Application robotique dronique

Vincent Lecrubier

Introduction

L'objectif de ce BE est de créer un robot sous RoboCode, en mettant en œuvre les concepts de robotique dronique appris en cours. La version de Robocode utilisée dans ce travail est la 1.7.2.0.

Deux objectifs principaux sont donnés : créer un robot répondant à une certaine mission, puis le modifier pour le rendre capable de coopérer au sein d'une équipe. La version du robot présenté ici est la version de Jeu en équipe, qui est capable de jouer aussi en solo, remplissant les deux fonctions.

Table des matières

Mission

La mission du robot est définie par les éléments suivants :

- Explorer un terrain de jeu de taille 5000*5000 dans son intégralité en temps fini.
- Détruire tous les robots de type « sample.sittingDuck » du terrain de jeu.
- Coopérer avec d'autres robots du même type.
- Éventuellement, être capable de détruire des robots d'autres types.

Afin de rendre un travail intéressant et proche des problématiques de l'intelligence artificielle, il a été décidé de créer un robot non déterministe. Les actions du robot, si elles suivent une certaine logique, sont programmées de manière à ne pas être reproductibles, et mettent en jeu beaucoup de paramètres aléatoires.

Ce choix et justifié par plusieurs éléments :

- Le comportement est plus robuste, c'est à dire qu'il risque moins de générer des situations bloquantes pour le robot de manière récurrente. Le robot réagis de manière différente à chaque fois, il a donc peu de chance de refaire la même erreur à chaque exécution.
- Comportement imprévisible pour un adversaire éventuel. Les robots de compétition sous Robocode utilisent des techniques de « pattern-matching » permettant de détecter et d'anticiper les réactions du robot adverse, si celui ci a des comportements récurrents. Le robot présenté sera moins prévisible pour ses adversaires éventuels.
- Au niveau pédagogique, il est plus intéressant de créer des solutions générales à des problèmes variés, plutôt que de mettre en place des algorithmes déterministes dictant le comportement du robot comme une télécommande.

Cependant le choix effectué présente aussi des inconvénients :

- Le temps d'exécution du robot peut devenir assez lent, surtout en cas de prise de décision, l'amenant parfois à dépasser son temps d'exécution imparti. Le robot passe alors son tour.
- Certains problèmes rares et peu reproductibles apparaissent parfois, la diversité de comportements du robot amenant une diversité de situations complexes.
- Le robot n'est pas optimal pour la mission de base demandée. Un robot déterministe irait beaucoup plus vite, mais serait rapidement perdu face a une situation différente. Le robot présenté s'adapte à toutes les situations, et il est encore possible d'améliorer largement cette adaptabilité, ce qui ne serais possible avec un robot « algorithmique » qu'au prix de milliers de lignes de code.

Présentation du Robot

Le robot, dénommé « CrubiTeamBot », est de type TeamRobot.

Les structures importantes du robot sont :

- Un automate à états permettant de basculer entre les modes de fonctionnement, représentant l'état **présent** du robot.
- Une structure contenant la planification du **futur** du robot.
- Une carte virtuelle du terrain de jeu, comprenant les informations recueillies dans le **passé** par le robot.
- Un moyen d'échange avec les autres robots.

La classe java Mode contenue dans le code du robot permet de représenter tous ces éléments. Nous allons maintenant détailler le fonctionnement de ces différents éléments.

Automate à états

Les états

Afin de maitriser avec précision les actions du robot, chaque état est divisé en plusieurs étapes. Chaque étape correspond à une ou plusieurs actions élémentaires effectuées en parallèle par le robot. La succession d'étapes permet d'ordonner la séquence d'actions correspondant à chaque état de manière claire.

On peut prouver que cette solution est équivalente à un automate à états simples (sans étapes). Par exemple, un état composé de 6 étapes pourrait être remplacé par 6 états simples différents.

Cependant la solution retenue est plus lisible et beaucoup plus facile à faire évoluer.

