Projet
IA 41

Rasende Roboter

Groupe 07 :

Étienne FRANCOIS
Loïc DEMMERLÉ
Yannick GUILLEMOT

Responsable UV :

Fabrice LAURI

Semestre automne 2010/11

Sommaire

I. Présentation du sujet « Rasende Roboter »

But du jeu :

Etre le joueur ayant accompli le plus de missions. Une mission consiste à trouver le nombre minimum de coups permettant à un robot d'une certaine couleur d'atteindre la cible placée au milieu du plateau. Le joueur ayant accompli le plus de missions a gagné la partie.

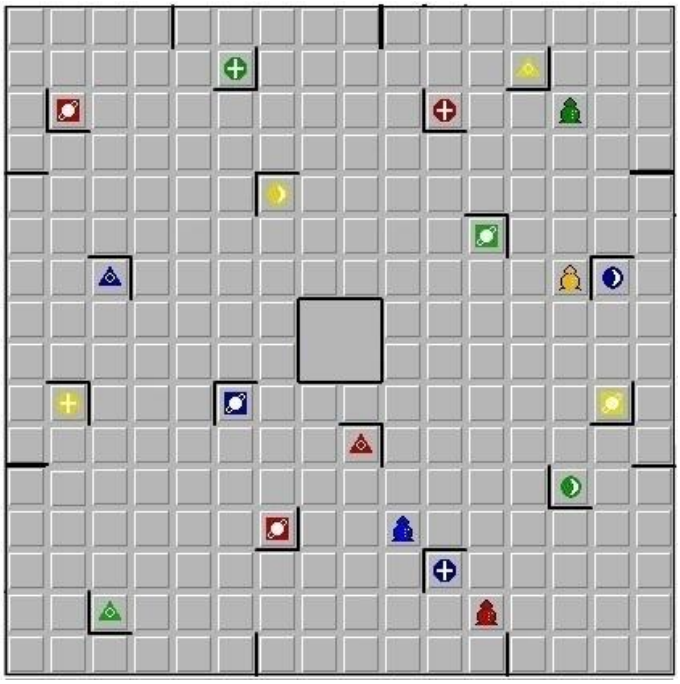
Objectif du projet :

L'utilisateur doit pouvoir effectuer des parties de Rasende Roboter contre l'ordinateur. Une solution doit être proposée par une intelligence artificielle à créer.

L'application doit avant tout mettre en avant une intelligence artificielle convaincante. Sa jouabilité n'est que secondaire car elle ne relève pas de l'objectif d'un projet sur l'élaboration d'une intelligence artificielle. Néanmoins, on doit pouvoir manipuler une interface graphique représentant le plateau de jeu et donnant la possibilité d'observer le résultat de l'ordinateur.

Une intelligence artificielle à plusieurs niveaux de difficulté est demandée dans le sujet mais cette contrainte n'a pas été jugée prioritaire dans l'élaboration de ce projet, car la seule position de départ des robots sur le plateau fait déjà varier le niveau de difficulté de la partie.

Plateau de jeu :



II. Formalisation du problème

L'intelligence artificielle doit fournir la solution la plus simple et la plus rapide. Pour cela, il est nécessaire de fournir dans le langage convenu les différentes caractéristiques du jeu. Le langage utilisé est PROLOG.
Les caractéristiques et paramètres à prendre en compte dans Rasende Roboter sont nombreux et il est nécessaire de les énumérer et de les formaliser correctement.

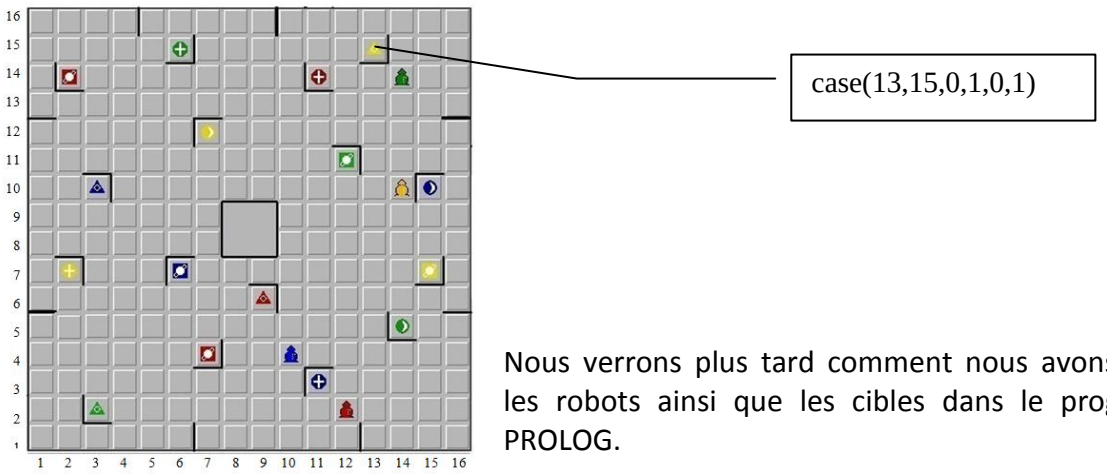
a. Caractéristiques du jeu

- 16 lignes, 16 colonnes
- 4 plateaux de 8x8 cases
- 252 cases
- Des murs entourent le plateau de jeu et sont disposés aléatoirement en plein milieu de celui-ci
- 4 robots de couleurs rouge, vert, bleu, jaune
- 16 cibles (4 par plateau)
- Une cible est entourée de 2 murs
- 4 couleurs pour les cibles : jaune, vert, bleu, rouge
- 4 symboles : triangle, carré, rond, plus
- Chaque couleur et chaque symbole sont présents sur tous les plateaux une seule fois.

b. Les cases

La première nécessité est de modéliser les cases, fondements même du jeu. Ainsi, le plateau est composé de 256 cases sur 16 lignes et 16 colonnes. On va donc identifier une case par son numéro de ligne X et son numéro de colonne Y.
De plus, certaines cases sont encadrées par un mur ou deux. Par exemple, la case ci-dessus est encadré par un mur sur son côté inférieur et sur son côté gauche. On doit donc préciser dans notre modélisation si la case est encadrée à gauche, à droite, en haut ou en bas.
On a donc 256 règles modélisant chacune les 256 cases. La règle PROLOG représentant donc une case est :

case(X,Y, MurHaut, MurBas, MurGauche, MurDroit)



Nous verrons plus tard comment nous avons traités les robots ainsi que les cibles dans le programme PROLOG.

III. Analyse du problème

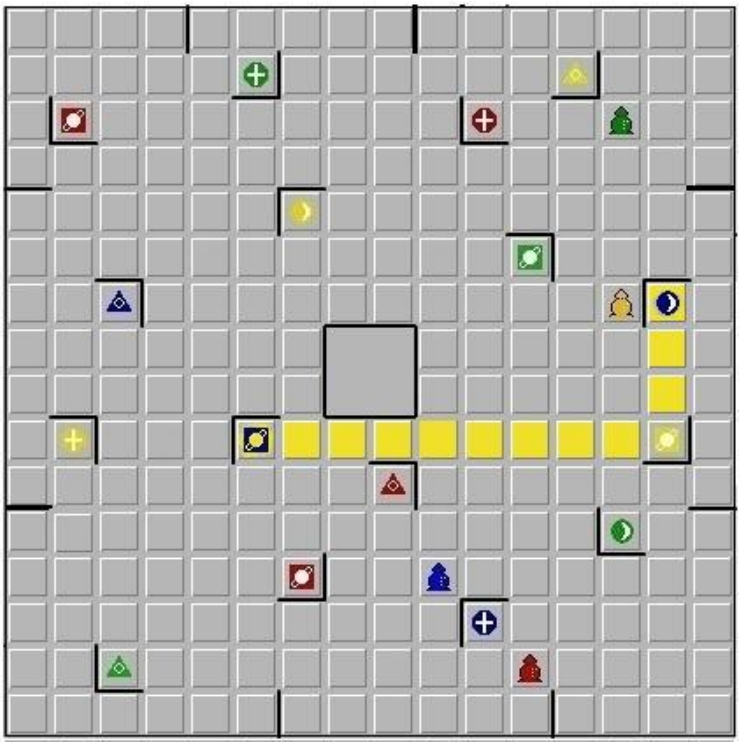
Au début d'une mission, le joueur voit apparaître au milieu du plateau la cible à atteindre ainsi que le robot avec lequel la mission doit être réalisée. Les robots sont disposés aléatoirement. On peut cependant manipuler les 3 autres robots présents sur le plateau et s'en servir pour accomplir la mission.
Les déplacements font la spécificité de Rasende Roboter. Chaque robot peut se déplacer à la verticale et à l'horizontale et il ne peut s'arrêter que lorsqu'il rencontre un obstacle. Un obstacle peut être un mur ou un autre robot.
La cible «objectif» doit être atteinte en un minimum de coups. Un coup correspond à un déplacement de robot quel qu'il soit.

