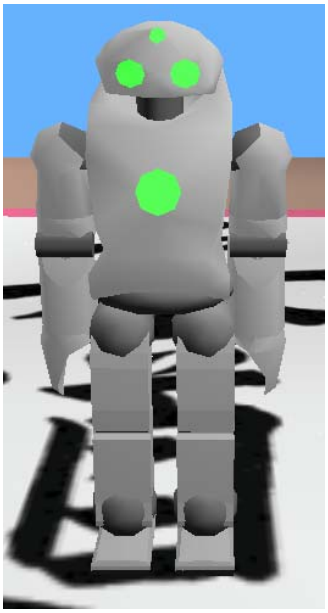
UFR de Sciences,
Section Informatique

Maîtrise Informatique

UE6 : Intelligence artificielle distribuée

Programmation d'un robot judoka



Julien Van Den Bossche / Benoît Moulin
info@julienvdb.com / bmoulin@etu.info.unicaen.fr

1. Introduction

Le but de ce devoir est de développer une architecture logicielle agent pour contrôler un robot Judoka. Nous allons vous montrer comment nous avons réussit à développer cette architecture en nous inspirant de l'architecture de subsomption.
Vous pouvez avoir des informations, sur notre code Java, sur nos méthodes par le biais de la commande *javadoc*. L'API de notre contrôleur est disponible sur http://www.julienvdb.com/universite/maitrise/IA/doc
Plusieurs types de marches ont été développés et sont visibles sur http://www.julienvdb.com/universite/maitrise/IA/videos
Le code source est disponible sur http://www.julienvdb.com/universite/maitrise/IA/sources

2. Présentation du scénario

2.1. Description générale

Nous avons, sur une aire de combat, deux robots judoka. Initialement ces deux robots se trouvent l'un en face de l'autre, sur un bord opposé du tatami. Chacun se trouve au milieu du bord de ce dernier.
Un robot reste immobile.
Un autre doit marcher pour se placer derrière le robot immobile.
Ce robot mobile doit pousser le robot immobile et ainsi le faire tomber par terre.
Le robot mobile possède un capteur qui permet de repérer un obstacle.

2.2. Comportement du judoka mobile

Ce robot doit pouvoir effectuer plusieurs actions :

Il doit être capable d'actionner ces servomoteurs pour pouvoir faire un pas et ce sans tomber.
Il peut donc marcher.
Il doit pouvoir tourner dans une direction souhaitée puis reprendre sa marche tout droit.
Il doit éviter les obstacles, tel que les rebords de l'aire de combat.
Il doit pouvoir contourner le robot immobile pour se placer derrière
Il doit pouvoir actionner ses bras pour pousser l'autre robot sans tomber.
Il doit pouvoir se relever en cas de chute.

3. Mise en place de l'architecture de subsomption

3.1. Principe générale de l'architecture

L'architecture de subsomption (également connue sous le nom de « contrôle du comportement ») est l'un des modèles de programmation le plus répandu pour résoudre des besoins de programmation au niveau élémentaire, avec peu de mémoire. Cette approche du contrôle du comportement a été élaborée par Rodney Brooks dans le laboratoire d'intelligence artificielle du MIT.

3.2. Programmation d'une architecture de subsomption

Un comportement s'appuie sur deux éléments principaux : des capteurs (entrées, informations) et des activateurs (sorties, actions). Le couple formé d'une condition et d'une action se nomme un comportement.

