Projet IA41 : Rasende Roboter

BLEIN / JAFFALI / JARDINO / RUFFY
UTBM P2015

Table des matières

I. Introduction

Nous sommes 4 étudiants de l'UTBM en semestre 1 et 2 en filière informatique. Ce projet est réalisé dans le cadre de l'UV IA41 afin de mobiliser les compétences acquises au cours du semestre.

L'intitulé de l'UV est : Intelligence artificielle : concepts fondamentaux et langages dédiés, UV où nous étudions différentes notions tels que l'algorithme min/max, ou l'utilisation de langages dédiés à l'intelligence artificielle.

Nous avons eu le choix entre différents sujet et nous sommes finalement décidés pour les Rasende Roboter, nous avons donc dû réaliser en Prolog un programme de résolution de problèmes.

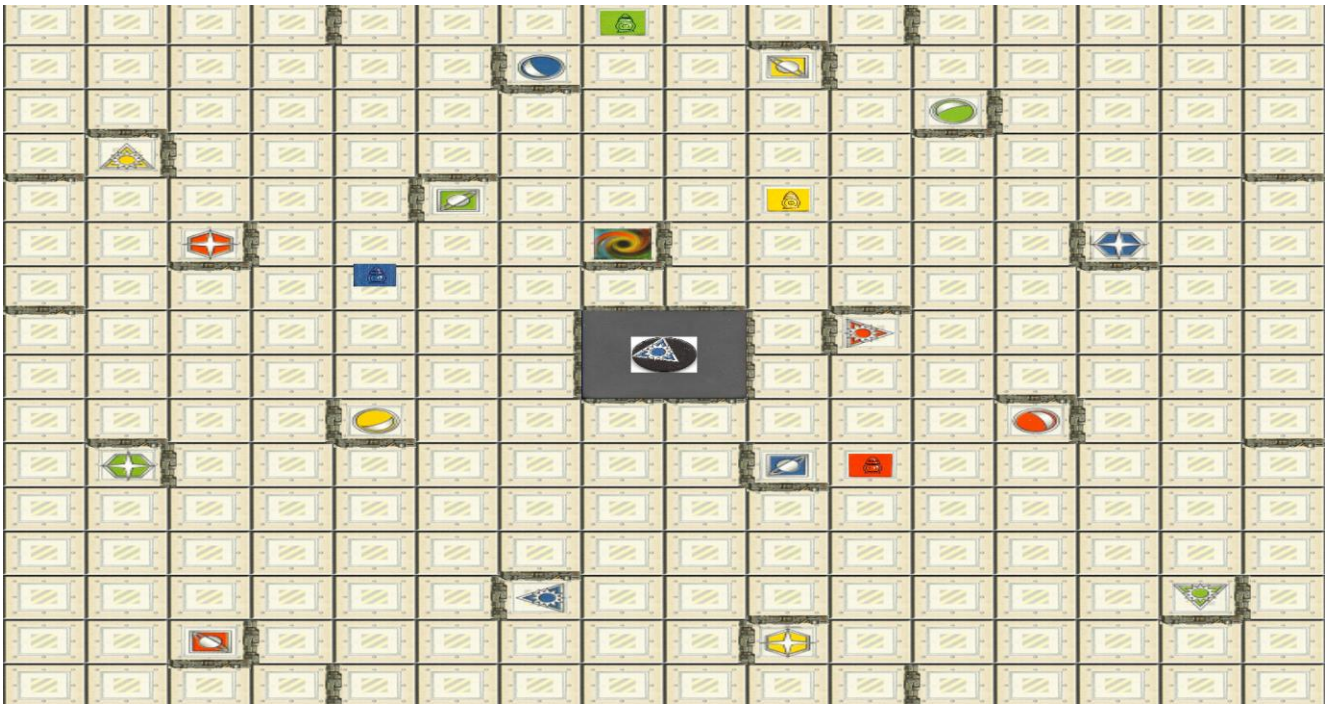
Nous présenterons dans un premier temps le sujet, puis nous évoqueront la solution que nous voulions implémenter à l'origine, nous parlerons ensuite de la solution adopté que nous expliquerons, nous enchaineront avec les améliorations possibles et finiront avec les difficultés rencontrées.