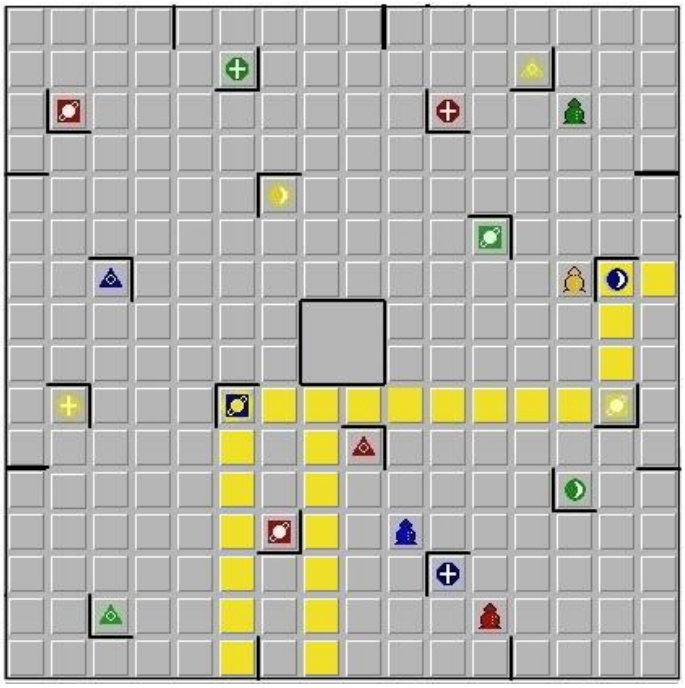
Pendant la période du projet, notre apprentissage de l'Intelligence Artificielle grandissant, nous nous sommes démenés à réaliser plusieurs analyses du jeu et des principes de résolution possibles.

a. Premières analyses

Tout au long du projet, la meilleure façon d'analyser le problème et sa possible résolution a été de jouer au jeu Rasende Roboter (http://www.ricochetrobots.com/RR.html).
La cible étant la case à atteindre, on part de celle-ci pour «remonter le parcours» menant à la cible.

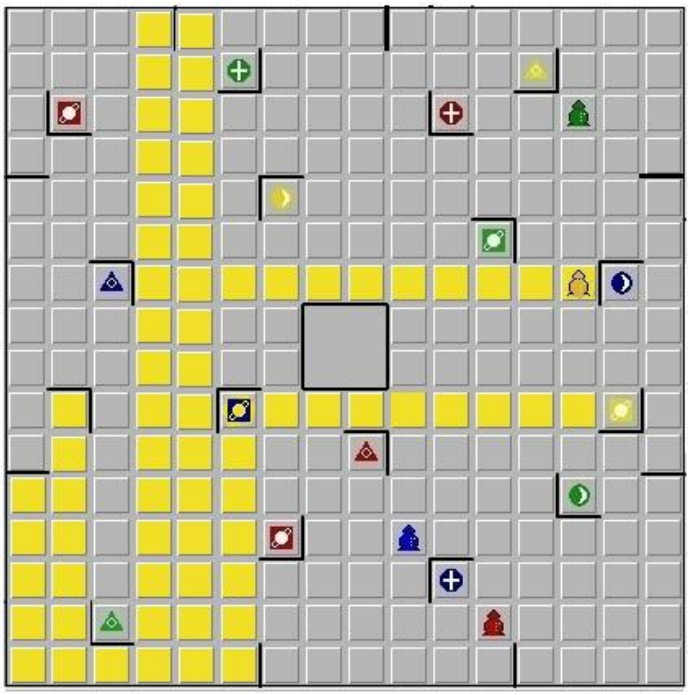
A la première étape, on a deux couloirs donnant accès direct à la cible en un coup. Ils sont surlignés en jaune. Si le robot s'y trouve, on peut considérer que la partie est gagnée.





On cherche ensuite tous les couloirs qui permettent d'accéder aux couloirs de l'étape précédente.
Et ainsi de suite... jusqu'à arriver à une situation d'arrêt : le robot est dans le couloir.

Cette proposition de résolution a finalement été rejeté car dégageant difficilement une méthode pouvant être modélisée en PROLOG. De plus, l'usage des autres robots ne peut pas être pris en compte.



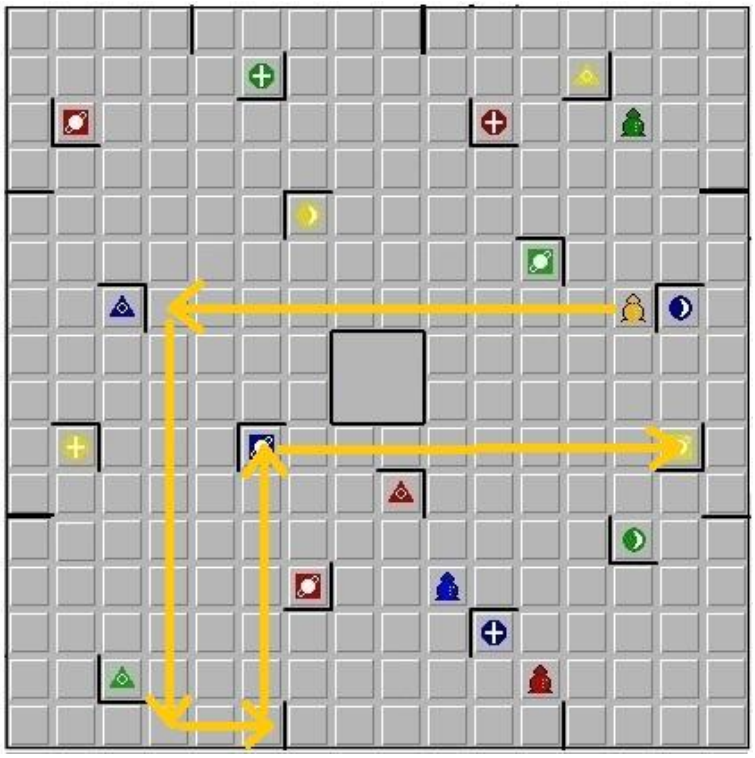
b. Analyse finale

Le rôle de l'intelligence artificielle est de trouver, pour un robot et une cible donnés, le plus court chemin qui mènerait le robot à la cible. L'algorithme de recherche A* convient à ce problème.
En effet, utilisé avec la distance euclidienne entre la position du robot et la cible comme heuristique, on avance dans la recherche en étudiant d'abord les chemins les moins coûteux et qui réduisent le plus la distance restant à parcourir. L'algorithme est donc plus efficace car il n'étudie que les positions les plus pertinentes. Nous allons voir ci-après comment cet algorithme a été adapté à notre problème et comment il a été programmé.

