur se déplacera vers cette position.

Danger imminent de rochers

t	*t+1*	*t+2*	*t+3*
			

Ici, contrairement au cas précédent, il n'y a pas de diamants sur la carte. Le mineur par défaut choisirait donc de rester sur place. Néanmoins, il doit d'abord passer par la *détection de danger imminent*.

- Au temps *t+1*, il détecte le rocher qui est deux cases au-dessus de lui. S'il ne se déplace pas, il mourrait à la prochaine itération. Il doit donc choisir de se déplacer et ainsi se placer en position de sécurité.
- C'est également le cas au temps *t+2*, où le rocher se situant juste au-dessus de lui représente un danger imminent. Il choisit donc encore de se déplacer.

A travers cet exemple, nous avons couvert les deux cas où un danger imminent de rocher pourrait survenir.

Danger imminent de monstres

Ici, le diamant est inaccessible. Il passe donc par la *détection de danger imminent* comme pour le cas précédent.

- Au temps t, aucun danger détecté. Il ne bouge pas.
- Au temps t+1, un monstre arriverait au temps t+2 à proximité. Il choisit de fuir et de se mettre en position de sécurité.

- Etant donné la configuration de la carte, il va se mettre ensuite en état de fuite indéfiniment.

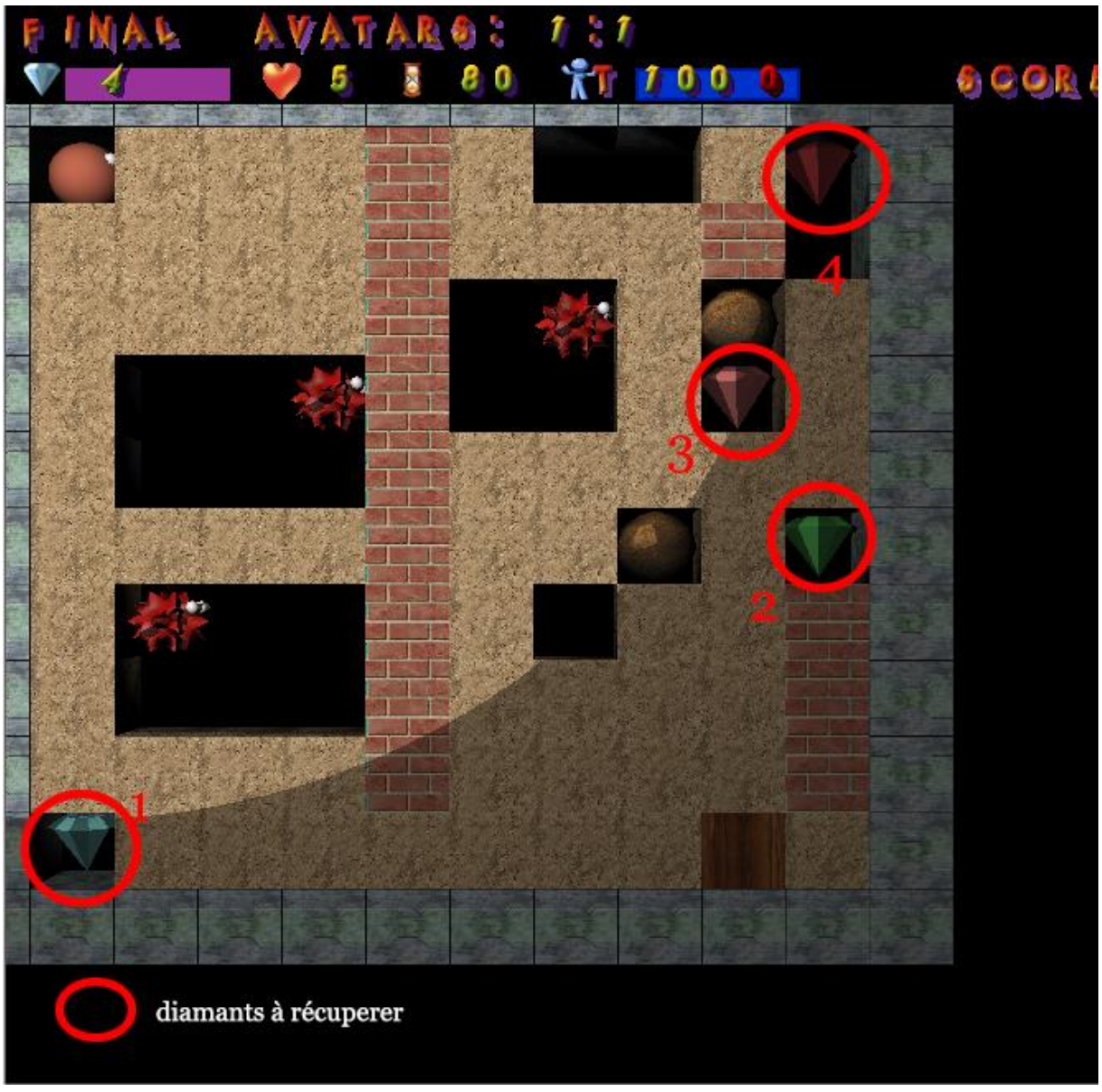
t	*t+1*	*t+2*
		

IV. SCENARIO

Dans cette partie, nous allons détailler un scénario complet. C'est-à-dire une séquence des actions que peut effectuer notre agent sur une carte.

Première situation

Notre agent repère les diamants sur la carte. Ensuite, après avoir calculé les longueurs des chemins pour arriver à chacun de ces diamants, il conserve en mémoire seulement le plus proche. Seulement après le déplacement commence. Cette phase de recherche est faite à chaque itération du jeu (expliqué en partie II).



Seconde situation

Notre agent commence son déplacement mais il doit faire attention aux monstres rouges. Pour cela, il peut prévoir le déplacement au prochain tour du monstre pour le prendre en compte dans sa recherche du plus court chemin, comme vous pouvez le voir dans l'illustration ci-dessous.

Troisième situation

Voici une autre situation où sur cette carte notre mineur doit aller chercher le prochain diamant au-dessus du mur à droite tout en évitant les monstres à proximité. Mais contrairement à la précédente situation nous pouvons remarquer que notre agent reste immobile durant trois cycles. En effet, en analysant le plus court chemin jusqu'au prochain diamant et le déplacement prévisionnel du monstre, il est plus économique de rester sur place que de ses déplacer. Cela peut paraitre surprenant puisque dans une même situation un humain aurait tendance à se déplacer même si ses déplacements rendraient son parcours plus long ou dans le meilleur des cas, ne réduiraient pas la distance. Suite à cela, notre mineur poursuit son chemin pour ramasser les derniers diamants sans encombre sur son chemin. Le troisième monstre étant bloqué, notre agent ne s'en soucie pas.