Veuillez trouver en annexe le code de certaine fonction expliquées dans la partie correspondante du rapport.

II. Présentation du sujet

Le jeu est constitué d'un plateau, comprenant des cases cibles, différents murs, et de robots. Le but du jeu étant de trouver la combinaison de coup utilisant le moins de déplacement possible pour amener sur une cible donnée, un robot avec une position initiale donnée.

Les robots peuvent se déplacer vers le haut, le bas, la gauche et la droite, et s'arrêtent uniquement contre des murs. Notre travail a donc été de codé une intelligence artificielle qui nous renvoie la combinaison de coup la plus rapide pour atteindre la cible. Dans le jeu original, un robot peu avoir besoin d'un autre robot pour pouvoir atteindre sa cible.



III. Notre première approche

Dans un premier temps, nous voulions utiliser un BFS en partant de la cible à atteindre pour construire le chemin permettant de rejoindre le robot. Mais nous nous sommes ensuite rendus compte que cela est difficilement réalisable. Nous voulions qu'après le premier mouvement si la case n'est pas accessible nous appelions un deuxième robot pour nous aider à nous y rendre, puis continuer à revenir jusqu'à notre robot, chose complexe. En effet, comme vous pouvez le voir sur ce dessin ci-dessous, il y a forcément des chemins qui ne sont pas réalisable dans les deux sens.

Dans ce cas, la cible à atteindre est la cible jaune en dessous du robot bleu. Si l'on cherche un chemin à partir de la flèche rouge à gauche, on trouvera la cible en 2 coups alors que si nous partons de la cible jaune nous partirons vers le haut (flèche verte). Suite à cette découverte nous avons décidé de faire un BFS à partir du point de départ. Il aurait été possible de partir de l'arrivée en rebondissant sur chaque case contenant au moins 1 mur et testé à chaque déplacement élémentaire si l'on traverse la position du robot souhaité. Chose qui semble très complexe.

IV. Solution retenue

Tout d'abord nous voulions une solution permettant à coup sûr de trouver une solution, au risque de restreindre les cas solvables. Nous nous sommes donc tournés vers un BFS avec stockage des cases parcourues qui nous retournera toujours la meilleure solution et ne passera pas deux fois par la même case. Afin de pouvoir délivrer dans le temps imparti une solution qui fonctionne nous nous sommes limités au cas des cases pouvant être atteinte par un seul robot.

Nous commençons par initialiser la position des robots, puis nous cherchons quel est le robot devant atteindre la cible donnée et nous prenons la position de départ et d'arrivée. Et nous partons dans notre BFS avec la position de départ et d'arrivée.

Etant donnée une position donnée nous cherchons et la filtrons avec une liste des cases déjà parcourue. Nous stockons les fils possibles à la fin d'une liste globale comprenant la position père et la destination fils, pour avoir les prochains déplacements à effectuer. Une fois une case traitée, nous prenons la prochaine case dans la liste des déplacements à effectuer, nous la retirons de cette liste, copions la case dans la liste des cases visité et nous copions le couple père fils dans la liste qui va nous permettre de remonter.

Une fois la case arrivée atteinte, nous remontons de l'arrivé vers le départ en prenant le père de chaque élément (comme une case ne peut être visité qu'une seule fois elle ne peut avoir qu'un seul père alors qu'un père peut avoir plusieurs fils) et nous stockons les positions successive.

Après avoir le numéro du robot à déplacer et maintenant la liste des positions successive, nous pouvons créer la liste des déplacements en ce comparant la position d'une case et de ça successeur. Nous avons donc maintenant la liste des déplacements nécessaires.

Si jamais la liste des déplacements à effectuer est vide c'est que nous ne pouvons plus bouger vers une nouvelle casse le programme renvoie donc la liste de mouvement vide.