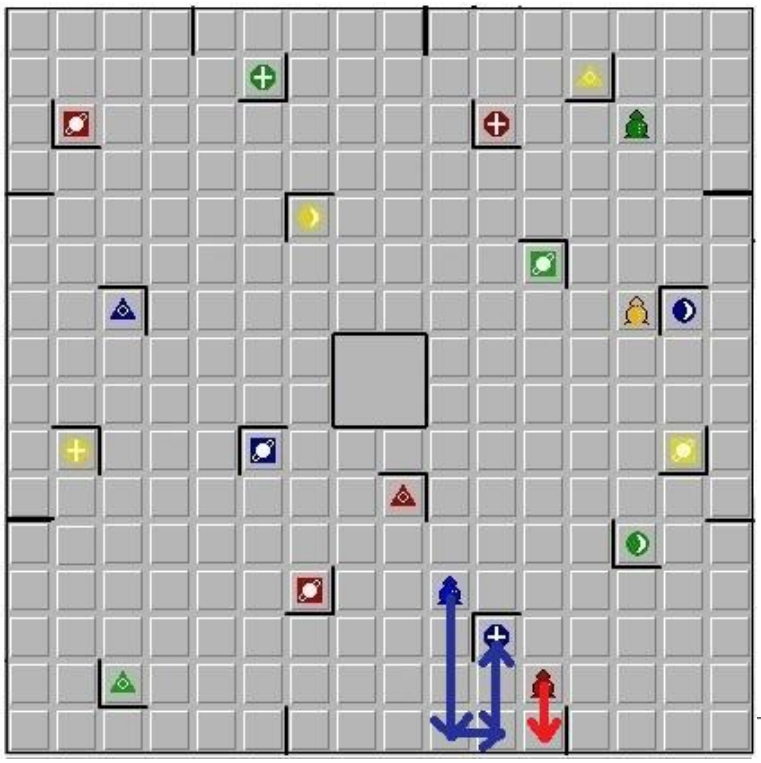
IV. Résolution du jeu

a. Cas pouvant être traité

La solution implantée permet de trouver le chemin le plus court entre le robot de départ et la cible désignée, dans le cas où il n'est pas nécessaire de s'aider d'un ou plusieurs des trois autres robots. En effet, dans certaines configurations il est impossible de rejoindre une cible si l'on ne s'aide pas d'un autre robot.



Exemple d'une solution en 5 coups qui peut être résolue sans déplacer un ou plusieurs autres robots.



Dans cet exemple, le robot Bleu serait incapable de rejoindre la cible si le robot rouge (par exemple) ne rejoignait pas la position (12,1). C'est le robot rouge qui sert d'obstacle lors du 2ème déplacement du robot bleu, et qui lui permet de s'aligner face à la cible.

Cette solution comporte 4 coups. 1 pour le robot rouge puis 3 pour le robot bleu. Elle ne peut pas être traitée par la solution proposée.

b. Fonctionnement de l'algorithme de résolution

Pour résoudre les cas du type énoncé précédemment, nous utilisons un algorithme A* implémenté en langage Prolog. Le fonctionnement précis ainsi que les principaux prédicats participant à la résolution du problème seront développés dans les sous-parties de ce chapitre.

i. Fonctionnement général

Le prédicat à appeler permettant de trouver la solution au problème est **jeu_1robot**. Il faut lui passer 11 arguments et il nous renvoie une liste représentant le chemin menant à la solution, ainsi que le nombre de coups joués. Si la cible n'est pas atteignable, le prédicat retourne **false**.

```
6  %-------------------------------------------------------------------------------------------------
7  %               ******* JEU RASENDE ROBOTER *******
8  % trouve le chemin le plus court entre un robot et une cible...
9  % si elle est atteignable sans déplacer d'autres robots
10 %-------------------------------------------------------------------------------------------------
11 jeu_1robot(XRBleu,YRBleu, XRVert,YRVert, XRRouge,YRRouge, XRJaune,YRJaune, RObj, XCible,YCible, Chemin, NbCoups) :-
12                         % enregistre la position des quatres robots en var. globales
13                         nb_linkval(rbtBleu,[XRBleu,YRBleu]),
14                         nb_linkval(rbtVert,[XRVert,YRVert]),
15                         nb_linkval(rbtRouge,[XRRouge,YRRouge]),
16                         nb_linkval(rbtJaune,[XRJaune,YRJaune]),
17                         %on récupère la position du robot qui doit atteindre la cible
18                         nb_getval(RObj,PosRobot),
19                         % Initialise l'openList et ajoute le noeud de départ
20                         nb_linkval(openList,[[PosRobot,PosRobot,0,0,0]]),
21                         % Initialise la closedList comme liste vide
22                         nb_linkval(closedList,[]),
23                         % Sauvegarde la position de départ et celle d'arrivée
24                         nb_linkval(start,PosRobot),
25                         nb_linkval(cible,[XCible,YCible]),
26                         a_star( RObj, [XCible,YCible], Chemin ),
27                         nb_coups(NbCoups).
28
```

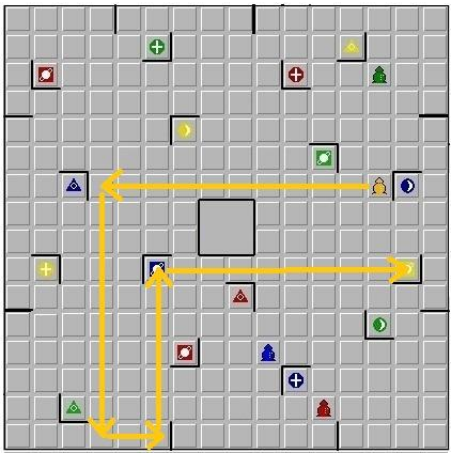
- **Arguments :**
  - XRBleu, YRBleu, …, XRJaune, YRjaune
    Les coordonnées de la position des quatre robots sur le plateau de jeu.
  - RObj
    C'est le nom du robot qui doit atteindre la cible
    (4 possibilités : rbtBleu, rbtRouge, rbtVert ou rbtJaune)
  - XCible et YCible
    Coordonnées de la cible à atteindre
- **Données retournées :**
  - Chemin
    Contient la liste des positions successives que le robot parcours depuis sa position initiale jusqu'à la cible.
  - NbCoups
    Renvoie le nombre de coups joués pour atteindre la cible
- **Variables globales :**
  - rbtBleu, rbtVert, rbtRouge, rbtJaune
    Chaque position de robot est mémorisée dans une variable globale qui pourra ensuite être réutilisée et modifiée par d'autres prédicats.
  - openList et closedList
    Listes contenant les nœuds à analyser, et ceux analysés par l'algorithme A.*
  - start et cible

Les positions du robot de départ et de la cible sont également mémorisées.

- **Exemple d'utilisation :**

Appel du prédicat jeu_1robot pour le cas présenté précédemment.

```
16 ?- jeu_1robot(10,4,14,14,12,2,14,10,rbtJaune,15,7,Chem,Nb).
Chem = '14,10|4,10|4,1|6,1|6,7|15,7',
Nb = 5 ;
```



Nous savons donc qu'il faut 5 coups pour atteindre la cible jaune en position (15,7) avec le robot jaune ayant comme point de départ la position (14,10).

