Patrick CHALARD
John Eric SANCHEZ SUAREZ

Bordeaux INP ENSC

Intelligence artificielle

Rapport de projet
Rush Hour

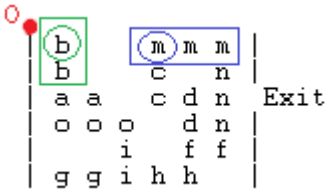
Introduction

Ce projet de deuxième année à l'ENSC s'inscrit dans le module d'intelligence artificielle. Il vise à créer un jeu de réflexion de type Rush Hour par le biais du langage Prolog.
Le projet a été réalisé par John Eric Sanchez Suarez et Patrick Chalard.

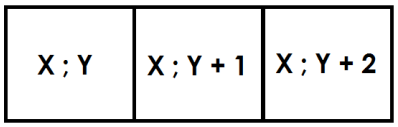
Représentation de la grille et des voitures

La grille de jeu en deux dimensions composée de lignes et de colonnes a pour origine le coin supérieur gauche. Les lignes équivalent aux indices X et les colonnes aux indices Y car par convention les lignes sont traitées en premier et X vient avant Y.

Les voitures occupent 2 ou 3 cases de la grille, à l'horizontale ou à la verticale. Ces voitures sont manipulées à partir des coordonnées de leur case la plus à gauche, ou de la plus en haut selon leur orientation.



Grille de jeu



Voiture horizontale

Nous avons choisi de représenter graphiquement la grille directement dans la console en utilisant des lettres pour les voitures et des barres horizontales pour les murs. Etant donné que le résultat nous a paru assez lisible, nous avons préféré concentrer notre temps sur le jeu en lui-même plutôt que de chercher des moyens externes à Prolog pour faire des interfaces graphiques plus sophistiquées.

Affichage de la grille (*show*)

L'utilisation de matrices sur Prolog passe notamment par des listes. Cependant, l'implémentation des listes sur Prolog n'étant pas évidente pour nous, nous avons préféré essayer une autre option qui ressemble un peu plus à la façon dont on afficherait une

matrice dans un langage impératif (comme C# par exemple), c'est-à-dire, la combinaison des boucles. Même si les boucles ne sont pas très utilisées sur Prolog, il est possible de les implémenter en appliquant de la récursivité. Un exemple très simple sur lequel nous avons basé cette démarche est l'affichage des chiffres de M à N qui peut être écrit de la manière suivante :

```
list(M,N) :- M<N, nl, write(M),
      NewM is M+1, list(NewM,N).
```

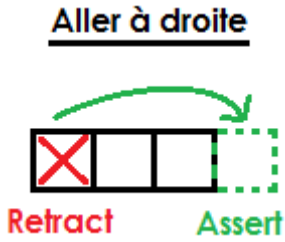
L'inconvénient de cette méthode était l'instruction pour pouvoir sortir de la boucle, car une fois que les règles qui limitent les lignes et les colonnes à 6 ont été appliquées, cela retournait faux comme réponse, plutôt que de continuer avec l'affichage de la ligne suivante.
Pour résoudre ce problème on a dû implémenter des règles qui traitent le cas où la valeur à afficher dépasse les limites de la grille et coupent ainsi simplement son exécution pour passer à ligne suivante ou pour afficher le message qui demande la saisie des valeurs dans le cas où la dernière ligne a déjà été affichée.

Les prédicats (faits) que l'on cherche correspondent aux faits **pos(X,Y,Car)** pour lesquels on prend la valeur de **Car** (une lettre) et l'affiche dans la position donnée. Pour les cases où il n'y a pas de voiture, on affiche un double espace afin de respecter la forme de la grille. Les règles '*show'* prennent aussi un paramètre **CarSelected** pour les cas où l'utilisateur a choisi la voiture à bouger il doit être capable de distinguer son affichage avec le caractère " ***** ".

Déplacements

Comme expliqué plus haut, les voitures sont manipulées par leur point en haut/à gauche. Lorsqu'un déplacement dans une direction donnée est demandé par le joueur, il convient de vérifier si le déplacement est possible avant de l'effectuer.
Ainsi, pour un déplacement vers le haut ou la gauche, la case directement en haut ou à gauche est analysée pour déterminer si elle est occupée.
Quand il s'agit d'aller vers le bas ou la droite, il faut tenir compte de la longueur de la voiture. Par exemple pour se déplacer vers la droite, on regarde si la case située deux cases à droite de l'origine de la voiture n'est pas occupée dans le cas d'une petite voiture. Alors que si c'est une grande voiture, on regardera la case située trois cases à droite de l'origine de cette voiture.