Quatrième situation

Notre agent a récupéré tous les diamants nécessaires. Maintenant, celui-ci localise la sortie et calcule le chemin le plus court (la flèche rouge) et le chemin le moins dangereux pour lui. Comme vous pouvez le remarquer, la pierre bloque le passage du monstre et le troisième monstre est toujours enfermé dans son espace, ce qui permet à notre mineur de ne pas les prendre en compte et d'arriver sain et sauf à la fin du niveau.

V. PROBLEMES RENCONTRES

Lors de la réalisation de notre produit, nous nous sommes confrontés à quelques problèmes.

- Le problème de périmètre de vision. Au départ, le périmètre de vision étant la carte entière, nous a rendu la tâche compliquée sur des cartes de grande taille. En effet, Prolog utilise une pile globale. Celle-ci se remplit rapidement et n'est vidé qu'au moment où l'intelligence artificielle est débranchée et rebranchée. La plus grosse difficulté liée à ce problème venait du fait que notre intelligence artificielle déclenchait des dépassements de pile et prenait des chemins incohérents lorsque nous lancions une carte deux fois de suite dû au fait que celle-ci gardait encore en mémoire les variables globales du premier lancement du jeu.

- Le problème lié aux rochers. En effet, les rochers ont une faille assez étrange lorsqu'ils sont en diagonale de notre mineur. Celui-ci peut passer en dessous sans mourir alors que dans les autres situations (voir partie III.4), cela le tue instantanément. Nous avons dû adapter notre programme à cette situation pour en tirer profit un maximum.

VI. AMELIORATIONS POSSIBLES

Dans notre projet, nous avons pu couvrir la majorité de nos objectifs. Malgré cela, l'intelligence artificielle de notre robot reste améliorable.

- Nous ne gérons qu'un seul type de monstre : le monstre rouge. Nous n'avons pas géré les déplacements des monstres bleus (ils ont un déplacement similaire aux monstres rouges mais font une « pause » tous les deux itérations.)

- Nous pourrions améliorer l'intelligence de notre robot en lui permettant d'être capable de tendre des pièges en se servant des rochers.

- Lorsqu'aucun chemin n'est trouvé vers sa destination (diamant ou sortie inaccessible), il pourrait pousser les rochers qui gêneraient éventuellement son itinéraire.

- Notre mineur ne sait pour l'instant prévoir une situation à l'instant *t+1*. Nous pourrions le doter d'une capacité de prédiction supérieure et ainsi l'éviter de se retrouver dans des positions dangereuses et « bloquantes ». Ceci est possible car on peut anticiper les déplacements des monstres ainsi que les chutes de rocher.

- Nous n'avons pas modifié le périmètre de vue de notre mineur. Nous l'avons laissé par défaut (toute la carte) et par conséquent, nous nous retrouvons avec un débordement de pile sur des cartes très grandes (40*24 par exemple). La première solution serait d'adapter notre périmètre de vue en fonction de la situation (de base, il serait très petit et augmenterait jusqu'à ce qu'un diamant soit dans son périmètre de vue). La deuxième solution, plus ardue, serait d'améliorer notre algorithme de recherche de diamant.

VII. CONCLUSION

Ce projet nous a permis de mettre en pratique nos compétences en Prolog acquises au cours du semestre. De plus, nous avons pu constater que la phase de réflexion du projet fut déterminante pour la suite. En effet contrairement à la programmation habituelle, l'intelligence artificielle demande une part de réflexion importante car tout ne peut pas être traité de façon algorithmique. De plus, le traitement demandant de nombreuses prévisions nous avons dû travailler sur la gestion de la mémoire (rochers et cartes au temps *t-1*) pour optimiser notre programme.