Le robot peut être dans trois états différents, et plusieurs étapes pour chacun :

- EXPLORE : le robot explore le terrain de jeu.
    1. Tourner le radar à 360° dans un sens aléatoire et commencer à choisir une destination.
    2. Tourner en direction de la destination, et mettre le radar à la perpendiculaire par rapport à la direction
    3. Avancer en ligne droite jusqu'à destination puis revenir à 1.
- TRACK : le robot traque un ennemi en esquivant les tirs, balayant avec son radar en direction de l'ennemi.
    1. Tourner dans une direction aléatoire, tourner le radar vers l'ennemi
    2. Avancer ou reculer d'une distance aléatoire
    3. Tourner le radar à 360° puis passer en mode EXPLORE.
- ATTACK : le robot ajuste son canon, et tire à plusieurs reprises sur l'ennemi.
    1. Calculer le point a viser en fonction de l'ennemi et son déplacement, tourner le canon dans la direction choisie
    2. Tirer si le canon n'est pas trop chaud, si la position de l'ennemi est validée, sinon, ne rien faire.
    3. Au bout d'un certain nombre d'itérations des étapes 1 et 2, passer en mode TRACK.

L'automate à états stocke l'état courant du robot sous forme d'un entier. Des variables statiques permettent de donner un nom aux états :

```
public static final int EXPLORE = 1;
public static final int ATTACK = 2;
public static final int TRACK = 3;
```

L'étape courante est stockée sous forme d'un entier.

L'état courant est déterminé par deux structures switch imbriquées : l'une pour l'état, et pour chaque état, un switch pour l'étape. Ainsi la méthode run() a la structure suivante :

```
do {
	switch (mode.get()) {

	case Mode.ATTACK:
		switch (mode.getStep()) {
		case 1:
			//Code Action ATTACK.1
			break;
		case 2:
			//Code Action ATTACK.2
			break;
		//ETC
		//...
		}

	case Mode.EXPLORE:
		switch (mode.getStep()) {
		case 1:
			//Code Action EXPLORE.1
			break;
		case 2:
			//Code Action EXPLORE.2
			break;
		//ETC
		//...
		}

	//ETC
	//...

	}
}
```

Les transitions

Les transitions sont déclenchées par plusieurs types de causes :

- Déclenchée automatiquement par l'arrivée à une certaine étape au sein de l'état courant. (Exemple : Voir plus haut : Mode TRACK.3 → EXPLORE.1)
- Déclenchée par un événement de simulation (Exemple : HitByBullet)
- Déclenchée par un évènement interne au robot (Exemple : le robot est suffisamment proche de sa cible)

Lorsqu'un évènement de simulation se produit, il peut déclencher une transition, en fonction de l'état actif. Par exemple, une collision avec un robot ennemi n'aura pas le même effet selon l'état dans lequel est le robot : S'il est en mode ATTACK ou TRACK, il continuera comme si de rien n'était, mais s'il est en mode EXPLORE, alors il passera en mode TRACK pour tirer sur le robot incriminé. Le code correspondant sera par exemple :

```
switch (mode.get()) {
case Mode.EXPLORE:
      mode.set(Mode.TRACK, 20);
      break;
case Mode.ATTACK:
case Mode.TRACK:
      break;
}
```

Plusieurs causes différentes peuvent vouloir déclencher différentes transitions au même pas de temps. Afin de conserver la maitrise du comportement du robot, il faut définir des priorités pour les différentes transitions partant d'un même état. A la fin du pas de temps, la transition ayant la priorité la plus grande sera effectivement tirée. Dans l'exemple précédent, on voit que le passage en mode TRACK est donné avec une priorité de 20. Si, au même instant, le robot reçoit un événement qui déclenchera la transition en mode EXPLORE avec la priorité 5 selon l'appel suivant :

```
mode.set(Mode.EXPLORE, 5);
```

Alors le robot passera en mode TRACK, car la priorité de la transition vers le mode TRACK est plus importante, malgré que l'appel demandant une transition vers le mode EXPLORE soit postérieur à l'appel vers le mode TRACK.