Etant donné qu'à chaque position en partant du robot nous essayons tous les déplacements possibles, tous les cas sont traités pour l'utilisation d'un seul robot.

V. Explication de notre solution

Le but de la recherche de solution est de pouvoir fournir une liste de déplacement pour les robots à partir d'un stimulus reçu par le prédicat move, lui-même appelé par think.

On part alors d'une configuration connue du plateau, des robots et de la cible à atteindre, et on voudrait fournir la liste des mouvements du robot pour atteindre convenablement la cible. Dans le cas où aucune solution n'est possible, on retournera alors une liste de déplacement nulle.

Codage du plateau

Afin de modéliser le plateau nous avons décidé de coder chaque case du plateau en indiquant pour chacune les positions éventuelles des murs autour. Enfin on repère chaque cible en associant des coordonnées à chaque numéro de cible.On définit pour cela deux prédicats : *case/6 et target/2.*

Gestion des déplacements

Afin de pouvoir parcourir notre graphe modélisant notre plateau, il faut d'abord définir les règles relatives au déplacement d'un nœud à un autre. Les règles du jeu stipulent que l'on ne peut se déplacer qu'en ligne droite, en ne s'arrêtant que lorsqu'un obstacle (mur ou robot) est rencontré. L'idée est donc de définir un prédicat qui, à partir d'un nœud du graphe (une case dans le plateau), nous renvoie la liste des nœuds accessible conformément aux règles du jeu.

Pour ce faire on décompose le problème en un test de déplacement dans chacune des 4 directions permises : haut, bas, gauche et droite. A partir d'une case, on vérifie si l'on peut se déplacer dans une direction précise, case par case. On vérifier alors d'abord s'il y a un mur dans la case adjacente, sinon on vérifie s'il y a un robot, sinon on lance récursivement notre prédicat sur la case adjacente accessible. Une fois un obstacle atteint, on remonte jusqu'à la case d'appel du prédicat, et l'on récupère les coordonnées de la case accessible dans la direction choisie. En combinant les prédicats intermédiaires pour chaque direction, on arrive à construire une liste de nœuds accessible, différents de celui d'origine.

Lancement de move/2

L'appel du prédicat move(+Stimulus,-Actions) va engendrer l'appel de plusieurs prédicats qui se chargeront de résoudre les sous-problèmes induits par la recherche de cette solution, cette liste de mouvements.

On commencera tout d'abord par sélectionner le robot à bouger en fonction de la cible à atteindre, puis on lance ensuite l'algorithme de parcourt du graphe jusqu'à atteindre la solution. Une fois la solution trouvée on remonte vers chaque parent pour obtenir la liste des positions successive et la convertir en une liste de robots et d'actions qui peut être interprétée.

Choix du robot mobile

Comme expliqué précédemment, nous ne pouvons résoudre que les déplacements totaux qui ne nécessitent que le mouvement du robot de la même couleur que la cible. On ajoutera aussi que pour la cible multicolore, on fixe le robot ayant le droit de l'atteindre : le robot bleu. On définit alors un prédicat selectRobot/10 qui prend en paramètres les coordonnées des chaque robot, le numéro de la cible, et qui nous renvoie la position du robot à déplacer, en fixant les autres robots comme obstacle, et retirant le robot mobile des obstacles (pour ne pas qu'il "rebondisse" sur sa position initiale).

Algorithme de parcours : BFS

Le BFS est l'algorithme de parcours en largeur d'un graphe. Pour ce faire on manipule 2 prédicats principaux : go/2 et path/1.

Le prédicat go prend en paramètre le point de départ et le point d'arrivée et va lancer la recherche en largeur dans le graphe. Il renverra "No" s'il ne trouve pas de solution et "Yes" s'il en trouve une, en stockant les relations entre les nœuds et les nœuds parcourus à l'aide de prédicats et de listes en variables globales.

