MANUEL WARBOT II

Prise en main d'un jeu de guerre dont vous êtes le héros en Java

Jacques Ferber **Jessy Bonnotte** **Fabien Michel**

Burc Pierre **Duplouy Olivier** **Polizzi Mathieu**

FMIN207 – M1 Informatique
TP Programmation Orientée Agents
Année 2013

I. Introduction

WARBOT II est la réécriture en Java de la plateforme WarLogo, dédiée à la simulation dans un environnement logique et graphique d'un jeu de stratégie pour y intégrer une intelligence artificielle agissant au niveau individuel. Divers projets similaires existent pour des jeux du marché (par exemple Pogamut permet d'intégrer des agents pour Unreal Tournament et Defcon). L'outil développé ici est volontairement plus simple pour vous permettre de produire une IA simple rapidement tout en laissant la possibilité à ceux qui le souhaitent de complexifier le comportement de leurs agents.

L'objectif de ce TP est de développer des agents au comportement intelligent afin de satisfaire le but du jeu : détruire la base ennemie à l'aide de ses robots. La définition d'agent est encore floue dans la communauté mais en voici une idée simple. Un agent est un "esprit" qui habite un "corps". Cet "esprit" ne peut voir qu'au travers des yeux de son "corps" et ne peut agir que via les effecteurs de son corps (bras, jambes). Plus formellement, votre agent a le pouvoir de percevoir via son environnement et doit décider d'une action à réaliser. L'environnement décidera du résultat de son action (et celle des autres agents) et lui fournira de nouveaux percepts.

Dans ce document, vous trouverez tout ce dont vous avez besoin pour connaitre le fonctionnement de l'environnement, des percepts et des actions.

II. Les différents types d'agents

Il y a trois types de corps dont vous aurez à programmer le comportement : bases, exploreurs et rocket-launchers.

1) Bases

Les bases représentent le quartier général de votre équipe. Ce sont des agents ne pouvant pas se déplacer physiquement dans le monde. En revanche elles peuvent construire des unités de type exploreurs et rocket-launchers.

2) Exploreurs

Les exploreurs sont des agents dédiés à l'exploration du monde. Ils sont pacifistes et ne peuvent donc pas attaquer d'unités ennemies. Ils peuvent se déplacer et servent donc essentiellement à découvrir le territoire, le surveiller, récupérer de l'énergie sous forme de nourriture dispersée dans l'environnement. À terme ils permettront de donner des informations sur le reste du monde aux autres agents grâce à un module de communication.

3) Rocket-launchers

Les rocket-launchers sont les seules unités à pouvoir se battre. Ils peuvent se déplacer dans l'environnement et tirer des missiles (qui affectent les amis comme les ennemis). Ils ne peuvent tirer un missile que tout les k tours (k représentant un nombre de ticks prédéterminé). Contrairement à son prédécesseur, les rocket-launchers ne peuvent plus stocker plusieurs missiles. Les missiles seront construits automatiquement, quand l'action « tirer » sera appelée. Bien entendu, cette action entraînera une perte de vie du rocket-launcher. Les coûts des différentes unités sont indiqués plus bas dans le document.

III. Fonctionnement global du programme

Le paradigme agent limite vos possibilités de perception et d'action sur l'environnement à l'API que nous vous proposons. Imaginez-vous enfermé dans un sous-marin. Vous avez pour seule information un moniteur qui vous délivre certaines informations (écho sonar, état de la coque...) et un ensemble de manettes à tirer devant vous. La carlingue et l'océan décident pour vous du résultat. Essayez de vous placer en vue subjective.

WARBOT II a été conçu dans l'optique de gommer certaines contraintes imposées par NetLogo à WarLogo. En effet, en gardant l'exemple du sous-marin, nous pouvions autrefois avoir accès à l'ensemble des informations présentes dans l'océan, où que nous soyons. Maintenant, seules les informations perçues à un instant T sont exploitables par l'agent. Celles-ci ne seront pas mises à jour, car les informations reçues ne sont plus des pointeurs, mais seulement des données qui indiquent pour cet instant précis l'état de l'environnement présent dans notre champ de perception.

Vous êtes donc autorisés à stocker tout ce qui vous semble utile (énergie, identifiant, type d'agent, etc…), ceci constituant vos « croyances ». Celles-ci pourront être partagées avec le reste de votre équipe, par le biais d'un système d'envoi de messages.