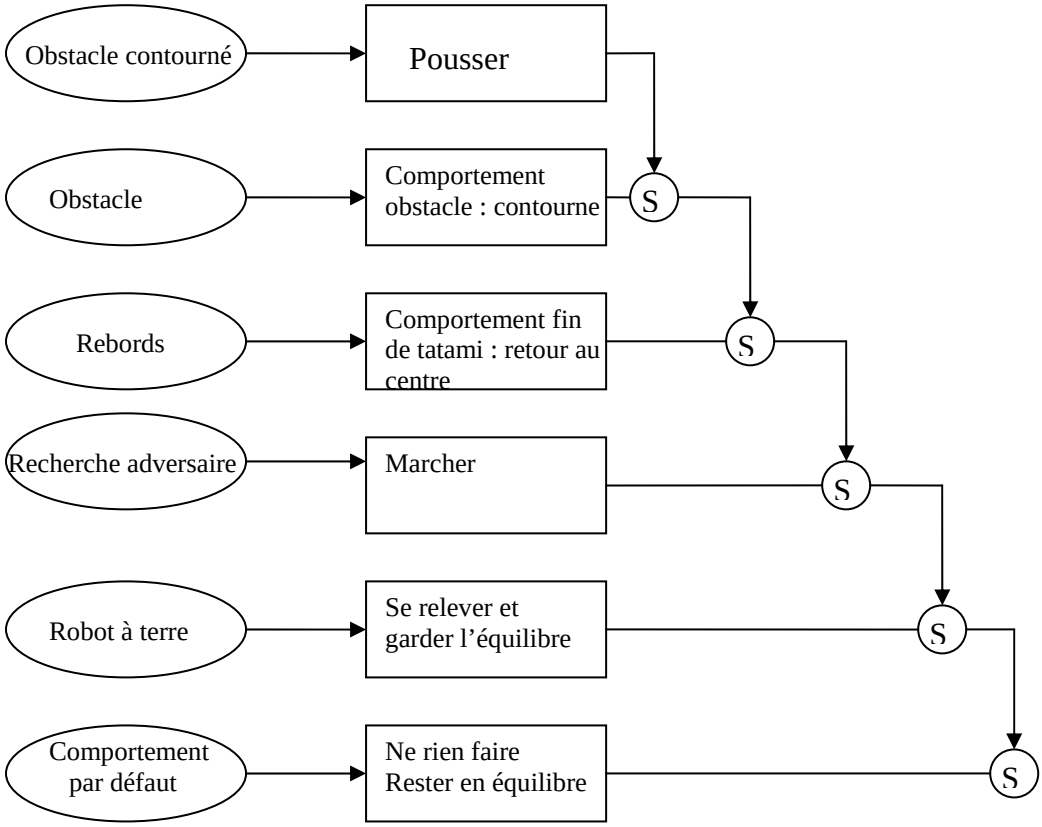
Comportement : détecte l'autre	
Condition :	**Action :**
capteur à repérer l'autre robot	le contourner pour le pousser

L'action d'un comportement peut être simple, comme faire un pas, ou plus complexe comme retourner au centre du tatami. Plus clairement, un comportement est un programme qu'exécute le robot pendant un certain temps. Un comportement s'engage lorsque la condition associée devient vraie ; le déclenchement ne dépend pas directement d'un capteur externe. Un robot peut donc aussi réagir à la survenue d'une condition interne, la position d'un servomoteur par exemple. Toute condition, qu'elle soit externe ou interne peut alors activer un comportement.

Dans certaines situations, plusieurs comportements entrent en compétition. Pour que le robot sache se décider, les comportements doivent être associés à des niveaux de priorité. Par exemple, un robot ne sert plus à rien s'il fait un pas et qu'il est sur le point de tomber. La marche est donc pour lui un comportement à faible priorité, car dès qu'il se sent déséquilibrer il doit rechercher son équilibre. Il faut alors que le programme en cours puisse interrompre l'action en cours pour démarrer l'exécution d'une action plus prioritaire. Cela revient à supprimer un comportement moins prioritaire.

Rodney Brooks a élaboré un diagramme standard pour représenter la hiérarchie des comportements. Il présente les différents comportements et leurs priorités relatives. Un comportement qui en subsume d'autres est repéré par la lettre S. La règle principale de l'architecture de subsomption est que tous les comportements de priorité inférieurs sont supprimés dès qu'un comportement de priorité supérieure doit s'engager. Un seul comportement peut prendre place à chaque instant.

3.3. Notre architecture pour le robot mobile



Notre stratégie est la suivante :

Dans cette partie et sur la figure ci-dessus nous décrivons l'architecture de subsomption pour le robot dans son « aire de jeu ». Nous n'intégrons pas l'architecture de la marche qui sera décrite plus bas.