Le chemin renvoyé correspond aux positions successives que le robot parcourt depuis sa position initiale jusqu'à la cible. Dans notre cas :

(14,10) →(4,10) → (4,1) → (6,1) → (6,7) → (15,7)

(*Le formalisme de la liste renvoyée a été choisi ainsi pour être facilement réutilisable par le code C++ de l'interface graphique.)*

ii. Les différents prédicats utilisés par A*

Voici la liste des prédicats utilisés par l'algorithme A* :
a_star(+nomRobot, +PositionFinaleSouhaitée, -CheminPourYArriver)
buildPath(+Robot, +PosFinale)
getBestNodeFromOpenList(-Node)
extractBestNodeFromOpenList(-Node)
insertNextPositionsInOpenList(+PosParent)

ajouterNoeud(+Noeud, +ListeDesNoeuds, -NouvelleListeAvecNoeud)
retirerNoeud(+Noeud, +ListeDesNoeuds, -NouvelleListeSansNoeud)
creerNoeud(+Pos, +PosParent, -[Pos, PosParent, h(Pos), g(Pos), f(Pos)])

getAllAccessiblePositions(+Position, -AccessiblePositionsList)

Fonctions de déplacement robot (utilisent le prédicat **case** définit dans le fichier case.pl) :
gauche(+XDepart, +YDepart, -XArrivee, -YArrivee)
droit(+XDepart, +YDepart, -XArrivee, -YArrivee)
haut(+XDepart, +YDepart, -XArrivee, -YArrivee)
bas(+XDepart, +YDepart, -XArrivee, -YArrivee)

iii. Fonctionnement de l'algorithme A*

L'algorithme A* manipule et analyse les nœuds contenus dans l'openList et les bascule dans la closedList une fois traités. Ci-dessous, le contenu des deux listes à la fin du traitement du cas précédent (robot jaune en position (14,10) et sa cible en position(15,7)).

Format d'un nœud :

```
[[1,13],[14,13],15.2315,2,17.2315]
```

[Position , Pos. Parent , heuristique , g , f = g + h]

L'heuristique h correspond à la distance euclidienne entre la position courante et la cible.
Le coût g correspond au nombre de coups joués pour atteindre la position courante.

```
***** OPENLIST ******
[[1,1],[4,1],15.2315,3,18.2315]
[[1,5],[10,5],14.1421,6,20.1421]
[[11,16],[11,14],9.84886,7,16.8489]
[[4,16],[4,10],14.2127,2,16.2127]
[[1,13],[14,13],15.2315,2,17.2315]

***** CLOSEDLIST ******
[[15,7],[6,7],0.0,5,5.0]
[[6,7],[6,1],9.0,4,13.0]
[[6,1],[4,1],10.8167,3,13.8167]
[[4,1],[4,10],12.53,2,14.53]
[[13,5],[10,5],2.82843,6,8.82843]
[[10,5],[10,16],5.38516,5,10.3852]
[[10,16],[16,16],10.2956,4,14.2956]
[[11,14],[13,14],8.06226,6,14.0623]
[[4,10],[14,10],11.4018,1,12.4018]
[[13,14],[13,1],7.28011,5,12.2801]
[[16,16],[16,13],9.05539,3,12.0554]
[[13,1],[16,1],6.32456,4,10.3246]
[[16,1],[16,5],6.08276,3,9.08276]
[[16,13],[14,13],6.08276,2,8.08276]
[[14,13],[14,10],6.08276,1,7.08276]
[[16,5],[14,5],2.23607,2,4.23607]
[[14,5],[14,10],2.23607,1,3.23607]
[[14,10],[14,10],0,0,0]
```

L'**OpenList** contient les nœuds non encore analysés. Si cette liste est vide c'est qu'aucune solution n'existe pour le problème posé.

La **ClosedList** contient tous les nœuds analysés. Le dernier élément de la liste est le dernier nœud à avoir été analysé. Dans notre cas, il s'agit de la position de la cible ce qui veut dire qu'un chemin a été trouvé. De plus h = 0 indique que nous sommes sur la cible.

La position du parent de la position courante nous permet de retracer le chemin le plus court entre la cible et la position de départ.
On remarquera que seulement 18 positions ont eu besoin d'être analysées. Le temps d'exécution de l'algorithme est quasiment instantané. Les positions déjà présentes dans l'openList ou closedList ne sont pas ajoutées à l'openList, à moins d'avoir une valeur de f inférieure à celle de la position déjà présente

iv. Prédicat buildPath

Ce prédicat modifie le contenu de l'openList et de la closedList. C'est lui qui exécute le principe de l'algorithme A* en se servant des prédicats **getBestNodeFromOpenList**, **extractBestNodeFromOpenList**, et **insertNextPositionsInOpenList**.

Ligne 54 : le prédicat est vrai si la position du meilleur nœud de la liste Open correspond à la position finale. Ce nœud est transféré dans la liste Closed (ligne 55) et la variable globale contenant la position du robot est modifiée (ligne 56).

Ligne 58 : on transfère le meilleur nœud de la liste Open vers la liste Closed. La position du robot en mouvement est actualisée (ligne 59). Le meilleur nœud est ensuite analysé (ligne 60), puis buildPath est rappelé récursivement (ligne 61). Analyser le meilleur noeud revient à ajouter les nouvelles positions accessibles depuis la position courante dans la liste Open.

```
50 %-------------------------------------------------------------------------------------------
51 % builPath analyse les noeuds contenus dans OpenList et les transfère dans closedList
52 % selon la méthode de l'algorithme A*
53 %-------------------------------------------------------------------------------------------
54 buildPath(Robot, PosFinale) :- getBestNodeFromOpenList([PosFinale,_,_,_,_]),
55                                 extractBestNodeFromOpenList([Pos,_,_,_,_]),
56                                 nb_linkval(Robot,Pos), !.
57
58 buildPath(Robot, PosFinale) :- extractBestNodeFromOpenList([Pos,_,_,_,_]),
59                                 nb_linkval(Robot,Pos),
60                                 insertNextPositionsInOpenList(Pos),
61                                 buildPath(Robot, PosFinale).
62 %-------------------------------------------------------------------------------------------
63 %-------------------------------------------------------------------------------------------
```

v. Prédicats extractBestNodeFromOpenList et getBestNode…

extractBestNodeFromOpenList(-Node)

Récupère le nœud de l'openList ayant la valeur f la plus faible et le transfère dans la closedList. Ce même nœud est également retourné par le prédicat.

```
68 %-------------------------------------------------------------------------------------------
69 %extractBestNodeFromOpenList( -Node ).
70 %l'enlève d'openList et le met dans closedList
71 %-------------------------------------------------------------------------------------------
72 extractBestNodeFromOpenList(Node) :-
73         getBestNodeFromOpenList(Node),
74         nb_getval(openList,OL), retirerNoeud(Node,OL,NewOL), nb_setval(openList,NewOL),
75         nb_getval(closedList,CL), ajouterNoeud(Node,CL,NewCL), nb_setval(closedList,NewCL).
76 %-------------------------------------------------------------------------------------------
77 %-------------------------------------------------------------------------------------------
78
79
80 %-------------------------------------------------------------------------------------------
81 %getBestNodeFromOpenList( -Node ).
82 %Renvoie le meilleur noeud de l'openList
83 %-------------------------------------------------------------------------------------------
84 getBestNodeFromOpenList(Node) :- nb_getval(openList,L), getBest(L,Node).
85 getBest([Node],Node) :- !.
86 getBest([[P1,PP1,H1,G1,F1]|R],[P1,PP1,H1,G1,F1]) :- getBest(R,[_,_,_,_,F2]), F1 =< F2, !.
87 getBest([[_,_,_,_,F1]|R],[P2,PP2,H2,G2,F2]) :- getBest(R,[P2,PP2,H2,G2,F2]), F1 > F2.
88 %-------------------------------------------------------------------------------------------
89 %-------------------------------------------------------------------------------------------
```

vi. Prédicat getAllAccessiblePositions

Il existe 2 variantes du prédicat.