Lorsque toutes les actions correspondant à l'étape courante de l'état courant sont terminées, alors la transition est tirée. Ce système évite que le comportement du robot soit erratique suite à différents changements de modes dus à différents évènements de simulation lors d'un même pas de temps.

Le code source permet d'avoir les détails du diagramme d'états.

Exemple d'une trace d'exécution du jeu, contenant le mode actif et les évènements de simulation :

```
      EXPLORE 1
      EXPLORE 2
→ ScannedRobot
      ATTACK  0
→ ScannedRobot
      ATTACK  1
      ATTACK  2
→ ScannedRobot
      ATTACK  3
      ATTACK  4
→ RobotKilled
      EXPLORE  0
      EXPLORE  1
      EXPLORE  2
→ ScannedRobot
      ATTACK  0
```

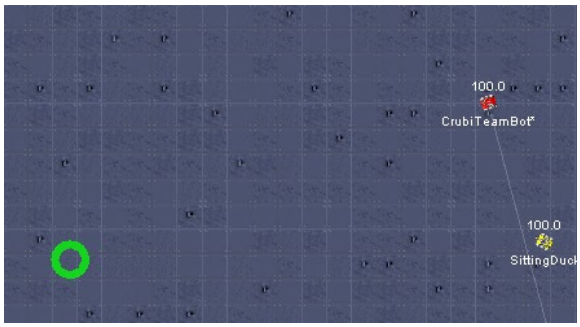
Planification

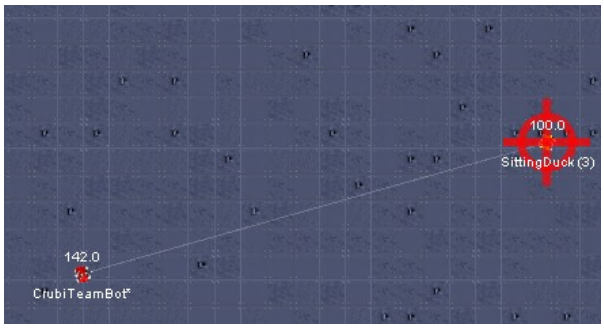
Cible

Le robot planifie ses actions de manière assez simple, en utilisant un seul point, représentant la cible. Selon le mode actuel du robot, la cible a différentes significations :

- ATTACK : La cible représente la position de l'ennemi.
- TRACK : La cible représente la direction menant vers la position probable de l'ennemi.
- EXPLORE : La cible représente la prochaine destination a atteindre.

Il est possible de montrer la Cible actuelle en activant le mode Paint du robot dans Robocode :





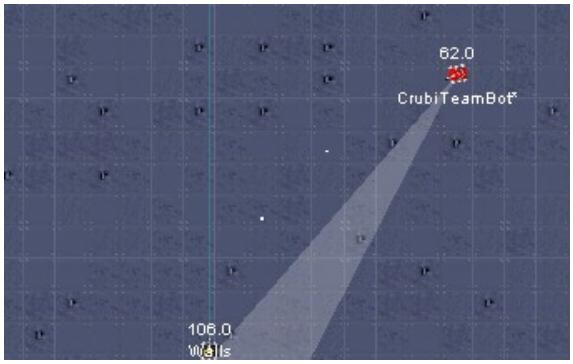
La cible est mise à jour dès que le robot recueille des informations intéressantes permettant d'en savoir plus.

En mode exploration, la cible est placé de manière non déterministe, et de manière à parcourir tout le champ de bataille en un temps réduit.

Anticipation des mouvements ennemis

Le robot est doté d'un algorithme assez simple permettant d'anticiper les mouvement du robot cible lorsqu'il l'attaque. Ce système repose sur l'hypothèse que le robot cible continuera son mouvement a vitesse constante et taux de virage constant. A la suite de quelques itérations, le robot trouve un angle de tir permettant de toucher sa cible même si celle ci est en mouvement. Si la position prédite de la cible lors de l'impact est en dehors du champ de bataille, alors cela veut dire que la cible aura forcément changé de direction d'ici la. Le robot annule donc le tir.