Dans le devoir, on nous demande d'aller tout droit depuis la position d'origine puis repérer le robot qui est en face et le faire tomber en se mettant derrière.
Notre stratégie de base sera donc celle là mais il se peut que le robot à pousser ne soit pas exactement en face. Donc ce cas sera aussi pris en compte dans notre stratégie.

La base de notre stratégie est de garder l'équilibre, donc si le robot est tombé on doit le faire relever.

Recherche de l'équilibre
Dans tous les niveaux au dessus on va toujours regarder si le robot est debout ou bien s'il tombe. On utilise le gps pour visualiser sa coordonnée sur l'axe OY. Si la valeur est inférieure à 0.5 alors cela signifie que le robot est tombé.
La méthode releve() sera appelée, puis on positionnera le robot au centre pour qu'il recherche son adversaire.

Recherche de l'adversaire

- Cas n°1 :

Si le robot est debout alors on le fait marcher tout droit, jusqu'à ce qu'il trouve son adversaire. Il passe à la stratégie d'attaque.

- Cas n°2 :

Il na pas trouvé son adversaire en allant tout droit, donc on le fait revenir au centre du tatami. Une fois le centre atteint, il va tourner sur 360° et regarder où l'adversaire se trouve.
Il prendra alors sa direction. Une fois le robot trouver il passe à la stratégie d'attaque.

Attaque

Pour l'attaque nous ne nous mettons pas derrière l'adversaire de façon parallèle sinon quand on tend les bras ces derniers se logent entre les épaules de l'adversaire sans le faire tomber. On le pousse donc de biais.

- Cas n°1 :

S'il arrive sur l'adversaire et qu'il est proche du bord du tatami alors il poussera l'adversaire et ne le contournera pas car sinon il sortirait du tatami.

- Cas n°2 :

S'il arrive sur son adversaire et arrive à s'arrêter à une distance de 150 alors il le contourne pour le pousser dans le dos.

- Cas n°3 :

S'il se situe tout proche du robot adversaire (distance<150) alors il le poussera car quand on le contourne on risque de tomber du fait que les épaules des deux robots vont se toucher.

En fait nous avons fait deux méthodes *contourne()* car quand on recherche l'autre robot, on n'arrive pas forcément de face par rapport à lui et une fois l'adversaire contourné on a du mal à le faire tomber selon comment notre robot se place (on peut arriver de biais dessus).
Donc on a mis une méthode *contourne()* en commentaire dans notre code qui est utile si l'on veut faire un combat avec un autre adversaire. Il faudra mettre la première méthode en commentaire. La méthode qu'il y a dans le fichier *Judoka0.java* et qui n'est pas en commentaire sert pour la situation où l'on part tout droit vers l'autre robot (sujet de ce devoir).

Pour coder cette stratégie avec JAVA nous avons définit une méthode strategie(int couche). Quand on commence le round, notre robot est debout donc on va aller à la couche 1 qui va nous permettre de marcher. Si dans la couche 1 nous captons un événement indiquant que l'on est proche du robot (capteur de distance < 150) alors on va passer à la couche 2 (appel de strategie(2)) qui va permettre de passer à l'étape de contournement du robot.
On procède avec ce principe de couche en couche.
Cela nous évite, par exemple, de chercher l'adversaire si notre robot est par terre.

Code de la méthode :

```
public static void strategie(int couche){
        if(couche==0){
            releve();
            robot_step(2000);
            init();
            robot_step(1500);
            strategie(2);
        }
        if(couche==1){
            for(;;){
                    marche();
                    robot_step(150);
                    matrix=gps_get_matrix(gps);
                    System.out.println("positionZ*="+gps_position_z(matrix));
                    System.out.println("distance="+distance_sensor_get_value(distance));
                    if(estTombe()){
                        strategie(0);
                        break;
                    }
                    //retour au milieu du tatami
                    if((gps_position_z(matrix)<-
0.8)||(gps_position_z(matrix)>0.8)||(gps_position_x(matrix)<-0.8)||(gps_position_x(matrix)>0.8)){
                        System.out.println("au centre");
                        robot_step(200);
                        init();
                        robot_step(200);
                        strategie(2);
                        break;
                    }
                    //si proche de l'adversaire le faire tomber
                    if(distance_sensor_get_value(distance)<200){
                        strategie(3);
                        break;
                    }
            }
        }
        if(couche==2){
           goToCenter();
        }
        if(couche==3){
            robot_step(200);
            init();
            robot_step(200);
            contourne();
        }
        if(couche==4){
            init();
        }
    }
```