- insertNextPositionsInOpenList(+PosParent)
  Récupère les positions atteignables depuis la position passée en paramètre et appelle la deuxième variante du prédicat avec cette liste de positions.
- insertNextPositionsInOpenList(+PosParent, +ListeDesPositionsEnfant)
  Insère les positions passées en paramètre dans la liste Open si elles n'y figurent pas encore, ni dans la liste Closed.

```
%-------------------------------------------------------------------------------------------------------------------------
%insertNextPositionsInOpenList( +PosParent).
%
%Insère dans openList les noeuds représentant les positions atteignables depuis la position passée en paramètre
%N'insère que les noeuds n'étant pas encore présents ou ceux ayant une valeur F inférieure à l'un des noeuds déjà présent
%-------------------------------------------------------------------------------------------------------------------------

insertNextPositionsInOpenList(PosParent) :-
           getAllAccessiblePositions(PosParent,ListPositions),
           insertNextPositionsInOpenList(PosParent,ListPositions).

insertNextPositionsInOpenList(_,[]) :- !.

%si le nouveau noeud n'est pas encore dans openList ni closedList, on l'ajoute
insertNextPositionsInOpenList(PosParent,[Pos|R]) :-
          creerNoeud(Pos,PosParent,Noeud), nb_getval(openList,OL), nb_getval(closedList,CL),
          not(member([Pos,_,_,_,_],OL)), not(member([Pos,_,_,_,_],CL)), nb_linkval(openList,[Noeud|OL]),
          insertNextPositionsInOpenList(PosParent,R),!.

%si le nouveau noeud n'est pas encore dans openList mais est dans closedList, on NE l'ajoute PAS
insertNextPositionsInOpenList(PosParent,[Pos|R]) :-
          creerNoeud(Pos,PosParent,_), nb_getval(openList,OL), nb_getval(closedList,CL),
          not(member([Pos,_,_,_,_],OL)), member([Pos,_,_,_,_],CL),
          insertNextPositionsInOpenList(PosParent,R),!.

%si le nouveau noeud est déjà dans openList, on l'ajoute si son F est inférieur au F du noeud déjà présent
insertNextPositionsInOpenList(PosParent,[Pos|R]) :-
          creerNoeud(Pos,PosParent,[Pos,PosParent,H,G,F]), nb_getval(openList,L),
          member([Pos,PosParent2,H2,G2,F2],L), F < F2 , retirerNoeud([Pos,PosParent2,H2,G2,F2],L,L2),
          nb_linkval(openList,[[Pos,PosParent,H,G,F]|L2]), insertNextPositionsInOpenList(PosParent,R),!.

%si le nouveau noeud est déjà dans openList, on ne l'ajoute pas si son F est supérieur au F du noeud déjà présent
insertNextPositionsInOpenList(PosParent,[Pos|R]) :-
          creerNoeud(Pos,PosParent,[Pos,PosParent,_,_,F]), nb_getval(openList,L),
          member([Pos,_,_,_,F2],L), F >= F2,
          insertNextPositionsInOpenList(PosParent,R).
%-------------------------------------------------------------------------------------------------------------------------
%-------------------------------------------------------------------------------------------------------------------------
```

vii. Prédicats de déplacement

Il existe 4 prédicats dits de « déplacement ». Un robot peut en effet se déplacer vers la gauche, la droite, le haut ou le bas. Il y a un prédicat par mouvement.
Il est important de ne pas oublier de tenir compte de la position des robots sur le plateau ainsi que du changement de position du robot au fur et à mesure des nœuds étudiés. Le prédicat buildPath se charge d'actualiser les variables globales de position des robots.

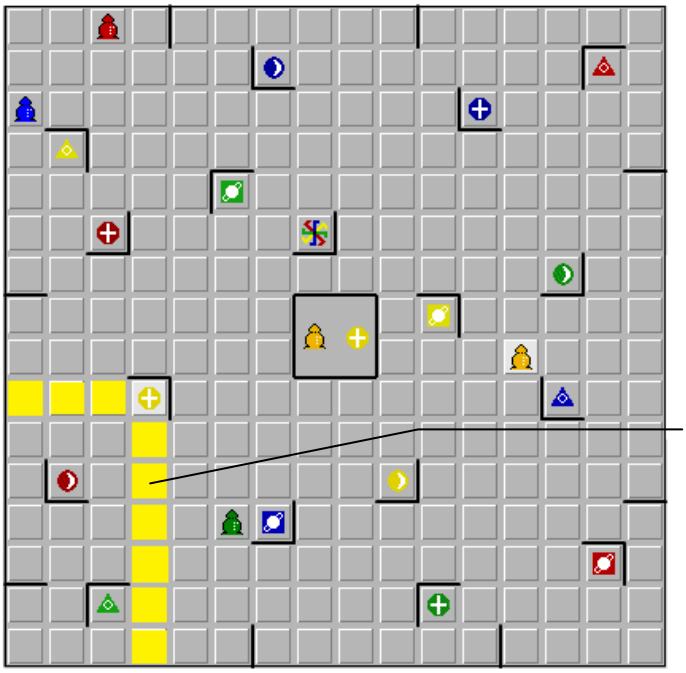
```
%-------------------------------------------------------------------------------------------------------------
%******* DEPLACEMENT VERS LA GAUCHE *******
%-------------------------------------------------------------------------------------------------------------
%dans le cas où la case courante contient un mur à gauche, la position reste inchangée
gauche(X,Y,X,Y) :- case(X,Y,_,_,1,_),!.

%dans le cas où la case voisine de gauche contient un robot, la position reste inchangée
gauche(X,Y,X,Y) :- X0 is X-1, nb_getval(rbtBleu,[X0,Y]),!.
gauche(X,Y,X,Y) :- X0 is X-1, nb_getval(rbtVert,[X0,Y]),!.
gauche(X,Y,X,Y) :- X0 is X-1, nb_getval(rbtJaune,[X0,Y]),!.
gauche(X,Y,X,Y) :- X0 is X-1, nb_getval(rbtRouge,[X0,Y]),!.


%sinon, on fait un appel récursif du prédicat gauche en se déplacement d'une case vers la gauche
gauche(X,Y,X2,Y) :- X1 is X-1, gauche(X1,Y,X2,Y).
```

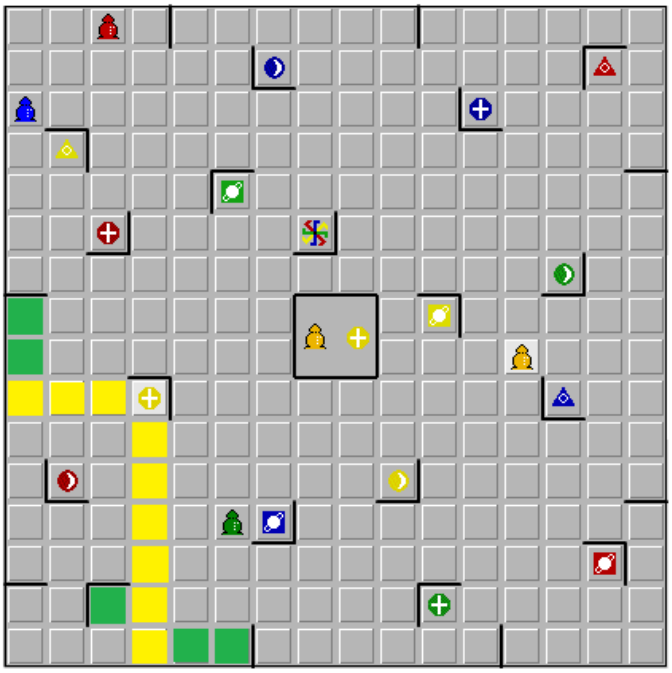
V. Méthode permettant de résoudre des cas plus complexes.

