Université Grenoble Alpes

Plan de tests

Projet Intelligence Artificielle

Eva DROSZEWSKI
Sara IKAN
Anesie MARTINIANI
Alix PRATABUY

I- Introduction

Ce document a pour but de décrire tous les tests effectués sur le robot afin de s'assurer de son bon fonctionnement, mais également de trouver des constantes nécessaires au codage de certaines méthodes. Enfin il est présenté les critères d'acceptation de chaque test et s'ils ont été respectés.

II- Tests unitaires

1. Tests unitaires pour la classe Capteurs

Test pour la méthode **getDistance()** :

- Description du test : le robot est placé à une distance mesurée à la règle d'un mur. La distance mesurée par l'UltraSonicSensor est ensuite affiché pendant dix secondes pour vérifier que la distance mesurée est la bonne. Une fois cette vérification effectuée nous avons fait la même chose en intercalant un palet entre le robot et le mur pour voir jusqu'à quelle distance celui-ci peut le détecter.
- Contraintes : il n'y a pas de contraintes pour ce test.
- Dépendances : il n'y avait pas de dépendances car le robot était déplacé à la main.
- Procédure de test : nous attendions une mesure renvoyée du capteur identique à celle mesurée au centimètre près. Lorsque nous avons effectué le test uniquement sur le mur , le critère de validation était respecté. Cependant, lorsque nous avons rajouté le palet et déplacé celui-ci à une distance d'environ 30 cm, le capteur ne nous donnait pas la mesure attendue. Il semblait que celle-ci soit une valeur défaut car elle était identique à chaque tentative (même à 10^{-4}).

Test pour la méthode **getPression()** :

- Description du test : le robot a les pinces ouvertes sans palet à l'intérieur. La mesure de la pression est effectuée et affichée pendant 10 secondes. Ensuite un palet est ajouté dans les pinces du robot, en vérifiant bien que celui-ci appuie sur la capteur de pression. De même, la pression est mesurée et affichée pendant 10 secondes.
- Contraintes : il n'y a pas de contraintes pour ce test.
- Dépendances : il n'y avait pas de dépendances car le palet est introduit à la main et placé au bon endroit à la main également.
- Procédure de test : nous attendions une pression nulle lorsque aucun palet n'était présent et une pression supérieure à zéro quand le palet était présent. Ce qui fut le cas lors de la réalisation du test.

Test pour la méthode **getCouleur()** :

- Description du test : avant de réaliser le test nous avons calibré le capteur de couleur avec le code fournit sur le page internet du projet. Ensuite nous avons placé le robot sur une ligne, blanche, rouge, jaune, verte et noire afin tout en affichant à chaque fois la couleur obtenue pendant 10 secondes.
- Contraintes : il n'y a pas de contraintes pour ce test.
- Dépendances : il n'y avait pas de dépendances car le palet est introduit à la main et placé au bon endroit à la main également.

- Procédure de test : nous attendions que la couleur soit identique à celle sur laquelle était positionnée le robot. Lors du test cela ne fut pas le cas, les couleurs verte et blanche étaient considérées comme identiques. Nous avons alors changé de capteur, aucune amélioration n'a été observée. Nous avons donc abandonné l'utilisation de ce capteur.

2. Tests unitaires pour la classe Actions

Test pour la méthode **fermer_pinces() et ouvrir_pinces()** :

- Description du test : nous avons ici testé l'angle pour lequel il faut ouvrir les pinces mais également la vitesse du moteur.
- Contraintes : les pinces ont une fermeture maximales et si celle-ci est dépassée parce que l'angle de fermeture est trop grand par rapport à l'angle d'ouverture cela désaligne les pinces
- Dépendances : pour faire les différents tests il faut pouvoir ouvrir les pinces et fermer les pinces (interdépendance entre les deux méthodes).
- Procédure de test : ici on s'attend à ce que les pinces s'ouvrent du bon angle et que celui-ci ne soit pas trop important pour ne pas perdre du temps mais en même temps assez grand pour récupérer le palet correctement.

Test pour la méthode **tourner_deA()** :

- Description du test : le robot est positionné face à un mur et on lui demande de tourner de 360° ou 90°.
- Contraintes : pas de contraintes.
- Dépendances : pas de dépendances.
- Procédure de test : il faut que le robot tourne de l'angle effectivement entré en paramètre dans la méthode (qu'il soit négatif ou positif).

Test pour la méthode **avancer_deA()** :

- Description du test : le robot est positionné à côté d'une règle et on lui demande d'avancer d'une certaine distance.
- Contraintes : pas de contraintes.
- Dépendances : pas de dépendances.
- Procédure de test : il faut que le robot avance effectivement de la distance mise en paramètre dans la méthode.

Test pour la méthode **reculer_deA()** : même test que pour la méthode **avancer_deA()**.

