Planification Automatique et Techniques d'Intelligence Artificielle
Rapport de projet

Amaury Huot Amelina Douard Maxence Ginet Peio Rigaux

Avril 2018

1 Introduction

L'objectif de ce projet était de réaliser un système de planification tournant sur un un robot de type LEGO Mindstorm. Le robot devait suivre les règles du concours "Percycup". Le planificateur a été implémenté à l'aide de la bibliothèque "PDDL4J". L'objectif du robot est de ramener tous les palets présents sur une arène (9 au maximum) dans une zone cible pour marquer des points. Pour cela nous avons séparé notre travail en deux parties indépendantes, une première concernant la réalisation de la planification et une deuxième concernant le pilotage du robot.

2 Planification

Le planificateur PDDL4J fonctionne en utilisant un domaine et un plan. Dans cette partie nous allons expliquer les choix que nous avons fait concernant la modélisation du problème, la création du domaine du et plan ainsi que leur implémentation.

2.1 Modélisation du problème

Comme cité dans notre introduction, le but de notre projet et de faire évoluer le robot sur une arène du concours Persycup.

Cette arène est un espace fermé par des murs et est quadrillée par différentes lignes de différentes couleurs. Ces lignes ont toutes une signification en fonction de leur couleur permettant d'avoir une indication sur la position spatiale du robot : les lignes noires partagent horizontalement et verticalement le terrain en deux, les lignes blanches délimitent les zones cibles de dépôt de palet, et pour finir les lignes des autres couleurs sont toutes uniques et séparent en deux chacun des demis terrains ou elle sont. Toutes ces lignes ne se croisent entre elles qu'une seule fois. Selon le règlement du concours , les neuf palets sont placés en début d'épreuve sur les croisements de lignes (sauf pour les lignes blanches).

En prenant notes des points précédemment cités nous avons décidé de modéliser le terrain sous la forme d'un graphe à douze sommets dans lequel chaque sommet représente l'intersection de deux lignes de couleur et un arc entre deux sommet signifie que sur le terrain un segment d'une couleur unique relie donc ces intersections.