Le prédicat state_record/3 se charge justement de faire le lien de "parenté" entre les nœuds. Il associe pour chaque nœud parcourut le nœud parent duquel on vient. On obtient donc un couple du type [Etat,Père] pour chaque nœuds du graphe.

Dans le prédicat go, en fait, on initialise donc la liste de parcours (FIFO) et de nœuds visités à liste nulle. On définit le premier état du graphe en associant les coordonnées du robot et la liste vide, afin de modéliser le fait qu'il n'y a pas de parent au premier état qui est le lancement de l'algorithme. On ajoute en queue de FIFO notre premier état, et on lance le prédicat path/1 qui va lancer la recherche récursive en largeur dans le graphe.

Le prédicat path extrait et supprime le premier élément de la queue FIFO tout d'abord. Il lance ensuite le prédicat déplacement pour trouver tous les éléments accessible à partir de cette tête de cette case. On filtre ensuite cette liste de déplacements en supprimant les cases déjà parcourues pour ne pas entrer les revisiter dans le parcours du graphe. On ajoute ensuite cette liste de déplacements possibles filtrées en queue de FIFO : à chaque fois que l'on ajoute une case à visiter en queue, on notifie avec state_record quel est le père de cet état, ici en l'occurrence l'état extrait en tête de FIFO par path. Lorsqu'un attribut est extrait de la liste FIFO il est ajouté à la liste FIN afin de pouvoir retracer le parcours ensuite.

On relance ensuite le même prédicat path avec en paramètre l'état d'arrivée à atteindre. On répète cela jusqu'à ce que l'état extrait de la queue FIFO corresponde à l'état d'arrivée en paramètre de path. Lorsque c'est le cas, le prédicat path renvoie "Yes", et on remontera dans le prédicat go qui renverra à son tour "Yes". Si la liste FIFO devient vide avant la fin c'est-à-dire que plus aucune case non visité ne peut le devenir, on renvoie "No".

Déduction du chemin

Une fois la cible atteinte, et donc la liste fin complète ainsi que les liens de parentés modifiés et mis en place par les prédicats go et path, il convient de retrouver le chemin des cases du plateau pour aller du robot à la cible de la même couleur.

On choisit alors de définir deux prédicats recherchesolutiondepart/1 et trouver/3.

Le premier va remonter au point de départ à partir du point d'arrivée. A chaque passage dans le prédicat recherchesolutiondepart, on détermine qui est son père, on lance récursivement la fonction sur le père, et après l'appel on ajoute à la liste notre état. Le prédicat ira donc récursivement jusqu'à la condition d'arrêt qui est qu'il trouve un père égal à la liste nulle, ce qui implique forcément que nous sommes à l'état de départ. On remontera alors la liste de récurrence en ajoutant au fur et à mesure les éléments à la liste des solutions, conservée en variable globale. C'est le prédicat trouve qui dans la liste fin recherche le père de l'état qui lui est envoyé.

Détermination de la succession de déplacements

A partir de la liste des cases à parcourir dans l'ordre pour se déplacer du départ à l'arrivée, on lance un parcours cette liste en construisant une autre liste contenant le numéro du mouvement à effectuer entre chaque couple de cases adjacentes. Pour ce faire, on utilise un prédicat convertisseur/3, qui à partir de deux coordonnées de cases renvoie le nombre de 1 à 4 correspondants au déplacement à faire entre ces deux cases, ceci en vérifiant la différence sur l'abscisse et l'ordonnée des deux cases.

Une fois la liste des directions établie, il convient d'intercaler entre chaque numéro de déplacement le numéro du robot à déplacer pour pouvoir obtenir la liste finale des actions à réaliser. On récupère alors, à partir d'une variable globale mise en place par le prédicat selectRobot, le numéro du seul robot à bouger dans notre configuration de lancement de move.

C'est alors le prédicat finalement/3 qui se charge de nous renvoyer la liste finale des déplacements de la forme : Movements = [R1,D1, R2,D2, ..., Rn,Dn], où chaque Ri désigne le robot à déplacer et chaque Di représente la direction associée à chaque déplacement de robot Ri.