VIII. ANNEXES

1. decision.pl

```
/*

  Définition de l'IA du mineur

  Le prédicat move/12 est consulté à chaque itération du jeu.

*/

:- module( decision, [
      init/1,
      move/12
] ).

:- set_prolog_stack(global, limit(100 000 000 000 000 000 )).
:- use_module( a_star ).
:- use_module( monster ).
:- use_module( rock ).
:- use_module( diamant ).
:- use_module( exit ).

isEmpty([]).

isObstacle(3).
isObstacle(4).
isObstacle(5).
isObstacle(6).
isObstacle(23).
isObstacle(24).
isObstacle(25).
isObstacle(26).
isObstacle(27).
isObstacle(28).
isObstacle(50).

isMonster(23).
isMonster(24).
isMonster(25).
isMonster(26).
isMonster(27).
isMonster(28).

isCaseVide(0).

isRock(3).

init(_).



/*
```

```
  move( +L,+LP, +X,+Y,+Pos, +Size, +CanGotoExit, +Energy,+GEnergy, +VPx,+VPy, -
ActionId )
*/

/*
  First case : movement to diamond
*/

move( L,_, _,_,Pos, Size, CanGotoExit, _,_, _,_, Action ) :-

      CanGotoExit =:= 0,

      % Récupère les anciennes listes de vision et d'états des monstres
      nb_getval(prev_list, LI),
      nb_getval(rock_state, RS),

      % Associe les nouvelles positions des rochers et les anciennes positions
      associationRocks(L, Size, RS, NRS),

      nb_setval(rock_state, NRS),

      % Verifie les cases TopTopRight, TopTopLeft, TopTop, Top et renvoie la
nouvelle liste de vision
      wallsRocks(L, Pos, NRS, Size, L2),

      % Prédit la prochaine vision en tenant compte des prochaines destinations
des monstres
      monster(L2, LI, Size, L3),

      % Cherche le chemin vers le plus proche diamant
      findNearestDiamond(Pos, L3, L3, Size, Path, 0),

      ((
            % Chemin trouvé
            not(isEmpty(Path)), !,
            reverse(Path, Path1),

            % Prend le premier élément du chemin et choisit l'action
correspondant pour se rendre sur celui ci
            removehead(Path1, Path2),
            Path2 = [E|_],
            nextCell(Pos, E, Action1, Size),
            Action is Action1,
            nb_setval(prev_list, L)
      ) ; (
            % Chemin non trouvé

            % Vérifie la nécessité de s'échapper ou non
            haveToEscape(L3, Pos, Size, NRS), !,

            % Le personnage doit s'échapper

            % Choisit la première case adjacente vide (pour s'échapper)
            escape(L3, Pos, Size, Destination),
            nextCell(Pos, Destination, Action1, Size),
            Action is Action1,
            nb_setval(prev_list, L)
      )).
```

```
/*
  Second case : movement to exit
*/

move( L,_, _,_,Pos, Size, CanGotoExit, _,_, _,_, Action ) :-

	CanGotoExit =:= 1,

	% Récupère les anciennes listes de vision et d'états des monstres
	nb_getval(prev_list, LI),
	nb_getval(rock_state, RS),

	% Associe les nouvelles positions des rochers et les anciennes positions
	associationRocks(L, Size, RS, NRS),

	nb_setval(rock_state, NRS),

	% Verifie les cases TopTopRight, TopTopLeft, TopTop, Top et renvoie la
nouvelle liste de vision
	wallsRocks(L, Pos, NRS, Size, L2),

	% Prédit la prochaine vision en tenant compte des prochaines destinations
des monstres
	monster(L2, LI, Size, L3),

	% Cherche le chemin vers la sortie
	findExit(L3, PosExit),


	findPathExit(Pos, L3, Size, Path, PosExit),


	((

		% Chemin trouvé

		not(isEmpty(Path)), !,
		reverse(Path, Path1),

		% Prend le premier élément du chemin et choisit l'action
correspondant pour se rendre sur celui ci
		removehead(Path1, Path2),
		Path2 = [E|_],
		nextCell(Pos, E, Action1, Size),
		Action is Action1,
		nb_setval(prev_list, L)

	) ; (

		% Chemin non trouvé

		% Vérifie la nécessité de s'échapper ou non
		haveToEscape(L3, Pos, Size, NRS), !,

		% Le personnage doit s'échapper

		% Choisit la première case adjacente vide (pour s'échapper)
		escape(L3, Pos, Size, Destination),
		nextCell(Pos, Destination, Action1, Size),
```

```
            Action is Action1,
            nb_setval(prev_list, L)
      )).


/*
  Third case : no path found and no need to escape
*/

move( L,_, _,_,_, _, _, _,_, _,_, Action ) :-
      nb_setval(prev_list, L),
      Action is 0.

/*
  nextCell(+Current, +Next, -Action, +Size)
*/

/*
Détermine l'action de mouvement en fonction de la position actuelle et la
destination
*/

nextCell(C, C1, 1, _) :-
      C1 =:= C + 1,
      !.

nextCell(C, C1, 3, _) :-
      C1 =:= C - 1,
      !.

nextCell(C, C1, 2, Size) :-
      C1 =:= C - Size,
      !.

nextCell(C, C1, 4, Size) :-
      C1 =:= C + Size,
      !.


/*
  haveToEscape(+Vision, +Pos, +Size, -NewRockState)
*/

/*
      Renvoie true si le personnage doit s'échapper (à cause des monstres ou des
chutes de rochers)
*/
haveToEscape(Vision, Pos, Size, NewRockState) :-
      fromMonsters(Vision, Pos, Size) ;
      fromFallingThings(Vision, Pos, Size, NewRockState).

/*
  escape(+Vision, +Pos, +Size, -Destination)
*/

/*
      Cherche une case vide autour du monstre pour s'échapper
*/
escape(Vision, Pos, Size, Destination) :-
```

```
	Top is Pos - Size,
	Bot is Pos + Size,
	Left is Pos - 1,
	Right is Pos + 1,

	nth0(Top, Vision, EleTop),
	nth0(Bot, Vision, EleBot),
	nth0(Left, Vision, EleLeft),
	nth0(Right, Vision, EleRight),

	(	(isCaseVide(EleRight), Destination is Right) ;
		(isCaseVide(EleLeft), Destination is Left)  ;
		(isCaseVide(EleTop), Destination is Top) ;
		(isCaseVide(EleBot), Destination is Bot)
	).

/*
	Utilitaires
*/

removehead([_|Tail], Tail).
```