Un exemple de tir avec anticipation :



Compréhension de l'environnement

Système implémenté

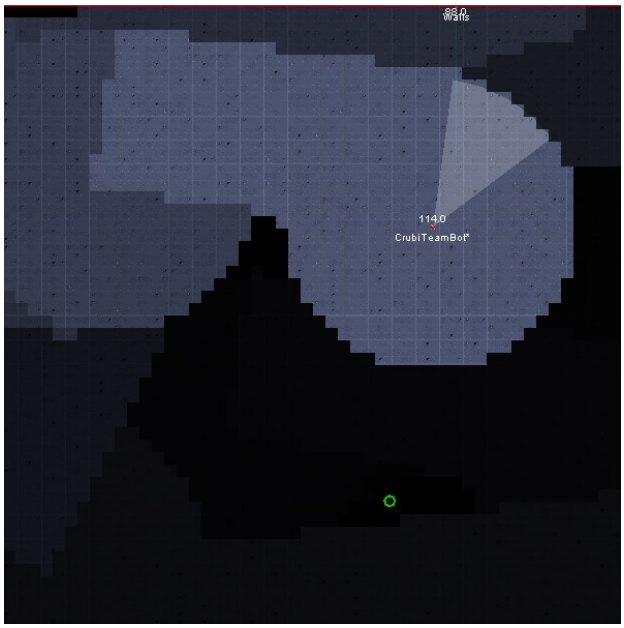
Afin d'optimiser le comportement du robot, il a été décidé de jeter les bases d'un comportement plus évolué. Pour cela il faut conserver des traces du passé du robot.

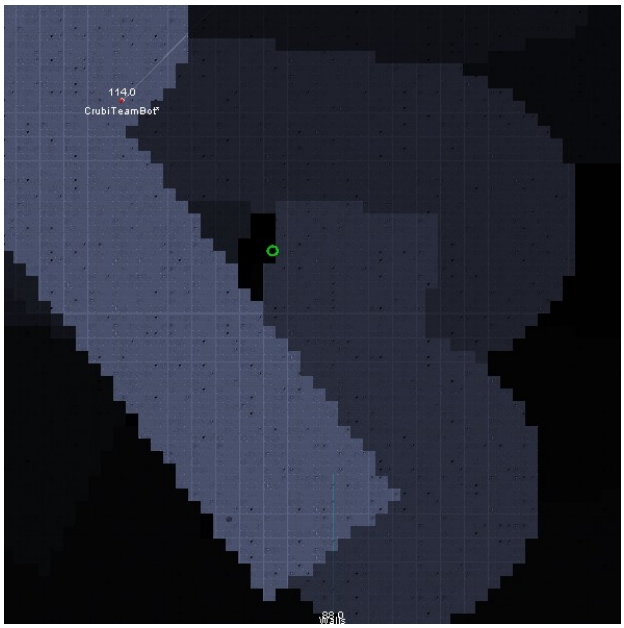
Le robot conserve donc en interne une carte du champ de bataille. En chaque point, la valeur de la carte représente le temps depuis lequel le robot est passé en ce point. La valeur 255 (transparent sur les images) signifie que le robot vient de voir le point dans son radar, alors que la valeur 0 (noir), signifie que le robot n'a pas vu ce point depuis un temps infini. A la fin de chaque mouvement, le robot entre dans la carte les endroits qu'il a déjà scanné.

Afin de faciliter le dessin des formes que le robot a scanné, la carte est stockée sous la forme d'une bufferedImage, ce qui permet d'y tracer des dessins avec les méthodes de Java.Graphics2D.

Lorsque le robot est en mode EXPLORE, il utilise la carte pour choisir quel est le point suivant à visiter. Pour cela, il sélectionne sur la carte les points dont la valeur est minimale, c'est à dire ceux qu'il a visité le plus anciennement. Lorsqu'il y a plusieurs points avec la même valeur (typiquement au début, lorsque le robot n'a rien visité et que tous les points sont donc a zéro), le robot les liste tous et en choisi un au hasard parmi eux. Enfin, la destination finale du robot est trouvée en ajoutant une petite composante aléatoire à la cible.