Pour effectuer le déplacement, une seule case de la voiture est “déplacée”, la case à l’opposé de la direction du déplacement.
En réalité le prédicat qui lie une coordonnée à cette case de la voiture est supprimé de la base de connaissance (*retract*) et nous ajoutons une nouvelle position (*assert*) de l’autre côté de la voiture.
De cette façon, il n’y a pas besoin de déplacer toutes les cases de la voiture.



Logique du déplacement d’une voiture

L’utilisation de *retract* nous permet de spécifier si nous voulons ajouter le prédicat au début ou à la fin de la base de connaissance à l’aide de *retracta* et *retractz.*

Par exemple si l’on souhaite déplacer la voiture vers la gauche, une case va être ajoutée à sa gauche avec un nouveau prédicat. Ce prédicat va être ajouté au début de la base de connaissance de telle sorte que la case en question devienne le nouveau point de référence de la voiture. Donc on souhaite que cette case soit retrouvée en premier pour cette voiture, pour cela nous utilisons *asserta.*
Le raisonnement est le même pour le déplacement vers le haut.

En revanche, dans les deux autres cas (droite et bas) nous voulons que la nouvelle case soit ajoutée à la fin de la base de connaissance pour laisser la 2eme case de la voiture devenir le point de référence. Dans ce cas c’est *assertz* qui est utilisé.

Restriction des commandes

Nous avons tâché au maximum de contrôler les commandes entrées par le joueur de manière à éviter des comportements indésirables du programme à la suite d’une commande non reconnue.
Par exemple lors de la sélection du niveau la saisie du joueur est lue, si elle correspond effectivement à un niveau existant celui-ci se crée. Mais dans le cas contraire un message d’erreur s’affiche et la règle de sélection de niveau est appliquée à nouveau.

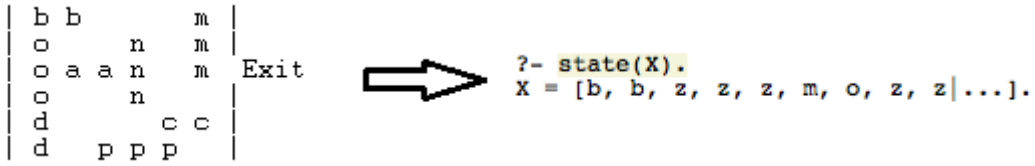
Il en est de même pour le choix de la voiture et de la direction de son déplacement.
De cette façon, l'on reste toujours dans une "boucle" que l'on contrôle.

Résolution automatique

Notre programme ne comporte pas de résolution automatique dans sa version finale. Cependant des solutions ont tout de même été envisagées.

Premièrement, une règle (*get_values*) permet d'obtenir le contenu de la grille de jeu sous forme d'une liste de lettres afin de manipuler plus aisément la configuration du jeu.

Par exemple pour l'état initial du niveau 1 la liste sera :



'z' étant la valeur pour une case vide.

Exploration d'un arbre de solutions

La première option que nous avons voulu essayer était de développer un arbre des solutions possibles à partir d'un état de la grille, et de l'explorer grâce à un algorithme de parcours en profondeur. Il consiste à choisir un des nœuds successeur du nœud initial et de poursuivre un chemin de nœud successeur jusqu'à sa fin. Puis il revient en arrière vers le nœud précédent et explore un autre chemin.
Ce type d'algorithme de parcours s'oppose à la recherche en largeur. Dans ce dernier cas, il s'agit d'analyser tous les successeurs d'un état donné avant de traiter les successeurs de ces successeurs.