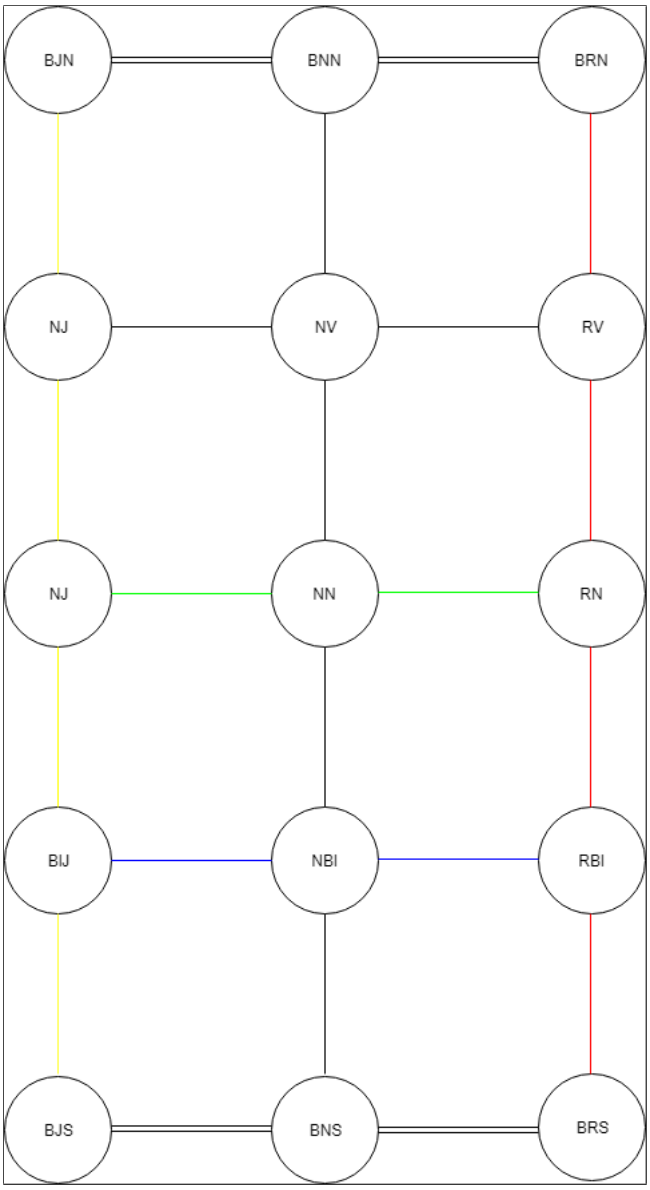

Figure 1: Schéma du graphe modélisant le terrain

2.2 Le domaine et le plan

Une fois que notre problème fut modélisé nous avons pu travailler sur le domaine utilisé par le planificateur pour résoudre le problème. En premier lieu, nous avons déterminé avec certitude que trois actions était primordiales, il s'agit des actions **move**, **saisir_palet** et **deposer_palet**.
Afin de pouvoir réaliser ces actions nous avons dû créer deux types : node, représentant les noeuds du graphe et claw représentant la pince du robot. Nous avons dû aussi créer une série de prédicats :

(at ?x)	:	le robot est proche du croisement représenté par le noeud **x**
(link ?from ?to - node)	:	il existe un arc entre le noeud **from** et le noeud **to**
(opened ?c - claw)	:	la pince du robot est en position ouverte
(closed ?c - claw)	:	la pince du robot est en position fermée
(free ?c - claw)	:	le robot peut attraper un palet, il n'est pas en train d'en transporter un
(paletin ?c - claw)	:	le robot transport un palet, sa pince est occupée
(paletat ?loc - node)	:	il y a un palet au croisement des lignes représenté par le noeud **loc**

move : Cette action prend en paramètre deux noeuds **from** et **to**, le premier est le noeud où doit se trouver le robot au moment de l'action, le deuxième est le noeud où doit se rendre le robot. Pour pouvoir être exécutée, cette action doit remplir le prédicat (link ?from ?to).

Elle consomme le prédicat (at ?from) et en génère un (at ?to).

saisir_palet : Cette action prend en paramètres la pince **c** du robot, et le noeud **loc** où se trouve le palet et va réaliser la saisie du palet. Pour cela le robot doit avoir la pince ouverte et ne pas transporter de palet. De plus il doit se trouver proche de l'intersection symbolisée par **loc**. Cette action consomme les trois prédicats (free ?c), (opened ?c), (paletat ?loc) et va en générer deux (closed ?c) et (paletin ?c).

deposer_palet : Cette action prend en paramètres la pince **c** du robot, et le noeud **loc** où va être déposé palet. Pour cela le robot doit avoir sa pince **c** fermée , un palet à l'intérieur et se trouver près de l'intersection correspondant à **loc**. Cette action consomme les deux prédicats (closed ?c), (paletin ?c) et va en générer trois (free ?c), (opened ?c) et (paletat ?loc).

Avec les trois actions citées ci-dessus, le planificateur peut exécuter sans soucis les enchaînements d'actions permettant la saisie et le dépôt en zone cible des palets si le robot commence avec la pince en position ouverte. Nous avons cependant ajouté deux autres actions **ouvrir_pince** et **fermee_pince** afin de rendre les situations de calcul d'une solution et de sa mise en exécution plus flexibles. En effet, grâce a la présence de ses actions la mise en exécution d'une solution peut être interrompue pour diverses raisons (non présence d'un palet , désynchronisation de la position du robot, etc.) et on peut en recalculer une indépendamment de l'état d'ouverture de la pince ou du transport d'un palet.

ouvrir_pince : Cette action prend en paramètre la pince **c** du robot. Pour pouvoir être réalisée, elle doit être dans l'état fermée. Cette action consomme le prédicat (closed ?c) et génère le prédicat (opened ?c).

fermee_pince : Cette action prend en paramètre la pince **c** du robot. Pour pouvoir être réalisée, elle doit être dans l'état ouverte. Cette action consomme le prédicat (opened ?c) et génère le prédicat (closed ?c).