4. Le déplacement du robot

4.1. Le robot marche

Pour que le robot marche correctement, il faut une marche qui soit à la fois stable, rapide et la plus droite possible. Pour atteindre ce but, nous avons donc réalisé plusieurs marches pour arriver à la plus intéressante.

4.1.1. La marche stable

Au début de ce projet, nous voulions trouver une marche stable pour pouvoir travailler le plus vite possible sur les problèmes de repérage de l'autre robot. Nous avons donc créé une marche où le robot traîne des pieds et où il écarte un peu les jambes pour permettre une plus grande stabilité. De plus, cette démarche permet au robot d'aller tout droit, sans dévier de la trajectoire voulue. Par contre, comme le robot traîne des pieds, cette marche est assez lente, ce qui fait qu'il met beaucoup de temps à atteindre son but (l'autre robot).

4.1.2. La marche réaliste

Ensuite, nous avons décidé de créer une marche la plus réaliste possible. En effet, avec cette démarche, le robot lève les pieds et fait des pas beaucoup plus grands qu'avec la marche précédente.
La principale difficulté pour réaliser cette marche est de passer d'un pied à l'autre tout en respectant l'équilibre.

Tout d'abord, le robot lève la jambe droite et doit se pencher vers la gauche pour garder son équilibre et se mettre sur le pied gauche (figure 1). Ensuite, il se penche vers l'avant et vers la droite pour se mettre en équilibre sur la jambe droite (figure 2), c'est ce qu'on appelle le transfert de masse. Après, il se remet sur son pied gauche par le même procédé (figure 3), pour revenir en position initiale (figure 4) :

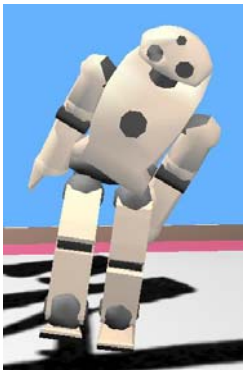


figure 1


figure 2

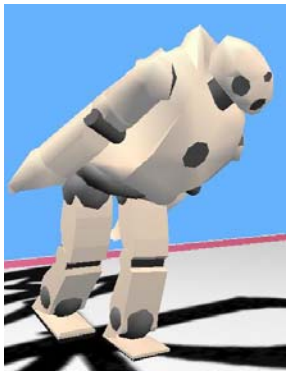


figure 3


figure 4

Cette marche réaliste est donc une répétition de cet enchaînement de mouvements. Mais malheureusement, nous ne l'avons pas utilisée dans la suite du projet car elle n'est pas rapide et le robot ne marche pas droit. En effet, il existe un léger décalage difficile à corriger étant donné que le logiciel Webots n'est pas fiable à 100 % et ne fonctionne pas toujours de la même manière d'un ordinateur à l'autre.

Nous avons donc décidé de créer une troisième marche qui est un peu moins réaliste, mais qui permet au robot d'avancer vite et droit.

4.1.3. La marche utilisée

Contrairement à la démarche précédente qui utilise quasiment la totalité des moteurs du robot (jambes et genoux pour avancer ; dos, chevilles et hanches pour l'équilibre…), la marche que nous avons utilisée ne se sert seulement que des jambes et des genoux. Pour cette marche, nous nous sommes moins souciés de l'équilibre. Nous avons accéléré l'enchaînement et donc le robot « n'a pas le temps » de tomber. Pour le faire avancer, nous l'avons juste penché un peu vers l'avant.