2. astar.pl

```
:- module( astar, [
	a_star/5
] ).

:- use_module(library(lists)).

isObstacle(3).
isObstacle(4).
isObstacle(5).
isObstacle(6).

isSortie(21).
isObstacle(23).
isObstacle(24).
isObstacle(25).
isObstacle(26).
isObstacle(27).
isObstacle(28).
isObstacle(50).


/*
	a_star(+InitialState, +FinalState, -Path, +Perceipt, +Size)
*/

/*
	Algorithme de l'astar servant à trouver un chemin à partir d'un point de
départ et de la destination
*/

a_star(InitialState, FinalState, Path, Perceipt, Size) :-

	/*
		Forme de l'openList
		[[Cout1, Chemin1], [Cout2, Chemin2], ...] où Cout(n) < Cout(n+1) et
Chemin(n) sont des listes de chemins
	*/

	/*
		Forme de la closedList
		[Position1, Position2, ...] où Position est l'index d'une case ayant
été déja visitée et vérifiée
	*/
	nb_setval(openList, [[[0,0], [InitialState]]]),
	nb_setval(closedList, []),
	expand_a_star(Path, Perceipt, FinalState, Size).

% Heuristique entre un point A et un point B
getHeuristicValue(A, H, B, Size) :-
	Xa is A mod Size,
	Ya is A // Size,
	Xb is B mod Size,
	Yb is B // Size,
	H is abs(Xb-Xa) + abs(Yb-Ya).
```

```
getTop(State,Perceipt,Top, Size) :-
	State > Size,
	Top1 is State - Size,
	nth0(Top1, Perceipt, P),
	nb_getval(closedList, ClosedList), not(member(Top1, ClosedList)),
	not(isObstacle(P)), !,
	Top is Top1.

getTop(_,_,Top, _) :-
	Top is -1.

getBottom(State,Perceipt,Bottom, Size) :-
	length(Perceipt, L),
	State < L - Size,
	Bottom1 is State + Size,
	nth0(Bottom1, Perceipt, P),
	nb_getval(closedList, ClosedList), not(member(Bottom1, ClosedList)),
	not(isObstacle(P)), !,
	Bottom is Bottom1.

getBottom(_,_,Bottom, _) :-
	Bottom is -1.

getRight(State,Perceipt,Right, Size) :-
	State mod Size <  Size - 1,
	Right1 is State + 1,
	nth0(Right1, Perceipt, P),
	nb_getval(closedList, ClosedList), not(member(Right1, ClosedList)),
	not(isObstacle(P)), !,
	Right is Right1.

getRight(_,_,Right, _) :-
	Right is -1.

getLeft(State,Perceipt,Left, Size) :-
	State mod Size > 0,
	Left1 is State - 1,
	nth0(Left1, Perceipt, P),
	nb_getval(closedList, ClosedList), not(member(Left1, ClosedList)),
	not(isObstacle(P)), !,
	Left is Left1.

getLeft(_,_,Left, _) :-
	Left is -1.

/*
	getAllAccessibleStates(+State, +Perceipt, -AccessibleStatesList, +Size)
*/

/*
	Retourne la liste des cases adjacentes à un point accessibles (ignore les
cases où il y a un obstacle)
*/

getAllAccessibleStates(State, Perceipt, AccessibleStatesList, Size) :-
	getTop(State,Perceipt,Top,Size),
	getBottom(State,Perceipt,Bottom, Size),
	getRight(State,Perceipt,Right, Size),
	getLeft(State,Perceipt,Left, Size),
```

```
	AccessibleStatesList = [Top,Bottom,Right,Left].


/*
	expand_a_star(-Path, +Perceipt, -FinalState, +Size)
*/

/*
	Prédicat récursif expand_a_star pour trouver le chemin avec l'algorithme
astar
*/

/*
	Cas 1 : Chemin trouvé
*/

expand_a_star(Path, _, FinalState, _) :-
	nb_getval(openList, OpenList),
	OpenList = [[_, Path]|_],
	Path = [FinalState|_],
	!.

/*
	Cas 2 : Chemin non trouvé
*/

expand_a_star([], _, _, _) :-
	nb_getval(openList, []),
	!.

/*
	Cas 3 : En cours de recherche
*/
expand_a_star(Path, Perceipt, FinalState, Size) :-


	nb_getval(openList, [[Cost, BestPath] | OthersOpen]),
	nb_getval(closedList, ClosedList),

	BestPath = [LastNode | _],

	nb_setval(openList, OthersOpen),

	% ajouter le noeud courant a la liste fermée
	append(ClosedList, [LastNode], ClosedList2),
	nb_setval(closedList, ClosedList2),

	% trouver tous les cases accessibles
	getAllAccessibleStates(LastNode, Perceipt, AccessibleStatesList, Size),