On voit dans les exemples suivants que le robot choisit un point qu'il n'a jamais visité afin de continuer son exploration.





Améliorations possibles

Afin d'optimiser le temps d'exploration du robot, il faudrait modifier l'heuristique permettant de déterminer quelle destination choisir. Au lieu de se baser sur le seul point de destination, on pourrait attribuer un score à chaque destination potentielle. Une formule de score intéressante pourrait être :

$$\text{Score} = \frac{\text{Surface encore inconnue balayée par le robot durant son trajet vers la cible}}{\text{Longueur du trajet vers la cible}}$$

Cependant, en l'état actuel de non-optimisation du code, le temps de calcul serait trop long, c'est pourquoi cette solution n'est pas implantée, au profit d'une solution qui apporte quand même un gain de temps par rapport à un ciblage aléatoire.

Évolutions futures possibles

En plus d'une carte, un robot plus avancé pourrait se faire une représentation interne de chaque robot présent sur le terrain de jeu, cette représentation pourrait contenir :

- L'énergie du robot, estimée a partir des tirs effectués sur celui ci.
- L'agressivité du robot, afin de savoir si il faut l'éliminer en premier, le fuir...
- Une évaluation de la position du robot, basée sur ses dernières positions connues, sa vitesse, etc...

Ces informations et estimations permettraient d'avoir un comportement plus optimal.