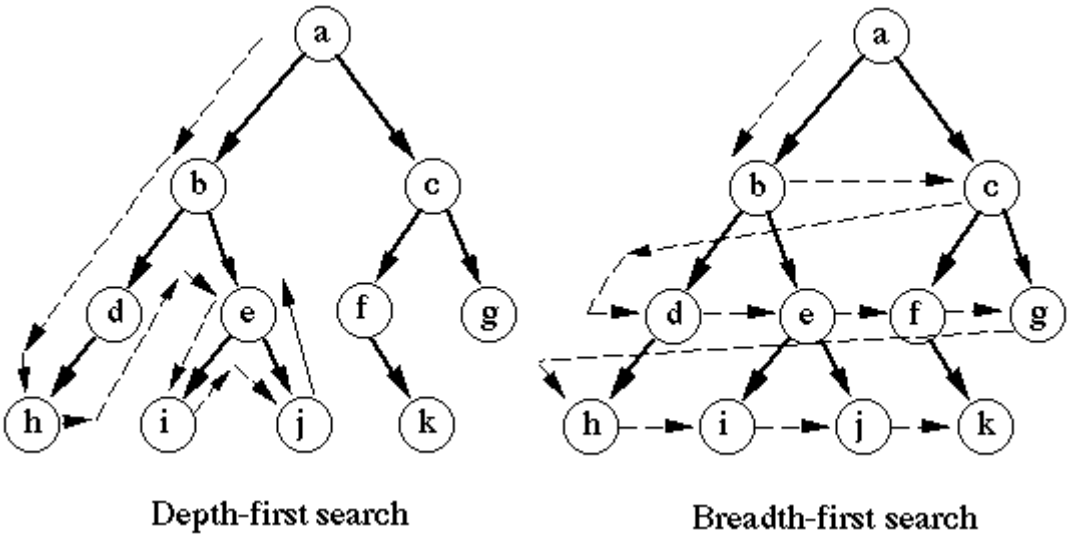


Illustration des 2 algorithme de parcours d'arbre

Un des problèmes à résoudre dans notre cas pour le parcours en profondeur était d'empêcher que le chemin parcouru soit infini. Ce cas de figure pourrait se produire naturellement si un des successeurs d'un nœud est son nœud parent. Concrètement, si une voiture est déplacée vers la gauche, il est possible de la déplacer ensuite vers la droite, ce qui n'a aucun intérêt pour le coup qui vient immédiatement après. La solution aurait donc été de retirer de la liste des successeurs son nœud parent direct. Cependant, le problème se pose aussi sur les 2 précédent coups par exemple. De manière générale, il s'agit d'éviter de traiter une configuration qui a déjà été explorée pour éviter tout chemin infini. La mise en place de cet algorithme avec ce qu'il implique nous a semblé compliquée.

Nous avons alors envisagé le parcours en largeur. Celui-ci nous a paru être la solution optimale car en l'utilisant, tous les nœuds sont traversés et la solution nécessitant le moins de coup est garantie d'être trouvée.
En revanche cet algorithme nécessite de garder en mémoire les différents niveaux de successeurs. Ce problème de stockage n'est pas handicapant avec le parcours en profondeur qui n'implique pas de tout sauvegarder.
Etant donné le grand nombre de successeur pour chaque nœud (pour chaque état de la grille de jeu) les calculs ont finalement semblé trop importants pour utiliser le parcours en largeur. En effet, les grilles de jeu comportent au minimum 8 voitures pouvant chacune se déplacer théoriquement dans 2 directions. Bien que tous les déplacements ne soient pas possibles en pratique, il reste tout de même un nombre élevé de successeur.

Finalement, l'algorithme A* extension de l'algorithme de Dijkstra, nous a semblé être le meilleur compromis. Il permet de trouver le plus court chemin dans un graphe entre un noeud initial donné (ici la grille de départ) et un noeud final également donné (la voiture du joueur est en face de la sortie).
Toutefois, par manque de temps nous n'avons pas implémenté cette solution.

Conclusion

Le paradigme de programmation déclaratif de Prolog nous a dérouté au début du projet tant il diffère du paradigme impératif du C# pour ne citer que lui.
Nous avions certains reflexes hérités de ce dernier qu'il a fallu refréner pour se conformer au fonctionnement de Prolog. Ainsi il a fallu appréhender différemment la façon de penser et de structurer une application. Finalement ce projet nous a fait découvrir une nouvelle façon de programmer qui s'est révélée très puissante pour certaines tâches.

Concernant les pistes d'amélioration de ce programme, nous aurions bien sûr préféré réussir la résolution automatique du jeu qui constitue véritablement la part d'intelligence artificielle de ce projet.
Il aurait également été plus intéressant d'améliorer l'expérience de l'utilisateur. Cela passe, à notre sens, d'une part par une interface plus esthétique en intégrant Prolog à un autre langage qui donne plus de possibilités liées à l'interface, et d'autre part par des interactions plus confortables et intuitives. En effet dans sa version actuelle il est nécessaire d'entrer la lettre d'une voiture puis son déplacement et répéter cette opération même si l'on souhaite déplacer une même voiture sur plusieurs cases. L'idéal serait de pouvoir faire glisser les voitures à la souris.