	% ajouter a la liste ouverte les cases accessibles
	addPath(BestPath, Cost, AccessibleStatesList, FinalState, Size),

	% recursivité
	expand_a_star(Path, Perceipt, FinalState, Size).

/*
	addPath(+BestPath, +Cost, +AccessibleStatesList, +FinalState, +Size)
*/
```

```
/*
	Ajoute les nouveaux chemins générés à la liste ouverte et trie la liste
	ouverte en fonction du cout pour obtenir le meilleur chemin en tête de liste
*/

addPath(_, _, [], _, _) :-
	nb_getval(openList, OpenList),
	sortOpenList(OpenList, NewOpenList),
	nb_setval(openList, NewOpenList).

addPath(BestPath, Cost, [-1|R], FinalState, Size) :-
	!, addPath(BestPath, Cost, R, FinalState, Size ).

addPath(BestPath, [G,F], [S | OtherNodes], FinalState, Size) :-

	% Recupere les noeuds et les poids
	% Calcule le nouveau cout
	getHeuristicValue(S, H, FinalState, Size),
	G1 is G + 1,
	F1 is G1 + H,

	NewCost = [G1,F1],

	% Ajoute a la liste ouverte le nouveau chemin
	append([S], BestPath, NewPath),
	addToOpenList(NewCost, NewPath),

	% Recursivité
	addPath(BestPath, [G,F], OtherNodes, FinalState, Size).

/*
	addToOpenList(+NewCost, +NewPath)
*/

/*
	Ajoute le chemin avec son cout correspondant à la liste ouverte
*/

addToOpenList(NewCost, NewPath) :-
	nb_getval(openList, OpenList),

	append([[NewCost, NewPath]], OpenList, NewOpenList),

	nb_setval(openList, NewOpenList).


/*
	sortOpenList(+List, +Sorted)
*/

/*
	Trie la liste avec la méthode du quick_sort en fonction du coût F
(croissant)
*/

sortOpenList(List,Sorted):-
	q_sort(List,[],Sorted).
```

```
q_sort([],Acc,Acc).

q_sort([[[H,I],J]|T],Acc,Sorted):-
       pivoting(I,T,L1,L2),
       q_sort(L1,Acc,Sorted1),
       q_sort(L2,[[[H,I],J]|Sorted1],Sorted).

pivoting(_,[],[],[]).
pivoting(I,[[[X,Y],Z]|T],[[[X,Y],Z]|L],G):- Y > I, pivoting(I,T,L,G).
pivoting(I,[[[X,Y],Z]|T],L,[[[X,Y],Z]|G]):- Y =< I, pivoting(I,T,L,G).
```

3. diamant.pl

```
:- module( diamant, [
	findNearestDiamond/6
] ).

:- use_module( a_star ).

isDiamond(2).

/*
	findNearestDiamond(+Pos, +Map, +Map, +Size, -Path, +Compteur)
*/

/*
	Trouver le chemin vers le diamant le plus proche
*/

findNearestDiamond(_, _, [], _, [], _).

/*
	Case 1 : Premier diamant trouvé
*/

findNearestDiamond(Pos, Map, [E|R], Size, Path, I) :-
	I1 is I+1,
	E =:= 2, !,

	a_star(Pos, I, TempPath, Map, Size),
	length(TempPath, Long),
	fnd(Pos, Map, R, Size, TempPath, Long, I1, NewPath),

	Path = NewPath.

/*
	Case 1 : Continuation de la recherche
*/

findNearestDiamond(Pos, Map, [E|R], Size, Path, I) :-
	I1 is I+1,
	E \= 2,
	findNearestDiamond(Pos, Map, R, Size, Path, I1).

/*
	fnd(+Pos, +Map, +Map, +Size, -Path, +Compteur)
*/

/*
	Appelé dès que le premier diamant a été trouvé et compare les longueurs
	entre le chemin le plus court actuel et le nouveau chemin trouvé
	Appel récursif avec le plus court chemin entre les deux
*/

fnd(Pos, Map, [E|R], Size, TempPath, Long, I, NewPath) :-
	I1 is I + 1,
	isDiamond(E), !,
```

```
	a_star(Pos, I, TempPath1, Map, Size),
	length(TempPath1, Long1),

	((
		Long =:= 0, !,
		fnd(Pos, Map, R, Size, TempPath1, Long1, I1, NewPath)
	);(
		Long1 =:= 0, !,
		fnd(Pos, Map, R, Size, TempPath, Long, I1, NewPath)
	);(
		Long1 < Long, !,
		fnd(Pos, Map, R, Size, TempPath1, Long1, I1, NewPath)
	);(
		Long1 >= Long, !,
		fnd(Pos, Map, R, Size, TempPath, Long, I1, NewPath)
	)).


fnd(Pos, Map, [_|R], Size, TempPath, L, I, NewPath) :-
	I1 is I + 1,
	fnd(Pos, Map, R, Size, TempPath, L, I1, NewPath).

fnd(_, _, [], _, L, _, _, L).
```