Maintenant nous allons décrire l'écriture du plan, deuxième partie nécessaire au planificateur pour calculer une solution.
Les points importants sont :

- La déclaration des objets modélisant le terrain, il faut un noeud pour chaque intersection (BJN BNN BRN VJ NV RV NJ NN RN BLJ BLN BLR BJS BNS BRS - node) et la pince du robot (pince - claw).
- L'initialisation, elle consiste en la déclaration de tout les arcs entre les noeuds par les prédicats *link*, la déclaration de la position du robot avec le prédicat *at*, l'état de pince avec le prédicat *closed* ou *opened*, l'occupation de la pince avec le prédicat *free* ou *paletin* et pour finir la déclaration de la position des palets cibles avec un prédicat *paletat* pour chaque palets.
- L'objectif est de constituer une conjonction de prédicats *paletat* donnant la position des palets ciblés après le calcul de la solution.

2.3 Implémentation en Java

La partie concernant la planification se trouve dans le package Planification, il contient les deux classes Ourplanner et Node qui forment une version de notre planificateur inspirée de celui appelé HSP fourni dans les sources de PDDL4J, allégé dans les sorties graphiques, et une classe Main. La classe Main est celle qui permet de créer l'archive exécutable. Cette classe va instancier le planificateur et ensuite charger un domaine et un plan pour générer un problème qu'elle soumettra au planificateur. Nous récupérons ensuite la suite d'actions de la solution dans une String que nous épurons de toute information inutile pour le robot. La String épurée est ensuite envoyée au robot via les sockets.

Nous avons choisi d'implémenter notre planification en dehors de la brique de commande du robot pour plusieurs raisons ; la première étant un impératif de version de Java, vu que la version 3.5 de PDD4J nécessite la version 8 de Java, qui ne peut tourner sur la brique programmable du robot. Le deuxième avantage qu'apporte le fait d'avoir le planificateur dé-localisé sur une machine externe est de réduire la consommation en énergie du programme, point important étant donné que les performances du robot sont directement impactées par le niveau de la batterie. Le dernier avantage à avoir le planificateur en dehors du robot est la vitesse de calcul des solutions de planification.

En effet la puissance de calcul de la brique programmable étant limitée, cela rallongerait grandement le temps de calcul si le planificateur se trouvait être exécuté sur la brique, contrairement à un fonctionnement sur PC (temps de calcul avoisinant les 0.20 secondes sur un PC). Par contre, faire tourner le planificateur en externe nécessite de communiquer avec le robot via des sockets.

3 Pilotage du robot

3.1 Configuration du robot

Chaque robot utilisé par un groupe a ses caractéristiques propres. Les moteurs sont branchés sur certains ports précis, et il en va de même pour les capteurs. De plus, des variables correspondant à la vitesse et à l'accélération (angulaires et linéaires) sont nécessaires pour le déplacement. Tout cela est géré dans le fichier Config.java. Ces valeurs servent notamment lors de l'instanciation des capteurs et des moteurs.

3.2 Gestion des couleurs

Notre robot se repère sur la table à l'aide de son capteur de couleurs. Il reconnaît les différentes lignes présentes une fois son capteur calibré.

3.2.1 Calibration

La calibration des couleurs est nécessaire pour que le robot puisse reconnaître une ligne de couleurs pour une luminosité donnée. Un échantillon contenant les valeurs Rouge, Bleu, et Vert est récupéré puis associé à chaque couleur.

Nous avons prévu une calibration alternative qui permet de faire la moyenne de trois échantillons au lieu d'un seul. Cela permet une meilleure précision.

Afin d'éviter de recalibrer à chaque lancement d'un programme, un fichier de calibration est créé. Il est lu à l'aide de la méthode setCalibration().

3.2.2 Méthodes importantes

La classe Couleur contient des prédicats permettant d'identifier chaque couleur (tels que isBlue(), par exemple) ainsi que la méthode getCurrentColor() qui permet de savoir de quelle couleur identifiée se rapproche le plus l'échantillon.

3.3 Gestion des déplacements

La classe DifferentialDrive contient nos méthodes de déplacement. Lors de l'instanciation, les moteurs et le MovePilot (pilote virtuel permettant d'interagir avec les moteurs) sont initialisés.