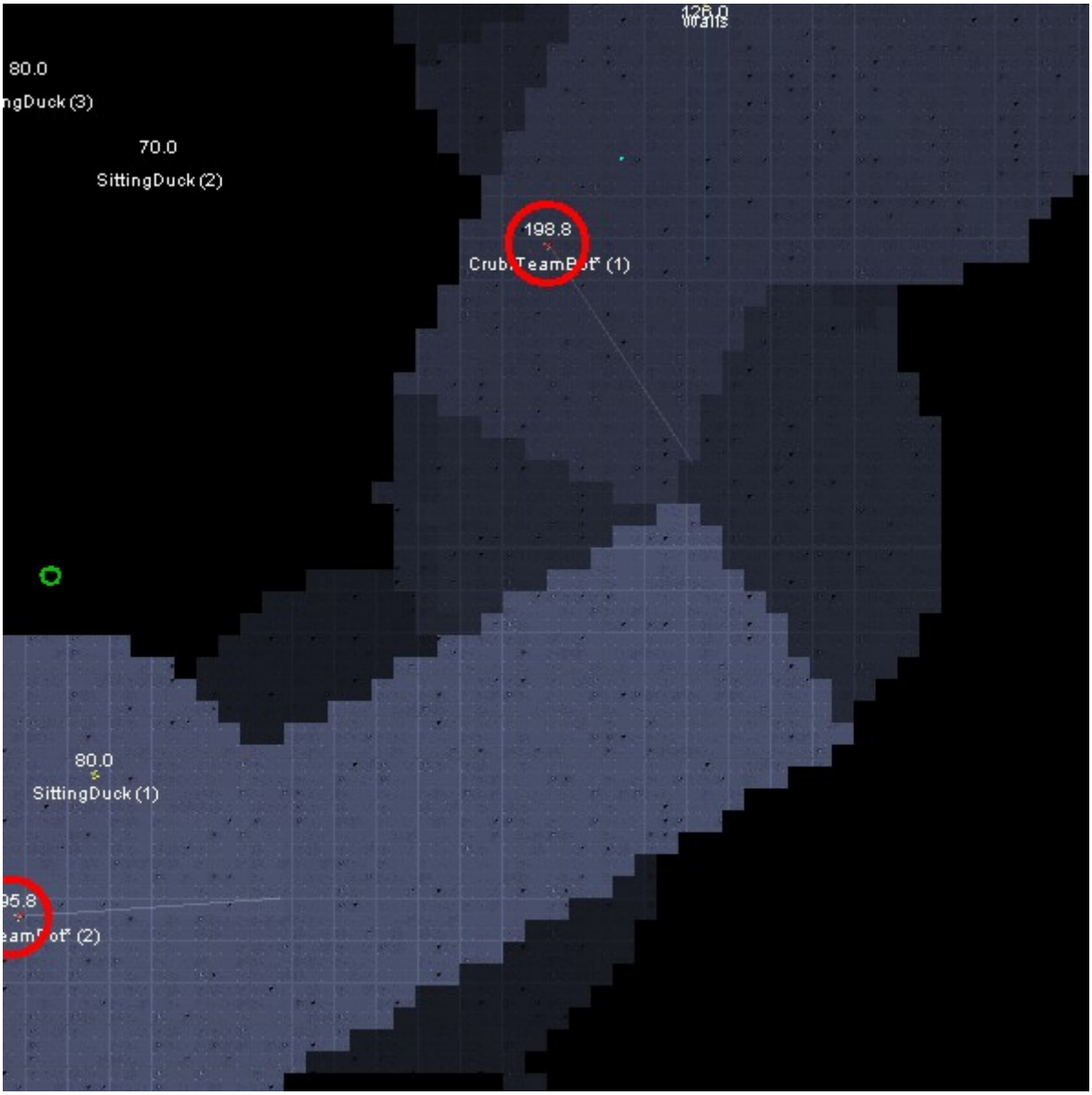
Coopération

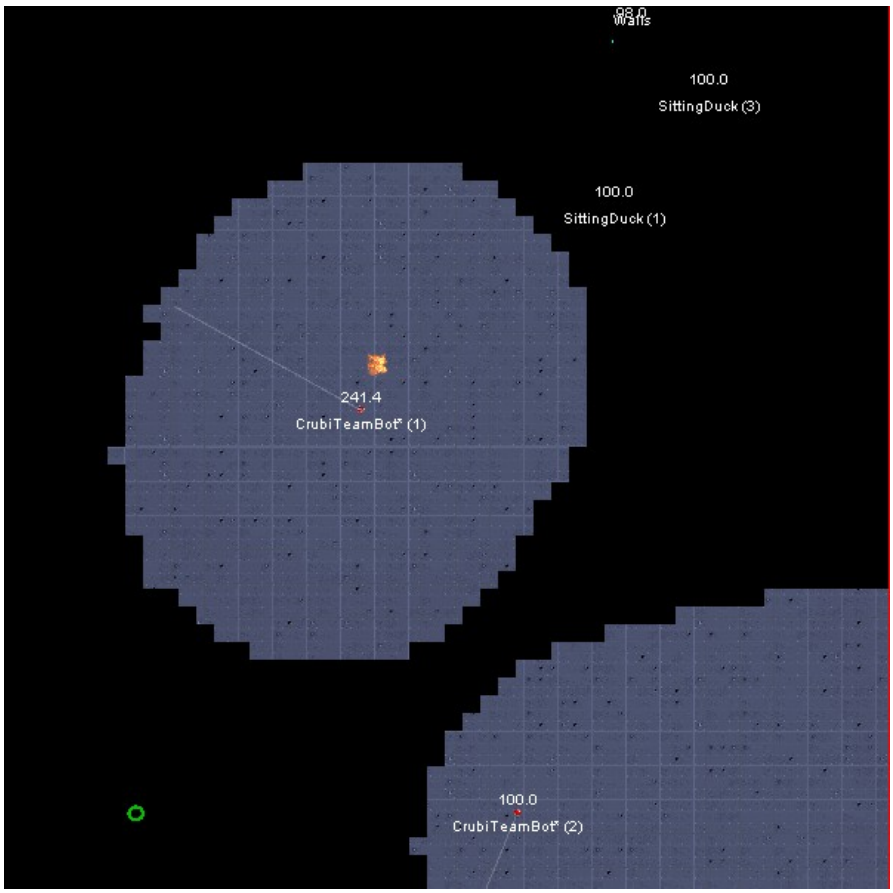
Pour le jeu en équipe, les robots sont strictement identiques, il n'y a pas de maitre, ainsi la mort d'un robot particulier ne signifie pas la mort de l'équipe. Les robots fonctionnent en quelque sorte en mode « pair à pair ».