4. monster.pl

```
:- module( monster, [
	monster/4,
	fromMonsters/3
] ).


isCaseVide(0).
isHerbe(1).


isMonster(23).
isMonster(24).
isMonster(25).
isMonster(26).
isMonster(27).
isMonster(28).

/*
  monster(+Vision, +PreviousVision, +Size, -ModifiedVision)
*/

/*
	A partir de la map actuelle et la map antérieure,
	retourne la map de prédiction avec les nouvelles positions des monstres
*/

/*
	Cas 1 : Première itération du jeu
	La PreviousVision n'existe pas et il faut la générer

	Prochaine destination des monstres par ordre :
		Top,
		Right,
		Bottom,
		Left
*/

monster(Vision, [], Size, ModifiedVision) :-
	!,
	% Liste des positions des monstres actuels
	findAllMonsters(Vision, ListMonsters),
	newPositionOfMonstersFirst(Vision, ListMonsters, Size, ListMonsters1),

	predictedMapV2(Vision, ListMonsters, ListMonsters1, Size, ModifiedVision).

/*
	Cas 2 : Pas la première itération du jeu
	La PreviousVision existe déja et est exploité pour avoir la direction
actuelle du monstre
	et prédire sa destination suivante

	Algo de déplacement des monstres :
		Le monstre cherche d'abord à gauche de sa direction actuelle,
		sinon continue dans sa direction actuelle,
		sinon cherche à droite,
```

```
            sinon fait demi-tour,
            sinon reste sur place.
*/

monster(Vision, PreviousVision, Size, ModifiedVision) :-

      predict(Vision, PreviousVision, Size, ModifiedVision).


newPositionOfMonstersFirst(_,[], _, []).
newPositionOfMonstersFirst(Map, [MPosition|R], Size, [NewMPosition|R1]):-
      (
            (
                  getTop(MPosition, Map, P, Size),
                  nth0(P, Map, Element),
                  isCaseVide(Element), !,
                  NewMPosition is P
            );
            (
                  getRight(MPosition, Map, P, Size),
                  nth0(P, Map, Element),
                  isCaseVide(Element), !,
                  NewMPosition is P
            );
            (
                  getBottom(MPosition, Map, P, Size),
                  nth0(P, Map, Element),
                  isCaseVide(Element), !,
                  NewMPosition is P
            );
            (     getLeft(MPosition, Map, P, Size),
                  nth0(P, Map, Element),
                  isCaseVide(Element),
                  NewMPosition is P
            )
      ),

      newPositionOfMonstersFirst(Map, R, Size,R1).

/*
  predict(+Vision, +PreviousVision, +Size, -ModifiedVision)
*/

/*
      A partir de la map actuelle et la map antérieure,
      retourne la map de prédiction avec les nouvelles positions des monstres
*/

predict(Vision, PreviousVision, Size, PredictedVision):-
      % Liste des positions des monstres actuels
      findAllMonsters(Vision, ListMonsters),
      % Liste des positions des monstres anterieurs
      findAllMonsters(PreviousVision, PreviousMonsters),

      % Determine les futurs positions des monstres
      newPositionOfMonsters(Vision, ListMonsters, PreviousMonsters, Size,
ListMonsters1),

      % Ajoute des murs invisibles autour des monstres
```

```
	predictedMapV2(Vision, ListMonsters, ListMonsters1, Size, PredictedVision).

predictedMapV2(Vision, ListMonsters, ListMonsters1, Size, PredictedVision) :-
	deleteMonsters(Vision, ListMonsters, NewVision),
	addMonsters(NewVision, ListMonsters1, ModifiedVision2),
	% Add walls
	predictedMap(ModifiedVision2, ListMonsters1, Size, PredictedVision).

deleteMonsters(L, [], L).

deleteMonsters(Vision, [E|R], NewVision) :-
	replaceIndex(Vision, E, 0, NewVisionTemp),
	deleteMonsters(NewVisionTemp, R, NewVision).

addMonsters(L, [], L).

addMonsters(NewVision, [E|R], NewVision2) :-
	replaceIndex(NewVision, E, 23, NewVisionTemp),
	addMonsters(NewVisionTemp, R, NewVision2).

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

/*
  newPositionOfMonsters(+Vision, +ListMonsters, +PreviousMonsters, +Size, -
ListMonsters1)
*/

/*
	A partir de la liste des positions des monstres actuelle et la liste des
positions des monstres antérieure,
	retourne la liste des positions de prédiction des monstres future
*/

newPositionOfMonsters(Vision, ListMonsters, PreviousMonsters, Size,
ListMonsters1) :-

	% Recuperer les alentours pour chaque monstre de la map anterieure
	findTargets(PreviousMonsters, Vision, Size, Alentours),

	% Alentours = [[E, [E,A,B,C,D]], [E2, [E2, A2,B2,C2,D2]],...]

	% Verifier si pour chaque element de Alentours, il est contenu dans
ListMonsters
	% Si c'est le cas alors sortie d'une liste de sous liste de la forme
[[PreviousPosition, ActuelPosition],...]

	associationPositionsMonstres(Alentours, ListMonsters, PrevAndCurrent),

	% checkDouble et checkDouble2 sont là pour corriger les incohérences
	% (dans le cas où dans les alentours d'un monstre antérieur, il trouve deux
monstres se situant sur ceux-ci,
	% un test sera fait pour déterminer qui est le vrai monstre associé)
	checkDouble(PrevAndCurrent, PrevAndCurrent, SingleList, DoubleList),
	checkDouble2(DoubleList, SingleList, PrevAndCurrent2),

	% Renvoie la liste des positions des monstres futurs
	predictionNewPos(PrevAndCurrent2, Vision, Size, ListMonsters1).
```

```
/*
  findTargets(+PreviousMonsters, +Vision, +Size, -Alentours),
*/

/*
	Renvoie la liste des monstres avec leurs cases adjacentes
	(on vérifiera si dans la liste antérieure, un monstre a été dans une de ces
cases auquel cas on pourra
	connaitre la position anterieure d'un monstre et ainsi predire sa direction)

	Alentours est de la forme
	[[PositionMonstre1, [PosActuelle1,Top1,Right1,Bottom1,Left1]],
[PositionMonstre2, [PosActuelle2,Top2,Right2,Bottom2,Left2]],...]
*/

findTargets([], _, _, []).

findTargets([M|R], Map, Size, [[M, Neighbours]|R1]):-
	findNeighbours(M, Map, Size, Neighbours),
	findTargets(R, Map, Size, R1).


findNeighbours(M, Map, Size, [M, Top, Right, Bottom, Left]) :-

	getTop(M, Map, Top, Size),
	getRight(M, Map, Right, Size),
	getBottom(M, Map, Bottom, Size),
	getLeft(M, Map, Left, Size).

getTop(State,_,Top, Size) :-
	State >= Size, !,
	Top is State - Size.

getTop(_,_,Top, _) :-
	Top is -1.

getBottom(State,Perceipt,Bottom, Size) :-
	length(Perceipt, L),
	State =< L - Size, !,
	Bottom is State + Size.

getBottom(_,_,Bottom, _) :-
	Bottom is -1.

getRight(State,_,Right, Size) :-
	State mod Size <  Size - 1, !,
	Right is State + 1.

getRight(_,_,Right, _) :-
	Right is -1.

getLeft(State,_,Left, Size) :-
	State mod Size > 0, !,
	Left is State - 1.

getLeft(_,_,Left, _) :-
	Left is -1.
```

```
predictionNewPos([], _, _, []).

predictionNewPos([E|R], Vision, Size, [NewPos|R1]) :-

      newPosition(E, Vision, Size, NewPos),
      predictionNewPos(R, Vision, Size, R1).