4.2. Faire tourner le robot

Pour faire tourner le robot, nous avons utilisé une technique qui lui permet de garder la même position, seule son orientation change. Comme le montre la figure 5, le robot tourne d'abord sa jambe pour que ses pieds soient perpendiculaires l'un à l'autre. Ensuite, il se penche du côté de la jambe tournée (figure 6) et se remet en position initiale (tous les moteurs sont à 0) (figure 7).

figure 5

figure 6

figure 7

Il est possible de régler la vitesse à laquelle le robot tourne ainsi que l'angle de rotation du robot grâce à la vélocité des moteurs.

4.3. Le robot se relève

Il peut arriver que le robot tombe. Nous avons donc décidé de créer une méthode permettant de le relever. Pour cela, nous avons réfléchit à plusieurs techniques. Pour que le robot se relève le plus facilement et le plus rapidement possible, il faut qu'il se trouve sur le ventre. Donc si le robot tombe sur le ventre, il se relève, si il tombe sur le dos, il se retourne avant de se relever. Les figures 8, 9, 10 et 11 montrent la phase de retournement :


figure 8

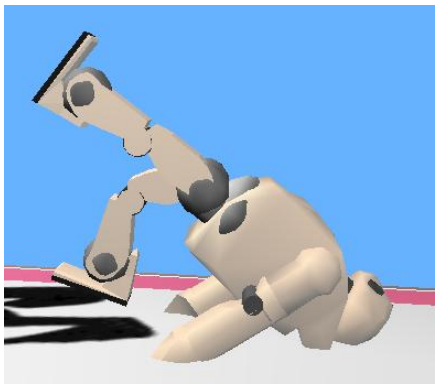


figure 9

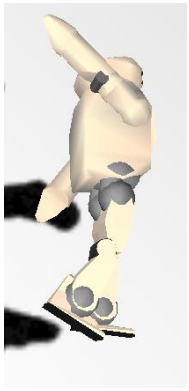


figure 10


figure 11

Pour relever le robot, nous commençons par le mettre sur les genoux (figure 12). Ensuite, à l'aide de ses bras et de sa tête, nous passons ses jambes sous son ventre (figure 13). Les points d'appui du robot sont alors la tête et les pointes des pieds (figure 14). Puis, une rotation des bras permet au robot de se retrouver sur ses pieds (figure 15). Enfin, il ne reste plus qu'à le déplier et à le remettre en position initiale (figure 16 à 20):

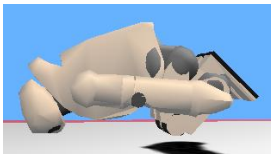


figure 12

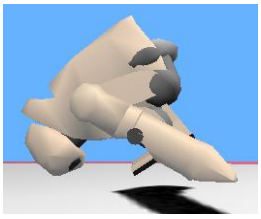


figure 13

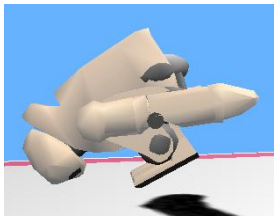


figure 14

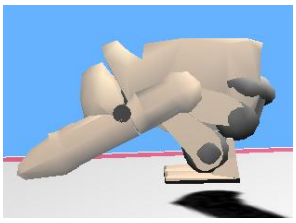


figure 15

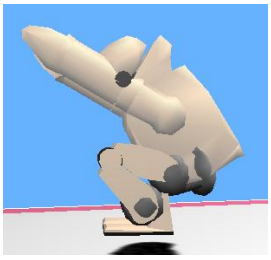


figure 16

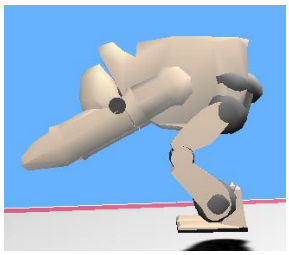


figure 17


figure 18


figure 19


figure 20

5. La navigation

Le robot se déplaçant sur le tatami doit être apte à ne pas en sortir. Il doit aussi pouvoir regarder si un autre robot est à proximité de lui.
Le tatami possède des coordonnées variant de 1 à -1