Test pour la méthode **getisMoving()** :

- Description du test : on demande au robot de bouger (tourner, avancer ou reculer) et on affiche un message indiquant qu'il bouge tant que la méthode **getisMoving()** renvoie true, le message s'affiche à un intervalle de temps régulier prédéfini.
- Contraintes : pas de contraintes.
- Dépendances : il faut que les différentes méthodes de déplacement fonctionnent (**avancer_deA()**, **reculer_deA()** et **tourner_deA()**).
- Procédure de test : il faut que le robot affiche le message indiquant quand le robot bouge et que le message cesse lorsque le robot cesse de bouger.

III- Tests d'intégration

3. Tests d'intégration pour la classe Capteurs

Test de la méthode **get_angle_trigo()** :

- Description du test : on fait tourner le robot de 90° et l'angle trouvé par la méthode est affiché pendant 10 secondes.
- Contraintes : il faut que le robot ne détecte que les murs et aucune autre distance sinon la mesure est fausse.
- Dépendances : il faut que la méthode **tourner_deA()** et **getDistance()** fonctionnent.
- Procédure de test : on s'attend à ce que l'angle trouvé par la méthode soit identique à celui donné en paramètre de la méthode à quelque degré près. Cependant, cela ne fut pas le cas.

Test de la méthode **choisir_methode_angle()** :

- Description du test : on fait tourner le robot de 90° et l'angle trouvé par les deux méthodes est affiché pendant 10 secondes ainsi que la méthode choisie.
- Contraintes : pas de contraintes.
- Dépendances : il faut que les méthodes **tourner_deS()**, **getDistance()**, **get_angle_trigo()** et **get_angle_points()** fonctionnent.
- Procédure de test : on s'attend à ce que la méthode choisie soit **get_angle_trigo()** quand les deux angles mesurés sont proches et l'autre méthode sinon. Cependant, nous avons remarqué après avoir effectué ce test avec différents angles et différentes configuration que la méthode **get_angle_trigo()** n'était pratiquement jamais choisie nous avons donc décidé de ne pas utiliser cette méthode.

4. Tests d'intégration pour la classe Actions

Test de la méthode **avancer_deS()** :

- Description : on fait avancer le robot et, tant que celui-ci avance, on affiche une mesure de distance dans la console avec un intervalle de temps régulier.
- Contraintes : il faut que le robot ne soit pas trop loin d'objets sinon la distance renvoyée est infinie.
- Dépendances : il faut que les méthodes **getisMoving()** et **getDistance()** fonctionnent.
- Procédure de test : on s'attend à observer des valeurs s'ajouter dans la console d'affichage du robot tant que le robot avance.

Test de la méthode **tourner_deS()** : le test est identique que le précédent on demande fait simplement tourner le robot au lieu de le faire avancer

Test de la méthode **reculer_deS()** : le test est identique que le précédent on demande fait simplement reculer le robot au lieu de le faire avancer.

Test de la méthode **mouvement_aleatoire()** :

- Description : on fait faire plusieurs mouvements aléatoires au robot. Dans la méthode, on a ajouté un System.out.print afin d'afficher l'angle et la distance à parcourir choisit.

- Contraintes : pas de contraintes.
- Dépendances : il faut que les méthodes permettant d'effectuer des mouvements fonctionnent
- Procédure de test : on s'attend à ce que les angles et distances affichées soient différentes pour les différents mouvements et qu'il ne dépasse pas les intervalles indiqué dans la méthode.

Test de la méthode **attraper_palet()** :

- Description : on dépose un palet à 32 cm du robot et on demande au robot de l'attraper.
- Contraintes : pas de contraintes.
- Dépendances : au préalable nous avons mesuré la distance pour laquelle le capteur de distance ne détecte plus le palet. Pour déterminer cela, nous avons placé le robot en face d'un mur et rapproché un palet du robot en regardant à quel moment celui-ci affichait la distance au mur et non celle du robot. Il faut également que les méthodes **getDistance()**, **avander_deA()**, **ouvrir_pinces()** et **fermer_pincesA()** marchent.

Test de la méthode **eviter_obstacle()** :

- Description : on place un objet d'une taille proche de celle d'un robot et fait tourner le code.
- Contraintes : pas de contraintes.
- Dépendances : nous avons mesuré la taille d'un robot au préalable (qui sera la distance maximale que le robot devra éviter). Il faut également que les méthodes **avancer_deA()** et **tourner_deA()** fonctionnent.
- Procédure de test : on s'attend à ce que le robot n'entre pas en collision avec l'objet et qu'il revienne dans sa direction initiale (avant de devoir contourner l'objet).