La solution implémentée déplace uniquement le robot devant atteindre la cible. Dans certaines configurations cette méthode peut nous empêcher de trouver une solution ayant moins de coups joués. Mais dans le pire des cas elle nous empêche tout simplement de trouver un quelconque chemin vers la cible.
Néanmoins, les paragraphes suivants ont pour but de proposer une stratégie simple qui pourrait permettre le déblocage de la majorité des situations. Prenons un exemple…



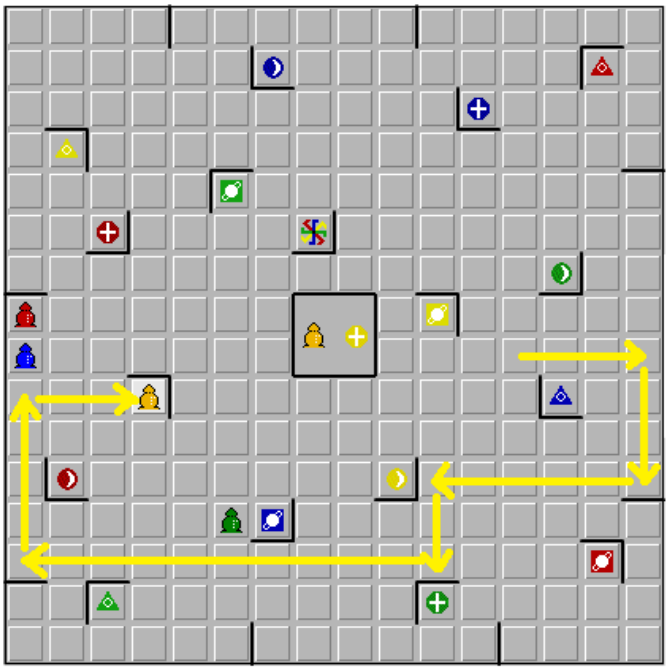
Ici, le robot jaune n'a aucun moyen d'atteindre la cible « + » jaune. En effet, si l'on analyse la situation de plus près, il n'y a aucun chemin qui permettrait de placer le robot jaune dans un des couloirs faisant face à la cible. Il manque en fait un ou plusieurs « obstacle ».

Impossible de placer le robot jaune dans l'une de ces cases, car aucun mur ou autre robot n'est placé le long de ces couloirs et permettrait au robot jaune de s'y loger.



Une solution consisterait donc à s'aider d'un, de deux ou des trois autres robots pour faire « obstacle » et permettre au robot jaune de se loger dans le couloir jaune. Les positions vertes représentent des positions facilement atteignables par les autres robots.

Il suffirait de rappeler le prédicat jeu_1robot avec une de ces nouvelles cibles et les différents robots pour placer des obstacles dans une de ces cases vertes, puis de tester une nouvelle fois si le robot jaune atteint sa cible avec la nouvelle configuration.



Voici une solution qui respecte la stratégie énoncée précédemment. Ce genre de méthode ne garantit malheureusement pas un nombre de coups minimum. Ici nous déplaçons d'abord les robots pouvant servir d'obstacles, puis finalement le robot devant atteindre la cible.

Or, après avoir testé le jeu dans diverses configurations nous avons remarqué que parfois, les déplacements des différents robots s'entrecroisent et s'aident mutuellement à diverses étapes du parcours. Cette méthode est d'un niveau apparemment encore plus complexe et n'a pas eu le temps d'être envisagée dans ce projet.

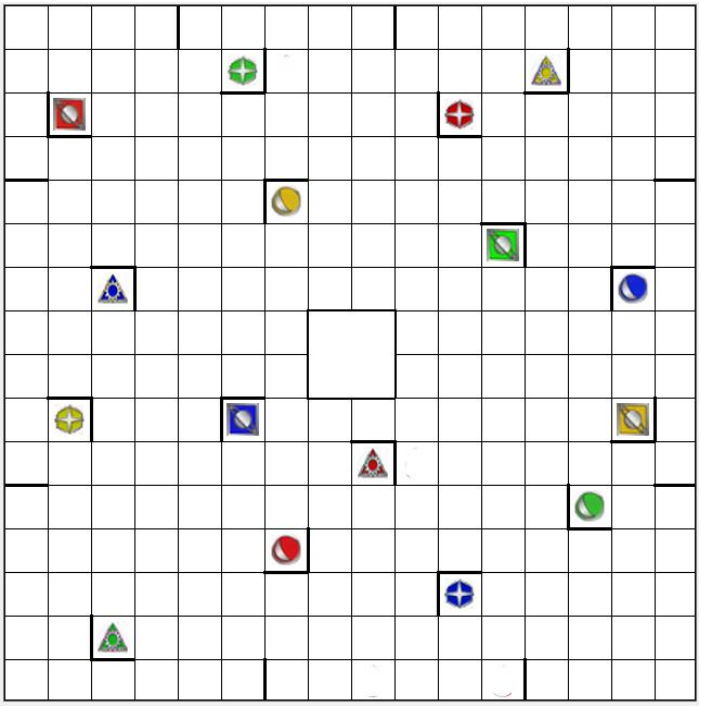
Implémentation de cette solution

La première alternative proposée a commencée à être implémentée, mais n'est pas totalement fonctionnelle dans l'avancée actuelle du projet.
Dans le cas où des positions pouvant potentiellement aider le robot à atteindre sa cible ont été identifié, on peut alors demander au prédicat « trouveBestRobotPos » quel robot peut atteindre une des positions avec le moins de coups possible.
Ce prédicat prend en paramètre une liste de positions et une liste de noms de robots. Il renvoie la cible et le robot pouvant l'atteindre avec le moins de coups possible.

Le code du prédicat trouveBestRobotPos peut être trouvé en annexe de ce rapport, mais n'a pas été joint à la solution finale.

VI. Interface graphique : communication C++/SWI-Prolog

a. Plateau fixe



Nous avons fait le choix d'utiliser un plateau de jeu fixe, autrement dit, les murs et cibles se trouveront toujours à la même place.

Pourquoi ?

Principalement, par souci pratique, pour être certain qu'il soit toujours possible de trouver un chemin menant un robot à la cible qui lui aura été désigné. Le plateau choisi s'inspire d'un de ceux testé sur http://www.ricochetrobots.com/RR.html

Le tracé du plateau est assez basique. Celui-ci est modélisé en background sous la forme d'un tableau de Case. La classe Case contient, elle, des données comme la présence ou non d'un robot ainsi que sa couleur, la présence ou non d'une cible avec sa couleur et son type, la présence ou non de mur, etc…

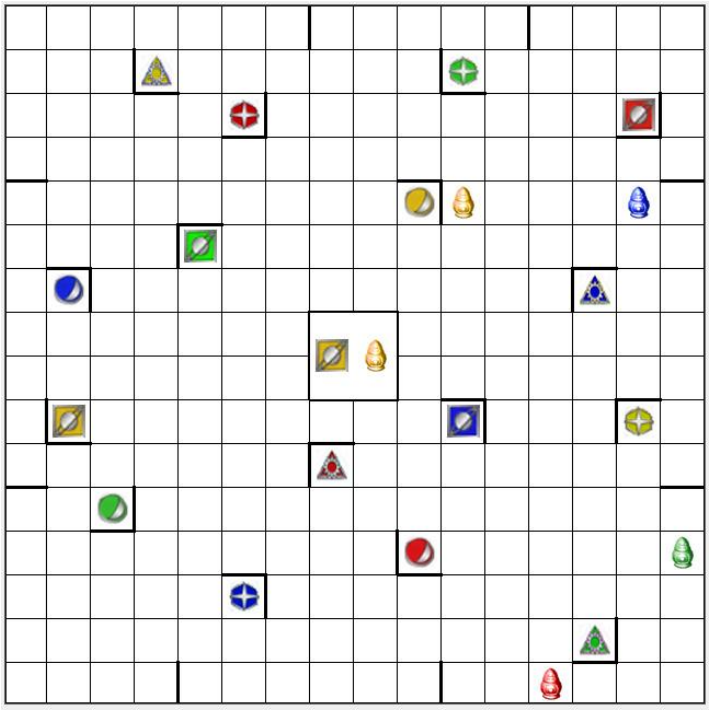
Toutes ces informations vont nous permettre de faire le tracé de la grille, le placement des cibles et des murs par rapport au tableau construit précédemment.
L'affichage se fait par simple dessin en récupérant l'objet Graphics du panel et en appliquant quelques formules simples pour obtenir les coordonnées.