VI. Possibilités d'améliorations

Notre programme fournit la solution pour résoudre une grande partie des cas solvables nécessitant le mouvement d'un seul robot. Cependant on peut se demander quelles sont les améliorations possibles à notre programme, et qu'aurait-il fallu implémenter en plus pour améliorer ce dernier.

Tout d'abord il apparait important de pouvoir résoudre toutes les configurations de cibles et de plateau, mais en utilisant cette fois ci toutes les ressources disponibles en termes de robots. Une amélioration serait donc la gestion de déplacement de plusieurs robots pour amener la solution.

Une autre idée d'amélioration serait de pouvoir proposer la meilleure solution possible de déplacement, pour un seul robot tout d'abord, et pour plusieurs si la dernière amélioration proposée est implémentée. En effet en utilisant un BFS, on aura accès à toutes les solutions possibles par un parcours en largeur et récoltant tous les chemins qui nous mènent à l'arrivée, ceci n'étant pas possible en A* car comme choisissant toujours le meilleur nœud par heuristique, on peut passer à côté de la meilleure solution si l'heuristique n'est pas assez pointue (comme c'est le cas de la distance de Manhattan). On pourrait alors évaluer le nombre de coups de chaque solution remontée par le BFS et choisir la moins coûteuse et donc s'assurer de gagner le tour de jeu assez rapidement face à des joueurs humains, étant donné la différence de vitesse de calcul entre l'homme et la machine.

Enfin il aurait moins coûteux en terme de calculs et de mémoire d'implémenter l'algorithme de parcours A*, qui retourne souvent l'une des meilleurs solutions, ceci dépendant en fait du choix judicieux de l'heuristique. L'implémentation de A* aurait probablement permis de trouver plus rapidement la du parcours.

VII. Difficultés rencontrés

Malgré la réussite finale de notre programme, pour atteindre l'objectif que nous nous étions fixés, les difficultés rencontrées ne nous ont pas laissé le temps d'aller plus loin dans notre sujet afin de répondre complètement au sujet avec la coopération des robots.

Tout d'abord malgré la compréhension du principe de BFS théorique, l'implémentation en Prolog ne fut pas chose aisé et nous demanda plusieurs phases de debugging.

La phase de debugging qui est facilitée par les affichages rapidement paramétrables afin de trouver l'origine des problèmes, le logiciel de traitement de texte que nous utilisions : Notepad ++ ne permettais pas d'avoir un code présenté de manière à différencier les différents éléments. Par exemple une couleur pour les ',' ou le signalement d'une ligne incohérente.

Le temps également de comprendre les cas d'utilisation des listes global, et les cas où la recopie est nécessaire nous a également pris du temps.

Pour finir l'utilisation du member() que nous pensions capable de trouver une liste à l'intérieur d'une autre nous a menés à chercher une erreur autour de lui avant de se rendre compte plus tard qu'il ne gérait pas ce cas et de recoder nous-même la solution à notre besoin.

VIII. Conclusion

Durant la construction de ce projet, nous avons progressivement réalisé la difficulté de celui-ci. Même si les notions théoriques utiles pour ce projet sont connues et maîtrisées, les adapter à notre projet et les utiliser sur prolog c'est révélé compliqué. Nous avons donc passé beaucoup de temps à travailler sur l'élaboration des prédicats et leur implémentation.

Le programme nous permet de faire bouger un robot pour qu'il atteigne une cible de même couleur. Nous n'avons pas codé de coopération entre les robots. S'il ne peut pas atteindre une cible seul, il ne fait rien et attend la prochaine.

Ce projet nous a, tout de même, permit de nous confronter à la confection d'une IA et d'en apprendre plus à ce sujet. De ce fait, je pense que les projets sont nécessaires afin de réaliser un peu mieux les enjeux et l'utilité des IA. De plus, nous avons encore pu travailler au sein d'un groupe ce qui nous prépare encore mieux à notre stage futur.