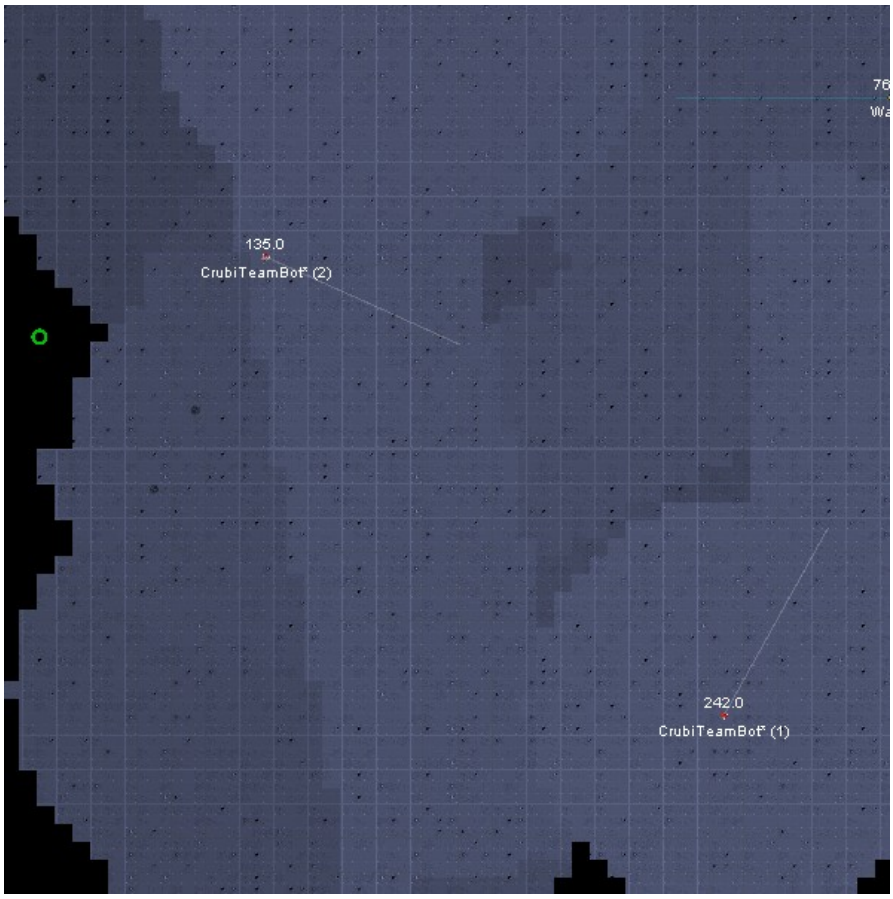
La coopération entre les robots est implémentée de manière simpliste. En effet, les robots échangent leur carte interne. Dès qu'un robot signale avoir exploré certains points de la carte, il envoie la carte à ses coéquipiers qui mettent a jour la leur. Deux robots n'iront donc pas visiter le même endroit.

Cependant, ce mode de coopération est dépendant du mode de choix des cibles, et du choix des heuristiques, débattu plus haut. De plus, au delà de 4 coéquipiers, l'empreinte mémoire et les synchronisations de cartes deviennent trop lourdes et il arrive que les robots soient éliminés.

Le reste du temps, tout se passe comme si les robots partageaient la même carte, et le temps d'exploration s'en trouve considérablement réduit.




100.0
SittingDuck (3)
100.0
SittingDuck (1)
241.4
CrubiTeamBot* (1)
100.0
CrubiTeamBot* (2)


135.0
CrubiTeamBot* (2)
242.0
CrubiTeamBot* (1)