b. Les robots placés aléatoirement

Par souci, évident, de faire varier les parties, nous avons du rendre le placement des robots aléatoire. Une fonction Random permet de générer des coordonnées aléatoires pour l'entrée dans le tableau des quatre robots.
Les robots ne doivent pas être placés sur des cibles ou au même endroit qu'un autre robot. Grâce aux attributs de l'objet Case, il nous est possible de le vérifier et donc d'éviter cette situation.
De même, les quatre cases du milieu du plateau ne doivent pas être accessibles à un robot. Un attribut active de la classe Case nous permet également de les exclure des cases disponibles.

c. Détermination du robot et de la case qu'il doit atteindre

Les coordonnées des robots et des cibles étant stockés dans des tableaux lors de l'initialisation du plateau de jeu, il reste à déterminer la couleur du robot et de la cible puis le type de la dite cible.
Comme pour le placement, un procédé aléatoire sera utilisé pour la couleur puis pour le type de cible. Le robot et la cible finale sélectionnés sont ensuite affichés au centre en information pour l'utilisateur.
La couleur du robot et les coordonnées de la case cible sont ensuite stockées dans des variables en vu de la résolution par l'intelligence artificielle.



d. La liaison C++/SWI Prolog

La solution
Nous allons présenter ici la solution apportée pour faire communiquer les deux langages que sont respectivement C++ et (SWI) Prolog.
Après des recherches et essais divers, nous sommes arrivés à la solution que l'on décrira ici, qui a semblé être la plus adapté mais également la plus simple à mettre en œuvre pour faire le lien entre les deux langages.

Nous avons utilisé des classes fournies avec SWI Prolog qui permettait d'utiliser Prolog avec du C++. Elle utilise le code des anciennes fonctions de lien entre C et Prolog.
Le principe est plutôt simple puisque grâce à ce code, nous créons une instance de la console de SWI Prolog. Les diverses fonctions utilisées ensuite nous permettent simplement de faire des appels, dans cette instance, de prédicats Prolog par le biais du C++.
Nous commençons donc par initialiser l'instance de la console Prolog avec :
PL_initialise(argc, argv);
Il a cependant été nécessaire plutôt d'inclure la librairie libpl.dll et quelques arguments que nous ne détaillerons pas ici.

Nous poursuivons ensuite comme si nous étions dans une console prolog.

```
try{
//PlCall( "consult(swi('plwin.rc'))" );
PlCall( "consult('case.pl')" );
} catch ( PlException &ex )
  { cerr << (char *) ex << endl;
}

try{
//PlCall( "consult(swi('plwin.rc'))" );
PlCall( "consult('Jeu_Rasende_roboter.pl')" );
} catch ( PlException &ex )
  { cerr << (char *) ex << endl;
}
try{
//PlCall( "consult(swi('plwin.rc'))" );
PlCall( "consult('jeu_1robot.pl')" );
} catch ( PlException &ex )
  { cerr << (char *) ex << endl;
  }
```

La compilation des fichiers Prolog grâce à la fonction « consult »

Les attributs de la fonction sont stockés dans un tableau de PlTermv pour être passer en argument de la fonction appelant le prédicat que l'on veut utiliser.
Les attributs ne peuvent pas être transmis sous forme de liste avec cette méthode. Il nous est donc autorisé les chaines de caractères et les nombres.

```
PlTermv av(13);
av[0]=coordonneeRobot[1][0]+1;
av[1]=16-coordonneeRobot[1][1];
av[2]=coordonneeRobot[2][0]+1;
av[3]=16-coordonneeRobot[2][1];
av[4]=coordonneeRobot[0][0]+1;
av[5]=16-coordonneeRobot[0][1];
av[6]=coordonneeRobot[3][0]+1;
av[7]=16-coordonneeRobot[3][1];
if(robotCible==0)
    av[8]="rbtRouge";
else if(robotCible==1)
    av[8]="rbtBleu";
else if(robotCible==2)
    av[8]="rbtVert";
else
    av[8]="rbtJaune";
av[9]=cibleX+1;
av[10]=16-cibleY;
```

```
PlQuery q("jeu_1robot", av);
int rval= PL_next_solution(q.qid);
```

La requête est lancée avec son nom et ses arguments grâce à la fonction PlQuery.

rval est le résultat de la requête récupéré plus haut.
Si elle échoue ou retourne « false », rval vaut 0, sinon rval est supérieur à 0.
Comme précédemment, on ne peut récupérer que des chaines de caractères et des nombres.
Ici, la liste des déplacements du robot sont sous forme de chaine de caractères.

```
if ( !(rval) )
{ term_t ex;

    if ( (ex = PL_exception(q.qid)) )
      PlException(ex).cppThrow();
    else{
        tb->Text = "Impossible";
        button2->Enabled = true;
        button1->Enabled = false;
    }
}else{
    try{
            s = gcnew String(((string) av[11]).c_str());
            tb->Text = s;
            tb2->Text= gcnew String(((string) av[12]).c_str());
            deplacement(s);
            button1->Enabled = false;
            /*System::Drawing::Rectangle rec = System::Drawing::I
            panel1->Invalidate(rec);*/
    }
    catch(PlTypeError &ex3){
    }
}
```

La chaine est ensuite exploitée et mise sous forme de tableau de coordonnées. Les déplacements sont appliqués au robot au rythme d'un déplacement par seconde pour laisser à l'utilisateur le temps de voir la solution de l'intelligence artificielle.

Problèmes et autres solutions étudiées

L'un des problèmes de cette solution a été la récupération des résultats. Ceux-ci s'affichaient d'abord avec un « write » dans le code Prolog mais il n'était pas possible de récupérer les informations de la sortie standard. La solution s'est trouvée dans le fait qu'il fallait que la solution soit stockée dans un des arguments de la fonction Prolog. La récupération se faisait alors directement dans l'argument.

Problème plus mineur, les incompatibilités de types entre le C++ managé et le C. Dans les autres solutions étudiées, la piste la plus sérieuse était dans la confection d'un code C faisant un lien spécifique avec nos fichiers Prolog. Il fallait ensuite compiler le tout sous forme d'une dll utilisable ensuite dans notre programme. Des problèmes divers, comme de « linckage » par exemple, qui nécessitait une plus grande connaissance de la méthode et sur lesquels les informations disponibles étaient limitées. La méthode utilisée ici l'a remplacé pour une utilisation plus simple.

VII. Guide d'utilisation de l'interface graphique

a. Installation de l'application sur un ordinateur Windows

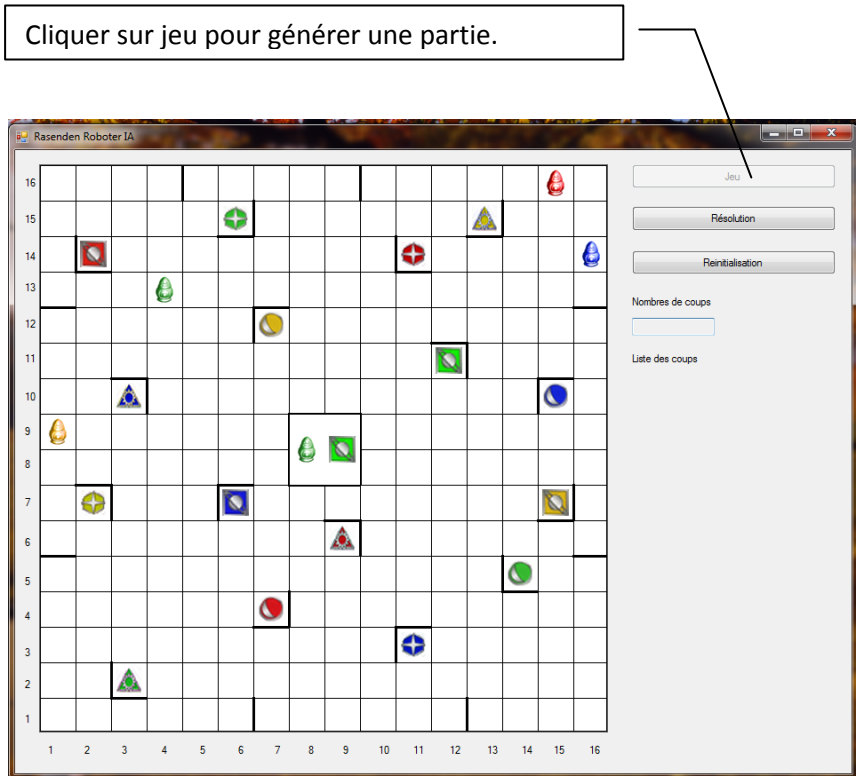
- Au préalable, il est nécessaire d'avoir installé **SWI Prolog** sur son ordinateur.