newPosition([E,F], Vision, Size, NewPos) :-
      E =:= F, !, ((F1 is F+Size, nth0(F1, Vision, Element),
isCaseVide(Element), !, NewPos is F1) ;
                        (F1 is F+1, nth0(F1, Vision, Element),
isCaseVide(Element), !, NewPos is F1) ;
                        (F1 is F-1, nth0(F1, Vision, Element),
isCaseVide(Element), !, NewPos is F1) ;
                        (F1 is F-Size, nth0(F1, Vision, Element),
isCaseVide(Element), !, NewPos is F1)
                        ).


newPosition([E,F], Vision, Size, NewPos) :-
      E + 1 =:= F, !, ((F1 is F-Size, nth0(F1, Vision, Element),
isCaseVide(Element), !, NewPos is F1) ;
                              (F1 is F+1, nth0(F1, Vision, Element),
isCaseVide(Element), !, NewPos is F1) ;
                              (F1 is F+Size, nth0(F1, Vision, Element),
isCaseVide(Element), !, NewPos is F1) ;
                              (F1 is F-1, nth0(F1, Vision, Element),
isCaseVide(Element), !, NewPos is F1)
                              ).

newPosition([E,F], Vision, Size, NewPos) :-
      E - 1 =:= F, !, ((F1 is F+Size, nth0(F1, Vision, Element),
isCaseVide(Element), !, NewPos is F1) ;
                              (F1 is F-1, nth0(F1, Vision, Element),
isCaseVide(Element), !, NewPos is F1) ;
                              (F1 is F-Size, nth0(F1, Vision, Element),
isCaseVide(Element), !, NewPos is F1) ;
                              (F1 is F+1, nth0(F1, Vision, Element),
isCaseVide(Element), !, NewPos is F1)
                              ).

newPosition([E,F], Vision, Size, NewPos) :-
      E + Size =:= F, !,  ((F1 is F+1, nth0(F1, Vision, Element),
isCaseVide(Element), !, NewPos is F1) ;
                              (F1 is F+Size, nth0(F1, Vision, Element),
isCaseVide(Element), !, NewPos is F1) ;
                              (F1 is F-1, nth0(F1, Vision, Element),
isCaseVide(Element), !, NewPos is F1) ;
                              (F1 is F-Size, nth0(F1, Vision, Element),
isCaseVide(Element), !, NewPos is F1)
                              ).

newPosition([E,F], Vision, Size, NewPos) :-
      E - Size =:= F, !, ((F1 is F-1, nth0(F1, Vision, Element),
isCaseVide(Element), !, NewPos is F1) ;
                              (F1 is F-Size, nth0(F1, Vision, Element),
isCaseVide(Element), !, NewPos is F1) ;
                              (F1 is F+1, nth0(F1, Vision, Element),
isCaseVide(Element), !, NewPos is F1) ;
```

```
                                (F1 is F+Size, nth0(F1, Vision, Element),
isCaseVide(Element), !, NewPos is F1)
                                ).


/*
  associationPositionsMonstres(+Alentours, +ListMonsters, -PrevAndCurrent),
*/

/*
      Retourne la liste des positions précédentes et courants des monstres en
étudiant
      la liste des alentours d'un monstre antérieur et de la liste des positions
des monstres actuels
*/

associationPositionsMonstres([], _, []).

associationPositionsMonstres([[E,[A|R]]|Autre], ListMonsters, [[E,A]|R1]) :-
      member(A, ListMonsters), !,
      associationPositionsMonstres([[E,R]|Autre], ListMonsters, R1).

associationPositionsMonstres([[E,[A|R]]|Autre], ListMonsters, R1) :-
      not(member(A, ListMonsters)), !,
      associationPositionsMonstres([[E,R]|Autre], ListMonsters, R1).

associationPositionsMonstres([[_,[]]|Autre], ListMonsters, R1):-
      associationPositionsMonstres(Autre, ListMonsters, R1).

checkDouble2([],L,L).

checkDouble2([[E,N]|R], SingleList, PrevAndCurrent2) :-
      not_already_a_destination(N, SingleList), !,
      checkDouble2(R, [[E,N]|SingleList], PrevAndCurrent2).

checkDouble2([[_,_]|R], SingleList, PrevAndCurrent2) :-
      checkDouble2(R, SingleList, PrevAndCurrent2).

not_already_a_destination(_, []).

not_already_a_destination(N, [[_,E]|R1]) :-
      N \= E,
      not_already_a_destination(N, R1).

checkDouble([], _, [], []).

checkDouble([[E,N]|R], PrevAndCurrent, [[E,N]|R1], DoubleList) :-
      single(E, PrevAndCurrent), !,
      checkDouble(R, PrevAndCurrent, R1, DoubleList).

checkDouble([[E,N]|R], PrevAndCurrent, PrevAndCurrent2, [[E,N]|R1]) :-
      checkDouble(R, PrevAndCurrent, PrevAndCurrent2, R1).

single(_, []).

single(E, [[N,_]|R1]) :-
      E \= N,
      single(E, R1).
```

```
/*
  predictedMap(+ModifiedVision, +ListMonsters, +Size, +PredictedVision).
*/

/*
	ModifiedVision est la liste de vision future avec les nouvelles positions de
monstres
	Ajoute des murs invisibles (code 50) autour de ces monstres et retourne la
liste
	Le joueur ne pourra pas accéder à ces cases là à la prochaine itération
*/

predictedMap(Vision, [E|R], Size, PredictedVision):-
	Top is E - Size,
	Bot is E + Size,
	Left is E - 1,
	Right is E + 1,


	MonsterWall = [Top, Bot, Left, Right],
	keepOnlyValid(Vision, MonsterWall, SecondVision),

	predictedMap(SecondVision, R, Size, PredictedVision).

predictedMap(L, [], _, L).

keepOnlyValid(Vision, L, SecondVision) :-
	keepInMap(Vision, L, L1),
	areObstacles(Vision, L1, L2),
	replaceAll(Vision, L2, SecondVision).

replaceAll(Vision, [E|R], SecondVision) :-
	E \= -1, !,
	replace(Vision, E, 50, SecVision),

	replaceAll(SecVision, R, SecondVision).

replaceAll(Vision, [_|R], SecondVision) :-
	replaceAll(Vision, R, SecondVision).

replaceAll(L, [], L).

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

areObstacles(Vision, [E|R], [E|R1]) :-
	nth0(E, Vision, Ele),
	(isCaseVide(Ele);isHerbe(Ele)), !,
	areObstacles(Vision, R, R1).

areObstacles(Vision, [_|R], [-1|R1]) :-
	areObstacles(Vision, R, R1).

areObstacles(_, [], []).

keepInMap(Vision, [E|R], [E|R1]) :-
	length(Vision, L),
	E >= 0 , E < L,
```

```
	keepInMap(Vision, R, R1).

keepInMap(Vision, [E|R], [-1|R1]) :-
	length(Vision, L),
	(E < 0 ; E >= L),
	keepInMap(Vision, R, R1).

 keepInMap(_,[],[]).

findAllMonsters(Vision, ListMonsters) :-
	fd(Vision, ListMonsters, 0).

fd([E|R],[I|S], I) :-
	isMonster(E),
	I1 is I+1,
	fd(R, S, I1).

fd([E|R], S, I) :-
	not(isMonster(E)),
	I1 is I+1,
	fd(R, S, I1).

fd([],[],_).

/*
	fromMonsters(+Vision, +Pos, +Size)
*/

/*
	Renvoie true si le personnage est en danger de mort s'il ne bouge pas à
cause des monstres (un monstre qui serait à côté du personnage)
	Renvoie false sinon (aucun danger à rester sur place)

	Pour cela, vérifie les cases autour du personnage (Top, Right, Left, Bottom)
*/

fromMonsters(Vision, Pos, Size) :-
	Top is Pos - Size,
	Bot is Pos + Size,
	Left is Pos - 1,
	Right is Pos + 1,

	nth0(Top, Vision, EleTop),
	nth0(Bot, Vision, EleBot),
	nth0(Left, Vision, EleLeft),
	nth0(Right, Vision, EleRight),

	(	isMonster(EleTop) ;
		isMonster(EleBot) ;
		isMonster(EleLeft) ;
		isMonster(EleRight)
	).
```