Test de la méthode **setVitesse()** :

- Description : on met la vitesse à "lent" on fait avancer le robot d'une certaine distance, ensuite on change la vitesse pour rapide et on fait de nouveau avancer le robot de la même distance
- Contraintes : pas de contraintes
- Dépendances : il faut que la méthode **avancer_deA()** fonctionne.
- Procédure de test : on s'attend ici à voir un changement de rapidité de robot dans le parcours de la distance mis en paramètre dans la méthode **avancer_deA()**.

Test de la méthode **deposer_palet()** :

- Description : on dépose un palet dans les pinces du robot que l'on place en face de la ligne blanche.
- Contraintes : pas de contraintes.
- Dépendances : il faut que les méthodes **ouvrir_pinces()**, **avancer_deS()**, **reculer_deA()** et **fermer_pinces()** fonctionnent.
- Procédure de test : on s'attend ici à ce que le robot dépasse la ligne blanche et ouvre les pinces, recule et puis referme les pinces.

5. Tests d'intégration pour la classe Main

Test de la méthode **get_angle_point()** :

- Description du test : on fait tourner le robot de différents angles (90°, 175°, 15°) puis on affiche l'angle calculé par la méthode pendant 10 secondes à chaque fois.
- Contraintes : pas de contraintes.

- Dépendances : on fait d'abord tourner le robot de 360° à une vitesse "lente" pour savoir combien de point est récupéré pendant un tour complet afin de pouvoir faire le produit en croix. Il faut que la méthode **tourner_deS()** et **getDistance()** fonctionnent.
- Procédure de test : on s'attend à ce que l'angle trouvé par la méthode soit identique à celui donné en paramètre de la méthode à quelque degré près.

Test de la méthode **rechercher_palet()** :

- Description : on place le robot à un endroit aléatoire sur la table et on place des palets sur la table avec des positions aléatoires également. On fait afficher true si le robot à trouver un palet et false sinon. On fait également afficher l'attribut boussole. On effectue ce test plusieurs fois en changeant la position des palets et/ou du robot.
- Contraintes : il faut que les palets soient assez éloignés du robot pour que le capteur de distance puisse les détecter. Il faut également effectuer ce test sur la table pour adapter le seuil de discontinuité en fonction des discontinuité détectée à différents endroits de la table.
- Dépendances : il faut que les méthodes **tourner_deS()**, **getisMoving()**, **getDistance()** fonctionnent.
- Procédure de test : on s'attend ici à ce que le robot s'arrête environ en face d'un palet et à ce que la boussole change de valeur entre le début de la recherche du palet et la fin. Le robot ne s'arrêtait pas toujours en face d'un palet, il détectait souvent un mur ou bien un coin de la table.

Test des différents switch-case :

- Descriptions : on isole ici un état et on exécute programme pour voir si le robot effectue ce qui est voulu dans l'état d'entrée.
- Contraintes : il faut effectuer ce test sur la table car pour l'état du début par exemple il nous faut prendre des mesures de distances.
- Dépendances : il faut que toutes les méthodes présentées précédemment fonctionnent.
- Procédure de test : on s'attend à ce que dans un état donné le robot fasse ce qui est attendu, par exemple rechercher un palet dans l'état recherche palet et s'il n'est pas trouvé effectuer un mouvement aléatoire.

Test de la méthode **run()** :

- Description : on exécute juste le code ici en affichant les différents état par lequel le robot passe pendant un délai de 5 secondes à chaque fois qu'il change d'état.
- Contraintes : il faut également effectuer ce test sur le terrain.
- Dépendances : il faut que toutes les méthodes présentées précédemment marchent
- Procédure de test : on s'attend ici à ce qu'il prennent les décisions définies dans l'automate présenté dans le plan de développement. Nous vérifions ici que le robot prend les bonnes décisions.

IV- Annexes

1. Récapitulatif du fonctionnement des différentes méthodes

Fonctions	Fonctionne	Fonctionne partiellement	Ne fonctionne pas	Fonctionne mais abandon de la méthode	Fonction non réalisée
Classe Actions					
tourner_de(angle) (synchrone et asynchrone)					
ouvrir_pince					
fermer_pince					
setVitesse					
mouvement_aleatoire					
avancer_sans_palet					
sarreter_sans_palet					
attraper_palet					
getisMoving					
avancer_avec_palet					
eviter_obstacle					
avancer_de(distance) (synchrone et asynchrone)					
sarreter_avec_palet					
reculer_de(distance)					
deposer_palet					
Classe Capteurs					
get_distance					
stocker_distance					
get_pression					
get_couleur					
get_position					
get_angle_trigo					
choisir_methode_mesure_dangle					
get_orientation					
stocker_orientation					
Classe main					
boussole					
get_angle_points					
rechercher_palet					
etat : debut					
etat : rechercher_palet					
etat : prendre_palet					
etat : ramener_palet					
run					