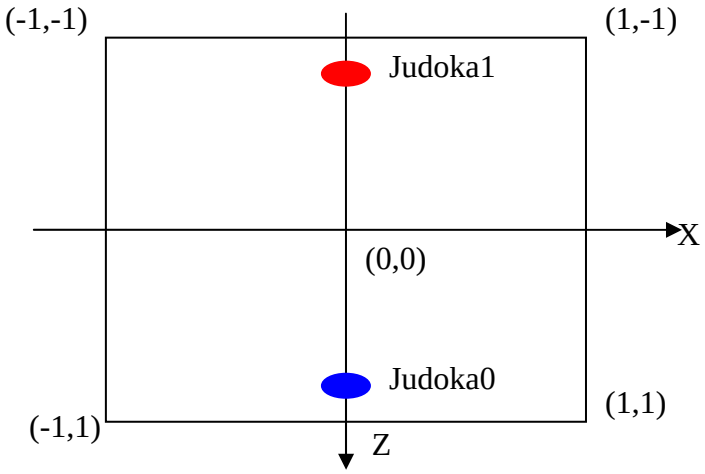


Fig.3. Représentation du tatami.

5.1. Rester sur le tatami

Nous avons décidé de faire tourner le robot quand ses coordonnées en X ou Z sont supérieures à 0.8 ou inférieure à -0.8.
Dès que le robot est sur le bord, cela signifie qu'il n'a pas trouvé de robot sur son chemin, on le fait aller au centre du tatami pour qu'il reparte à la recherche de son adversaire.
Pour repartir au centre du tatami, nous calculons par rapport à sa position, l'angle exprimé en radians, qui définit la route qu'il doit prendre pour aller au centre.
Ensuite nous faisons tourner le robot jusqu'à ce que sa direction soit la même que la route voulue.
Ensuite nous le faisons marcher jusqu'au centre.

5.2. Le repérage de l'autre robot

Le repérage de l'autre robot se fait à l'aide du capteur de distance.
Dans le cas où l'on est revenu au centre, c'est-à-dire que notre première stratégie d'aller tout droit n'a pas fonctionné, le robot tourne et analyse son capteur de distance. S'il indique une distance inférieure à 450 alors on s'arrête de tourner et on va dans la direction de l'adversaire.
Pour contourner l'autre robot, nous attendons que la valeur du capteur de distance soit inférieure à 150.

6. Difficultés rencontrées

Nous avons rencontrés plusieurs problèmes lors du développement de notre robot.
Nous avons était obligé de faire plusieurs marches car la première était très lente, la deuxième ne fonctionnait pas de la même manière selon les systèmes d'exploitations et les machines. Le robot perdait l'équilibre sur certaines machines et sur d'autres non.
La troisième marche, que l'on a adoptée, est plus rapide mais cependant quelques problèmes surviennent avec le simulateur :
Une fois que l'on a tourné pour revenir au centre le robot peut dévier de sa trajectoire selon la machine sur laquelle on fait tourner Webots.
De part, tout ces aspects on a eu du mal à respecter la précision que l'on souhaiterait avoir sur les angles (direction du robot), déplacements…

Le simulateur a quelques tendances à s'arrêter et faire des erreurs systèmes ce qui a ralentit le développement de notre robot. Nous sommes souvent obligé de redémarré notre simulation et redémarrer notre ordinateur pour que Webots fonctionne correctement.

7. Conclusion

7.1. Notre point de vue sur le projet

Nous avons pu découvrir un domaine que nous ne connaissions pas et qui nous a semblé fort intéressant. Nous avons surtout pu mettre en pratique ce que l'on a appris en cours et td et nous avons constaté qu'il était important de définir une bonne architecture avant de se lancer dans le code.

7.2. Améliorations possibles

Si on avait eu plus de temps et moins d'ennuis avec le simulateur nous aurions essayé de faire avancer le robot plus vite. Nous aurions essayé de le faire scruter le tatami en marchant au lieu de revenir au centre comme nous le faisons pour rechercher son adversaire.
On aurait essayer de reconnaître l'adversaire, en analysant les images acquises par la caméra du robot.