- Double-cliquer sur le fichier d'installation « **Setup** » contenu dans le dossier « Application », et suivre les instructions d'installation.

- Une fois l'installation terminée et avant de pourvoir utiliser l'interface graphique, modifier le fichier config.txt contenu dans '' C:\Program Files\ProjetIA\Rasenden Roboter IA ''.
  Y spécifier le chemin vers le dossier pl de SWI Prolog. Du type'' C:\Program Files\pl ''.
  Attention : Il est nécessaire de modifier les droits d'accès du fichier config.txt pour faire cette opération. Clique-droit sur le fichier -> propriétés -> Sécurité -> Utilisateurs
  -> Puis cocher « contrôle total »

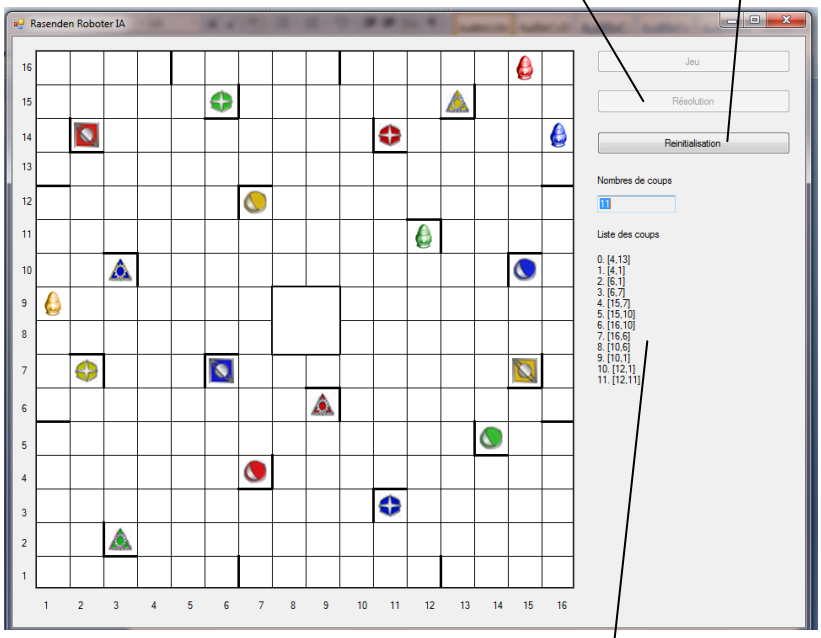
- L'application est prête à être utilisée, et peut être lancée en double-cliquant sur l'icône de raccourci placé sur le bureau.

b. Utilisation de l'application



Pour voir la solution du programme, cliquer sur « Résolution » et observer le déplacement du robot.

Pour générer une nouvelle partie



Récapitulatif du chemin parcouru pour atteindre la cible

VIII. Annexe

a. Code du prédicat « trouveBestRobotPos »

```
%trouveBestRobotPos( +ListePosition, +ListeRobots, +RBleu, +RVert, +RRouge, +RJaune, -Position, -Robot, -Chem, -NbCoups)
%renvoie le robot qui peut atteindre une des positions donnée avec le moins de coups possible

trouveBestRobotPos( Positions, [Rob], RBleu, RVert, RRouge, RJaune, [], aucun, [], 1000000) :-
                    trouveMeilleurePos( Positions, Rob, RBleu, RVert, RRouge, RJaune, [], _, [], _),!.

trouveBestRobotPos( Positions, [Rob], RBleu, RVert, RRouge, RJaune, Pos, Rob, Chemin, NbCoups) :-
                    trouveMeilleurePos( Positions, Rob, RBleu, RVert, RRouge, RJaune, Pos, Rob, Chemin, NbCoups),!.

trouveBestRobotPos( Positions, [Rob|Reste], RBleu, RVert, RRouge, RJaune, Pos1, Rob1, Chemin1, NbCoups1) :-
                    trouveMeilleurePos( Positions, Rob, RBleu, RVert, RRouge, RJaune, Pos1, Rob1, Chemin1, NbCoups1),
                    trouveBestRobotPos( Positions, Reste, RBleu, RVert, RRouge, RJaune, _, _, _, NbCoups2),
                    NbCoups1 =< NbCoups2,!.

trouveBestRobotPos( Positions, [Rob|Reste], RBleu, RVert, RRouge, RJaune, Pos2, Rob2, Chemin2, NbCoups2) :-
                    trouveMeilleurePos( Positions, Rob, RBleu, RVert, RRouge, RJaune, _, _, _, NbCoups1),
                    trouveBestRobotPos( Positions, Reste, RBleu, RVert, RRouge, RJaune, Pos2, Rob2, Chemin2, NbCoups2),
                    NbCoups1 > NbCoups2.
```

```
%-------------
trouveMeilleurePos( [Pos], Rob, RBleu, RVert, RRouge, RJaune, [], aucun, [], 1000000) :-
                   not(jeu_1robot(RBleu, RVert, RRouge, RJaune, Rob, Pos, _, _)),!.

trouveMeilleurePos( [Pos], Rob, RBleu, RVert, RRouge, RJaune, Pos, Rob, Chemin, NbCoups) :-
                    jeu_1robot(RBleu, RVert, RRouge, RJaune, Rob, Pos, Chemin, NbCoups),!.

trouveMeilleurePos( [Pos|R], Rob, RBleu, RVert, RRouge, RJaune, Pos, Rob, Chemin1, NbCoups1) :-
                    jeu_1robot(RBleu, RVert, RRouge, RJaune, Rob, Pos, Chemin1, NbCoups1),
                    trouveMeilleurePos( R, Rob, RBleu, RVert, RRouge, RJaune, _, _, _, NbCoups2),
                    NbCoups1 =< NbCoups2 ,!.

trouveMeilleurePos( [Pos|R], Rob, RBleu, RVert, RRouge, RJaune, Pos2, Rob2, Chemin2, NbCoups2) :-
                    jeu_1robot(RBleu, RVert, RRouge, RJaune, Rob, Pos, _, NbCoups1),
                    trouveMeilleurePos( R, Rob, RBleu, RVert, RRouge, RJaune, Pos2, Rob2, Chemin2, NbCoups2),
                    NbCoups1 > NbCoups2,!.

%si jeu_1robot ne trouve pas de solution alors NbCoups2 est la meilleure solution
trouveMeilleurePos( [Pos|R], Rob, RBleu, RVert, RRouge, RJaune, Pos2, Rob2, Chemin2, NbCoups2) :-
                    not(jeu_1robot(RBleu, RVert, RRouge, RJaune, Rob, Pos, _, _)),
                    trouveMeilleurePos( R, Rob, RBleu, RVert, RRouge, RJaune, Pos2, Rob2, Chemin2, NbCoups2).
```