2. Implémentation des fonctions non utilisées

Classe Capteurs

```
private ArrayList<Double> distances;
private ArrayList<Double> historiqueOrientations; // Historique des orientations
//Méthode pour fermer le capteur de couleur
	public void closeSensors() {
		colorSensor.close(); // Fermer le capteur lorsque ce n'est plus nécessaire
	}
 public ArrayList<Double> get_distances(){
		return distances;
	}
 // Méthode pour mettre à jour et obtenir l'orientation actuelle du robot
	public double get_orientation(double angleRotation) {
		orientation += angleRotation;
		if(orientation > 180) {
			orientation -= 360;
			System.out.print(orientation);
		}
		else if (orientation <= -180){
			orientation += 360;
		}
		return orientation;
	}
     //Méthode pour stocker l'orientation
	public ArrayList<Double> stocker_orientation() {
		historiqueOrientations.add(orientation);
		return historiqueOrientations;
	}
```

```
//Méthode pour détecter les discontinuités dans les distances mesurées
	public List<Integer> detecterDiscontinuite(){
		int l = distances.size();
		List<Integer> indicesDiff = new ArrayList<>();
		int seuil = 10; // A CHANGER JE SAIS PAS DU TOUT COMMENT ON VA DEFINIR CE
SEUIL
		for (int i = 0; i< l; i++) {
			float diff = (float) Math.abs(distances.get(i+1) -
distances.get(i));
			if (diff > seuil) {
				indicesDiff.add(i);
			}
		}
		return indicesDiff;
	}
     //Méthode pour fermer le capteur ultrasonique
	public void closeUltrasonicSensor() {
		ultrasonicSensor.close(); // Fermer le capteur lorsqu'il n'est plus
nécessaire
	}
//Méthode pour fermer le capteur tactile
	public void closepressionSensor() {
		pressionSensor.close(); // Fermer le capteur lorsque ce n'est plus
nécessaire
	}
     //Méthode privée pour lire la distance (simule la lecture d'un capteur)
	private double lireDistance() {
		// Code pour lire la distance depuis le capteur à ultrasons
		// Simule une valeur aléatoire pour l'exemple
		return this.getDistance();  // À remplacer par la lecture réelle du capteur
	}


// Méthode pour calculer l'angle de rotation en degrés avec la trigonométrie
	public double getAngleTrigo() {
			// Mesure de la distance avant la rotation
			this.distanceInitiale = this.lireDistance();
			System.out.println("Distance initiale: " + distanceInitiale + " cm");
			// Effectuer la rotation du robot (simulation)
			// Ajoutez ici le code pour tourner le robot
			// Mesure de la distance après la rotation
			this.distanceFinale = lireDistance();
			System.out.println("Distance finale: " + distanceFinale + " cm");
			// Vérification que les distances ne sont pas nulles pour éviter la
division par zéro
			if (distanceInitiale == 0 || distanceFinale == 0) {
				System.err.println("Les distances ne peuvent pas être égales à
zéro.");
				return -1; // Indiquer une erreur
			}
			// Calcul de l'angle de rotation en utilisant la trigonométrie
			double cosTheta = distanceFinale / distanceInitiale;
			double angleRadians = Math.acos(cosTheta); // Calcul de l'angle en
radians
			double angleDegres = Math.toDegrees(angleRadians); // Conversion en
degrés
			System.out.printf("Angle de rotation: %.2f degrés%n", angleDegres);
			return angleDegres;
		}
	// Méthode pour calculer l'angle de rotation en degrés à partir du nombre de points
mesurés


	public double choisir_methode_mesure_dangle(int nombrePointsMesures) {
			// Calcul des angles à partir des deux méthodes
			double angleTrigo = getAngleTrigo();
			double anglePoints = getAnglePoints(nombrePointsMesures);
```

```
                        // Vérification des erreurs pour les deux méthodes
                        if (angleTrigo == -1 || anglePoints == -1) {
                                System.err.println("Erreur lors du calcul des angles.");
                                return -1; // Indiquer une erreur si l'un des calculs a échoué
                        }
                        // Comparaison des deux angles
                        double difference = Math.abs(angleTrigo - anglePoints);
                        //System.out.printf("Différence entre les angles: %.2f degrés%n",
difference);
                        // Privilégier la méthode basée sur les points si la différence est
significative
                        if (difference > 10.0) { // Par exemple, une différence de plus de 10
degrés
                                //System.out.println("Utilisation de la méthode
getAnglePoints");
                                return anglePoints; // Retourner l'angle basé sur les points
                        } else {
                                System.out.println("Utilisation de la méthode getAngleTrigo");
                                return angleTrigo; // Retourner l'angle basé sur les distances
                        }
                }
```

Classe Actions

```
 // Retourne l'état des pinces (ouvertes ou fermées)
        public boolean getEstOuvert() {
                return this.estOuverte;
        }
```