Pour récupérer les palets, il faut détecter leur présence. La méthode GoUntilTouch(TouchSensor ts, boolean grab) permet au robot d'avancer jusqu'à ce que le détecteur de pression soit déclenché. Elle fut utilisée dans les tests de déclenchement du capteur en fonction de la vitesse du robot.

Pour la détection des couleurs, nous utilisons followLine(Couleur c, TouchSensor ts) et recoverLine(Couleur c, float[] couleur). recoverLine() est appelée par followLine si le robot se pert. followLine() demande au robot d'avancer d'un pas défini à l'avance si il est toujours sur la ligne de couleur. Si il n'est plus dessus, il va tourner pour retrouver la ligne tant que la variable attestant que le robot est perdu n'est pas atteinte.

Des squelettes de méthodes sensées correspondre aux ordres du planificateur ont été créés mais non testés...

3.4 Problèmes rencontrés

La réalisation de la partie pilotage ne s'est pas du tout passée comme prévu. Premièrement, nous n'avions pas récupéré les exemples testés du professeur. De plus, nous avons surtout utilisé le code "Patia2016". Ce code nous a causé bien des problèmes... Des problèmes de compatibilité pour commencer, car il utilisait notamment DifferentialPilot pour le déplacement, alors que cette classe est dépréciée pour notre version de LeJos. Nous avons donc repris toute la documentation afin de comprendre comment implémenter MovePilot, la nouvelle classe recommandée par cette dernière.

Au cours de l'implémentation de cette classe, nous nous sommes retrouvés à avoir des problèmes de compréhension par rapport au fonctionnement de certaines méthodes de déplacement (ce qui fait que nous avons perdu encore plus de temps). La méthode forward() a particulièrement attiré notre attention car nous n'avions pas compris qu'il fallait l'appeler avant une boucle déterminant quand elle termine... C'est pourquoi le déplacement se finissait avant même que le robot ne bouge.
Nous avions donc utilisé travel() à la place, sans réussir à ce que le robot se déplace de façon fluide. De la même façon, nous avons essayé de tester des accélérations et vitesses différentes, qui faisaient grandement varier la précision du déplacement.

Cette limitation par rapport aux déplacements du robot a eu une incidence sur le non-déclenchement du détecteur de pression, ce qui a posé un gros problème ensuite pour la détection des palets.

À la mi-projet, nous nous sommes rendus compte qu'il nous manquait certains codes des anciens étudiants, dont nous avions besoin pour comprendre la gestion des capteur et l'intégration du planificateur. Nous les avons donc récupérés, et nous avons essayé de faire fonctionner les capteurs (nous avions perdu tellement de temps précédemment, que l'utilisation et les tests de tous les capteurs dans le temps restant étaient tout bonnement impossible). Le capteur de couleur a retenu notre attention car il est plutôt simple à implémenter. Nous avons réussi à faire suivre une ligne de couleur au robot.

4 Les limites de notre réalisation

Nous avions prévu que notre robot récupère les palets à l'aide des échanges répétés entre le planificateur et le robot afin de planifier la saisie des palets en plusieurs étapes. Malheureusement, suite à nos retards dans la réalisation du pilotage du robot nous n'avons pu réaliser et mettre en oeuvre cette partie. De plus, nous n'avons pu récupérer que un palet, car nous n'avons pas pu tester la récupération de tous les palets à l'aide des lignes de couleur.

5 Conclusion

Ce projet nous a fait découvrir l'utilisation d'un planificateur (PDDL4J) appliquée à la robotique. L'utilisation de robots Mindstorm simples nous a permis de nous rendre compte des contraintes propres à ce genre de robots (la difficulté d'avoir un déplacement précis, les divers ajustements etc...). Le planificateur permet d'optimiser le temps de résolution du problème posé en fonction de la position actuelle du robot. Le planificateur ici n'est pas utilisé au maximum de son potentiel. Cependant, la gestion d'évènements non prévus est un véritable avantage par rapport à l'approche par automates. Nous avons pu remarquer l'importance de la cohésion et de la communication du groupe ainsi que l'intérêt de remettre en question les choix de départ, afin de en pas être mis en difficulté par des idées non viables.