IX. Annexes

a. Prédicat move/2

```
%--------------------------PREDICAT MOVE-------------------------

%Prédicat move du jeu
%move( [0,0,0,0,TargetNb,CoordBleu,CoordVert,CoordJaune,CoordRouge],
%      ListeAction).

move([0,0,0,0,TargetNb,XB,YB,XV,YV,XJ,YJ,XR,YR],Actions):-
    target(TargetPos,TargetNb),
    selectRobot(XB,YB,XV,YV,XJ,YJ,XR,YR,TargetNb,RobotPos),

    go(RobotPos,TargetPos),

        nb_setval(solutionList,[TargetPos]),
        recherchesolutiondepart(TargetPos),
    nb_getval(solutionList,Liste),

    convertir(Liste,TargetPos,NewList),

    nb_getval(numRobot,N),
    finalement(NewList,N,Actions),
    format("~n Actions ~W~n",[Actions]),!.

move(_,[]).
```

C'est le prédicat global qui renvoie la liste d'action en prenant en entrée la configuration du plateau, le numéro de la cible et les positions des robots.

Il lance les prédicats pour copier la position de la cible, et lance les prédicats successifs pour la résolution.

b. Prédicat go/2

```
%-----------

go(Racine, Arrivee) :-
    state_record(Racine, [], Init),

    nb_setval(finList,[]),
    nb_setval(fifoList,[]),
    nb_setval(visitedList,[]),
    nb_setval(solutionList,[]),
    nb_getval(fifoList,L),


    ajouterElementEnQueueDeFifo(Init)
nb_getval(fifoList,F),


    path(Arrivee),!.
```

Go réinitialise les valeurs des listes globales à vide pour effacer le traitement précédent et lance path. Il prend en entrée le départ et l'arrivée.

c. Prédicat path/1

```
%-----------

path(Arrivee) :- nb_getval(fifoList,[]),
                 write('Graph parcouru, aucune solution'),nl.

%lorsque on l'a trouvé
path(Arrivee) :-

    copyTeteDeFifo(Next_record),

    state_record(Arrivee, _, Next_record),
    nb_getval(fifoList,Hey),

    nb_setval(dernier,Next_record).

%tant qu'on l'a pas trouvé
path(Arrivee) :-

    extraireTeteDeFifo(Next_record),

    state_record(State, Pere, Next_record),


    ajouterEnVisited(State),


    deplacement(State,Children),

    filtre(Children,Children2),

    ajouterListeEnQueueDeFifo(Children2,State),

    path(Arrivee).
```

Le prédicat path gère la création du chemin depuis le départ jusqu'à l'arrivée il prend en entrée l'arrivée. L'ensemble des cases parcourues avec leur parents sont stockés dans une liste.

d. Prédicat déplacement/2

```
deplacement(Actuel,L):- deplacement(Actuel,L,1).

deplacement(Actuel,[H|R],1):-move_rectiligne(Actuel,H,_,_,_),H\=Actuel,deplacement(Actuel,R,2),!.
deplacement(Actuel,L,1):-move_rectiligne(Actuel,Actuel,_,_,_),deplacement(Actuel,L,2).

deplacement(Actuel,[B|R],2):-move_rectiligne(Actuel,_,B,_,_),B\=Actuel,deplacement(Actuel,R,3),!.
deplacement(Actuel,L,2):-move_rectiligne(Actuel,_,Actuel,_,_),deplacement(Actuel,L,3).

deplacement(Actuel,[G|R],3):-move_rectiligne(Actuel,_,_,G,_),G\=Actuel,deplacement(Actuel,R,4),!.
deplacement(Actuel,L,3):-move_rectiligne(Actuel,_,_,Actuel,_),deplacement(Actuel,L,4).

deplacement(Actuel,[D],4):-move_rectiligne(Actuel,_,_,_,D),D\=Actuel,!.
deplacement(Actuel,[],4):-move_rectiligne(Actuel,_,_,_,Actuel).

mov(A,B):-deplacement(A,L),member(B,L).
```