5. rock.pl

```
:- module( rock, [
	associationRocks/4,
	wallsRocks/5,
	fromFallingThings/4
] ).


isMonster(23).
isMonster(24).
isMonster(25).
isMonster(26).
isMonster(27).
isMonster(28).

isCaseVide(0).

isRock(3).

isObstacle(1).
isObstacle(2).
isObstacle(3).
isObstacle(4).
isObstacle(5).
isObstacle(6).
%isObstacle(20).
isObstacle(23).
isObstacle(24).
isObstacle(25).
isObstacle(26).
isObstacle(27).
isObstacle(28).
isObstacle(50).

/*
	wallsRocks(+Vision, +Pos, +NewRockState, +Size, -NewVision)
*/

/*
	En fonction des états actuels des rochers, ajoute des murs invisibles sur la
map afin de
	permettre notre personnage d'éviter les zones de danger (chutes de rochers)
*/

wallsRocks(Vision, Pos, NewRockState, Size, NewVision) :-
	TTT is Pos - (3 * Size),
	TTR is Pos - (2 * Size) + 1,
	TTL is Pos - (2 * Size) - 1,
	TT is Pos - (2 * Size),

	getPosWallsTTD([TTT, TTR, TTL], Vision, NewRockState, Size, PosWalls),
	getPosWallsTT(TT, Vision, NewRockState, Size, PosWalls1),
	append(PosWalls, PosWalls1, PosWalls2),
	replaceAllByWalls(Vision, PosWalls2, NewVision).
```

```
/*
	replaceAllByWalls(+Vision, +PosWalls, -NewVision)
*/

/*
	Ajout des murs invisibles sur la carte sur les positions à risque (code 50)
*/

replaceAllByWalls(Vision, [E|R], SecondVision) :-
	E \= -1, !,
	replace(Vision, E, 50, SecVision),
	replaceAllByWalls(SecVision, R, SecondVision).

replaceAllByWalls(Vision, [_|R], SecondVision) :-
	replaceAllByWalls(Vision, R, SecondVision).

replaceAllByWalls(L, [], L).



/*
	replace(+List, +Position, +Element, -NewList)
*/

/*
	Remplace l'élément à la Position position de la liste par Element
*/

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

/*
	getPosWallsTT(D)(+Position, +Vision, +NewRockState, +Size, -
PositionInvisibleWall)
*/

/*
	Vérifie si l'élément à la position Position est un rocher et
	retourne la position des murs invisibles à placer (une position en dessous
pour la position TT et
	deux positions en dessous pour les positions TopTopRight et TopTopLeft)
*/

getPosWallsTT(E, Vision, NewRockState, Size, [E1]) :-
	E > 0,
	nth0(E, Vision, Element),
	isRock(Element), !,
	% isFalling ?
	getState(E, NewRockState, State),
	((
		State =:= 1, !,
		E1 is E + Size
	);(
		E1 is -1
	)).

getPosWallsTT(_, _, _, _, []).
```

```
getPosWallsTTD([], _, _,_,[]).

getPosWallsTTD([E|R], Vision, NewRockState, Size, [E1|R1]) :-
	E > 0,
	nth0(E, Vision, Element),
	isRock(Element), !,
	% isFalling ?
	getState(E, NewRockState, State),
	((
		State =:= 1, !,
		E1 is E + 2 * Size,
		getPosWallsTTD(R, Vision, NewRockState, Size, R1)
	);(
		E1 is -1,
		getPosWallsTTD(R, Vision, NewRockState, Size, R1)
	)).

getPosWallsTTD([_|R], Vision, NewRockState, Size, [E1|R1]) :-
	E1 is -1,
	getPosWallsTTD(R, Vision, NewRockState, Size, R1).

getState(E, NewRockState, State) :-
	memberSpecialPosState(E, NewRockState, PosState),
	PosState = [_, State].

/*
	fromFallingThings(+Vision, +Pos, +Size, +NewRockState)
*/

/*
	Renvoie true si le personnage est en danger de mort s'il ne bouge pas à
cause de chutes de rochers
	Renvoie false sinon (aucun danger à rester sur place)

	Pour cela, vérifie les deux cases au dessus du personnage (Top et TopTop)
*/

fromFallingThings(Vision, Pos, Size, NewRockState) :-
	T is Pos - Size,
	TT is Pos - Size * 2,
	(
		(haveToEscapeFromTop(Vision, T, NewRockState))
		;
		(haveToEscapeFromTopTop(Vision, TT, NewRockState))
	).

haveToEscapeFromTop(Vision, T, NewRockState) :-

	T > 0, nth0(T, Vision, EleTop),
	isRock(EleTop),
	getState(T, NewRockState, State),
	State =:= 1.

haveToEscapeFromTopTop(Vision, TT, NewRockState) :-

	TT > 0, nth0(TT, Vision, EleTopTop),
	isRock(EleTopTop),
	getState(TT, NewRockState, State),
	State =:= 1.
```

```
/*
	associationRocks(+Vision, +Size, +RockState, -NewRockState)
*/

/*
	Actualise l'état des rochers de la vision actuelle (l'état est 1 si état
'chute' ; l'état est 0 si état immobile)
*/

associationRocks(Vision, _, [], NewRockState) :-
	findAllRocks(Vision, ListRocks),
	buildState(ListRocks, NewRockState).

associationRocks(Vision, Size, RockState, NewRockState) :-

	findAllRocks(Vision, ListRocks),

	% Pour chaque rocher trouve son correspondant dans la liste de rochers
anterieures
	associate(Vision, Size, ListRocks, RockState, NewRockState).

findAllRocks(Vision, ListRocks) :-
	fd(Vision, ListRocks, 0).

fd([E|R],[I|S], I) :-
	isRock(E),
	I1 is I+1,
	fd(R, S, I1).

fd([E|R], S, I) :-
	not(isRock(E)),
	I1 is I+1,
	fd(R, S, I1).

fd([],[],_).

buildState([],[]).

buildState([E|R], [[E,0]|R1]) :-
	buildState(R, R1).

/*
	associate(+Vision, +Size, +ListRocks, +RockState, -NewRockState)
*/

/*
	A partir de la liste des rochers de la map actuelle et de la liste d'états
des rochers de la map antérieure,
	retourne la nouvelle liste d'états des rochers issue de la map actuelle
*/

associate(_, _, [], _, []).

% Si trouve la meme position alors meme rocher et meme etat
associate(Vision, Size, [E|R], RockState, [PosState|R1]) :-
	memberSpecialPosState(E, RockState, PosState), !,
	associate(Vision, Size, R, RockState, R1).
```

```
% Si trouve rocher en train de tomber et qu'il y a encore une case vide
% en dessous alors etat 'chute'
associate(Vision, Size, [E|R], RockState, [[E,1]|R1]) :-
	Bot is E + Size,
	nth0(Bot, Vision, EleBot),
	isCaseVide(EleBot), !,
	associate(Vision, Size, R, RockState, R1).


% Si trouve rocher en train de tomber et quil y a obstacle
% en dessous alors etat 'immobile'
associate(Vision, Size, [E|R], RockState, [[E,0]|R1]) :-
	Bot is E + Size,
	nth0(Bot, Vision, EleBot),
	isObstacle(EleBot), !,
	associate(Vision, Size, R, RockState, R1).

/*
	memberSpecialPosState(+Position, +RockState, -PosState)
*/

/*
	Vérifie si l'élément en position Position est dans la liste d'états des
rochers et
	retourne l'état de ce rocher  (utilisé dans le cas où le rocher n'a pas
changé de place depuis la dernière itération)
*/

memberSpecialPosState(E, [[E,State]|_], [E,State]) :- !.

memberSpecialPosState(E, [[F,_]|R], PosState) :- E \= F, memberSpecialPosState(E,
R, PosState).
```

6. exit.pl

```
:- module( exit, [
	findExit/2,
	findPathExit/5

] ).

:- use_module(library(lists)).
:- use_module( a_star ).

isExit(21).

/*
	findPathExit(+Pos,+ Map, +Size, -Path, +PosExit)
*/

/*
	Trouver le chemin vers la sortie
*/

findPathExit(Pos, Map, Size, Path, PosExit) :-
	a_star(Pos, PosExit, Path, Map, Size).

/*
	findExit(+Map, -PosExit)
*/

/*
	Trouver la position de la sortie
*/

findExit(Map, PosExit) :-
	isExit(X),
	nth0(PosExit, Map, X).
```