IV. Les actions

Chaque action coûte un tour à l'agent. Vous devez donc soigneusement choisir la meilleure action à lui faire effectuer en fonction du contexte qu'il perçoit et de son environnement.

1) Actions communes à toutes les unités

- **Eat** : Décrémente le nombre de food dans le sac de 1 et ajoute 200 PV à l'unité.
- **Idle** : l'agent ne bougera plus et ne fera aucune action.
- **Give** : Donner une ressource à un agent. Juste avant cette action, faire un setAgentToGive(int id_agent).

2) Actions de l'exploreur

- **Take** : Permet de ramasser une ressource présente sur le terrain. Si le sac est plein, rien ne se passe.

3) Actions de la base

- **Create** : Permet de créer un agent de type Explorer ou RocketLauncher. Juste avant cette action, faire un setNextAgentCreate(String agent).

4) Actions du rocket-launcher

- **Fire** : Envoyer une roquette en direction d'un agent ennemi. Avant cette action, faire un setAngleTurret(int angle), pour diriger la tourelle en direction de l'ennemi.
- **Reload** : Une fois enclenché, le RocketLauncher doit attendre un laps de temps de 50 ticks avant de pourvoir tirer.

- **Take** : Permet de ramasser une ressource présente sur le terrain. Si le sac est plein, rien ne se passe.

V. Primitives liées aux agents

Une multitude de primitives ont été mises en place pour vous faciliter le travail. Certaines renvoient des valeurs booléennes sur des choses que vous aurez très souvent à évaluer avant d'agir, d'autres renvoient des informations plus générales, comme l'identifiant de l'agent, son numéro d'équipe, etc…

Toutes les primitives liées aux agents sont citées ci-dessous, avec la signature et le rôle de chacune.

1) Primitives communes à tous les agents

- **emptyBag () : Boolean**
  - Retourne vrai si le sac est vide, faux sinon.
- **fullBag () : Boolean**
  - Retourne vrai si le sac est plein, faux sinon.
- **getEnergy () : Integer**
  - Retourne l'énergie actuelle de l'agent.
- **getID () : Integer**
  - Retourne l'id de l'agent.
- **getTeam () : String**
  - Retourne le nom d'équipe de l'agent.
- **broadCastMessage (String unite, String message, String[] content)**
  - Permet de diffuser le message « message » au type d'agent unite, et optionnellement des informations complémentaires dans le « content ».
- **sendMessage(int agent, String message, String[] content)**
  - Permet d'envoyer le message « message » à l'unité agent, et optionnellement des informations complémentaires dans le « content ».
- **reply(WarMessageFinal wmf, String message, String[] content)**
  - Permet de répondre au message wmf, avec le message "message" et optionnellement des informations complémentaires dans le « content ».
- **getPercepts () : List<Percept>**
  - Retourne une liste contenant l'ensemble des unités perçues par l'agent.
- **getMessage () : List<WarMessageFinal>**
  - Si appelée, retourne une liste contenant l'ensemble des messages reçus au tick actuel.

2) Primitives propres au rocket-launcher

- **getHeading () : Integer**
  - Retourne l'angle actuel de l'agent.
- **isBlocked () : Boolean**
  - Retourne vrai si l'agent est bloqué contre un bord, faux sinon.
- **isReloaded () : Boolean**
  - Retourne vrai si l'agent a déjà « rechargé », faux sinon.
- **isReloading() : Boolean**
  - Retourne vrai si l'agent est en train de « recharger », faux sinon.
- **setAgentToGive (int ID)**
  - Méthode appelée juste avant l'action « give ». Permettra de donner une ressource à l'agent ayant l'identifiant ID.
- **setAngleTurret (int angle)**
  - Placer la tourelle dans un certain angle pour tirer par la suite.
- **setHeading (double angle)**
  - Permet de changer de direction en modifiant la trajectoire.
- **setRandomHeading ()**
  - Change aléatoirement la trajectoire de l'agent.

3) Primitives propres à la base

- **setNextAgentCreate (String agent)**
  - Méthode appelée juste avant l'action « create ». Permet d'indiquer quel type d'agent sera créé.