Le prédicat déplacement prend en entrée une position, et retourne les positions accessibles.

e. Prédicat recherchesolution/1

```
recherchesolutiondepart(TargetPos):-TargetPos = [],nb_getval(solutionList,[T|L2]),nb_setval(solutionList,L2),!.
recherchesolutiondepart(TargetPos):-
    nb_getval(solutionList,Liste2),
    nb_getval(finList,FIN),
    trouver(TargetPos,Y,FIN),
    append([Y],Liste2,Liste3),
    nb_setval(solutionList,Liste3),
    recherchesolutiondepart(Y).
```

Le prédicat de recherche de solution prend en entrée l'arrivée, et insère dans une liste globale la succession de position prises. Elle se sert de trouver pour lui renvoyer la case précédente

f. Prédicat trouver/3

```
trouver(X,Y,[[X,Y]|_]):-!.
trouver(X,Y,[_|B]):-trouver(X,Y,B).
```

Le prédicat trouver prend en entrée une position et une liste de couple perd fils et renvoie le père.

g. Prédicat convertir/3 et convertisseur/3

```
convertir([X],X,[]).
convertir([A,Z|B],X,[C|D]):-convertisseur(A,Z,C),convertir([Z|B],X,D).

convertisseur([X|Y],[T|U],VAL):- T>X, VAL is 1 .
convertisseur([X|Y],[T|U],VAL):- Y>U, VAL is 2 .
convertisseur([X|Y],[T|U],VAL):- X>T, VAL is 3 .
convertisseur([X|Y],[T|U],VAL):- U>Y, VAL is 4 .
```

Convertir et convertisseur sont deux prédicats qui à partir d'une liste de positions successives, en fonction des différences entre coordonnées renvoie la liste des déplacements à effectuer.

h. Exemple

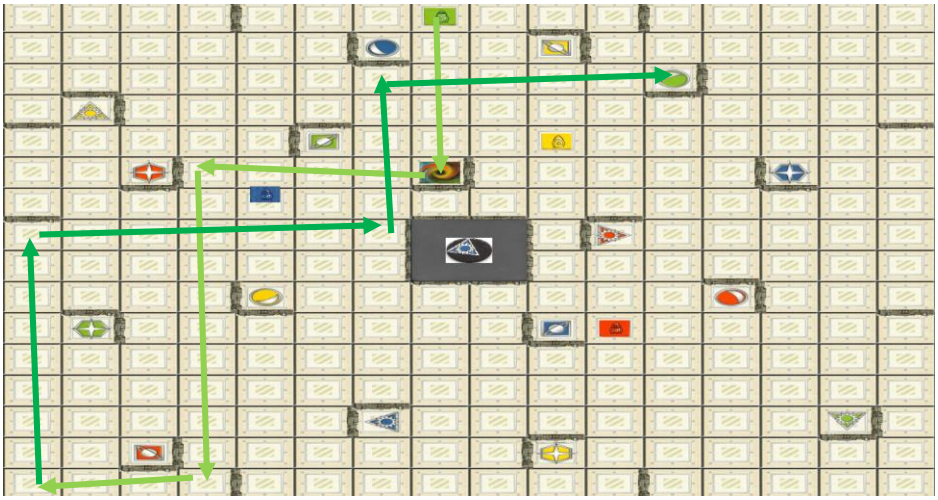
Lancement du prédicat move pour :

- la position de plateau codée : 0, 0, 0, 0.
- le numéro de la cible : 5
- les positions des robots :
  * Bleu : 5,7 * Vert : 8,1 *Jaune : 10,5 et *Rouge : 11,11

```
move([0,0,0,0,5,5,7,8,1,10,5,11,11],Actions).

Actions [1, 4, 1, 3, 1, 4, 1, 3, 1, 2, 1, 1, 1, 2, 1, 1]
```

On a donc pour déplacement du robot vert : bas, gauche, bas, gauche, haut, droite, haut, droite.



On a donc bien une solution en 8 coups qui est le nombre de coup minimum avec un seul robot.