4) Primitives propres à l'exploreur

- **getHeading () : Integer**
  - Retourne l'angle actuel de l'agent.
- **isBlocked () : Boolean**
  - Retourne vrai si l'agent est bloqué contre un bord, faux sinon.
- **setAgentToGive (int ID)**
  - Permettra de donner une ressource à l'agent ayant l'identifiant ID.
- **setHeading (double angle)**
  - Permet de changer de direction en modifiant la trajectoire.
- **setRandomHeading ()**
  - Change aléatoirement la trajectoire de l'agent.

VI. Primitives liées aux percepts

L'ensemble de ces primitives s'appliquent aux objets de types Percept, donc aux objets obtenus grâce à la primitive getPercept(), citée plus haut.

- **getAngle() : Integer**
  - Retourne l'angle où se trouve l'unité perçue, par rapport à notre propre direction.
- **getDistance() : Integer**
  - Retourne la distance qu'il y a entre l'unité perçue et nous-même.
- **getEnergy() : Integer**
  - Retourne l'énergie actuelle de l'unité perçue.
- **getId() : Integer**
  - Retourne l'identifiant de l'unité perçue.
- **getTeam() : Integer**
  - Retourne le nom d'équipe de l'unité perçue, ou une chaîne vide si l'agent est de type « WarFood ».
- **getType() : String**
  - Retourne le type de l'unité perçue, les différents types étant : « WarFood », « WarExplorer », « WarRocketLauncher », « WarBase » et « WarRocket ».

VII. Primitives liées aux messages

L'ensemble de ces primitives s'appliquent aux objets de types WarMessageFinal, donc aux objets obtenus grâce à la primitive getMessage(), citée plus haut.

- **getAngle() : Integer**
  - Retourne l'angle où se trouve l'agent nous ayant envoyé un message, par rapport à notre propre direction.
- **getContent() : String[]**
  - Retourne, s'il y en a, les informations complémentaires envoyées en tant que troisième paramètre de la primitive sendMessage() ou broadcastMessage().
- **getDistance() : Integer**
  - Retourne la distance qu'il y a entre nous et l'agent nous ayant envoyé le message.
- **getMessage() : String**

    - Retourne le message en lui-même, contenu dans le deuxième paramètre de la primitive sendMessage() ou broadcastMessage().
- **getSender() : Integer**
    - Retourne l'identifiant de l'agent nous ayant envoyé le message.
- **getSenderTeam() : String**
    - Uniquement si on a des alliés, donc pas pour maintenant.
- **getType() : String**
    - Retourne le type de l'unité qui nous a envoyé le message.

VIII. Définition de comportement

Plusieurs équipes sont fournies pour que vous puissiez les confronter les unes aux autres, pour voir les différents comportements. Cela va vous permettre de pouvoir vous mesurer localement à des agents basiques avant de pouvoir vous mettre en situation de compétition avec les autres étudiants. Vous devez donc vous occuper uniquement de la programmation des brains de vos agents. Pour définir par vous-même un comportement pour chacun des trois types d'agents, vous allez devoir définir 3 classes dans un nouveau package que vous créerez, chacune correspondant à l'IA d'une unité. Le nom n'a aucune importance, mais ces 3 classes doivent hériter de WarBrain, vous permettant d'avoir accès à l'ensemble des primitives citées plus haut.

Voici un exemple, pour le brain de la base :

```
public class BrainBase extends WarBrain{


	public BrainBase(){

	}

	@Override
	public String action() {

	if(!emptyBag()){
			return "eat";
	}

	.

	.

	}
```

C'est dans la méthode action() que vous implémenterez votre IA, avec bien entendu la possibilité de coder des fonctions annexes. La String retournée par action() correspondra à l'action que vous voulez effectuer pour ce tour. Une fois ces 3 classes implémentées, il vous faudra créer un JAR, contenant une multitude d'informations.

IX. Fichiers nécessaires pour la création du JAR

WARBOT II, à son démarrage, scanne l'ensemble des JARs présents dans le dossier jar, dans Tk2. Il faut donc le générer, mais il ne contiendra pas que les 3 classes constituant vos brains. Une image représentant votre logo d'équipe devra être ajoutée, le format n'étant pas important (png, jpeg, …). Une musique de victoire (au format wav) pourra également être incluse (Le fichier son est optionnel). Celle-ci se lancera dès lors que vous aurez remporté une rencontre. Un son d'une dizaine de secondes maximum est conseillé. . .

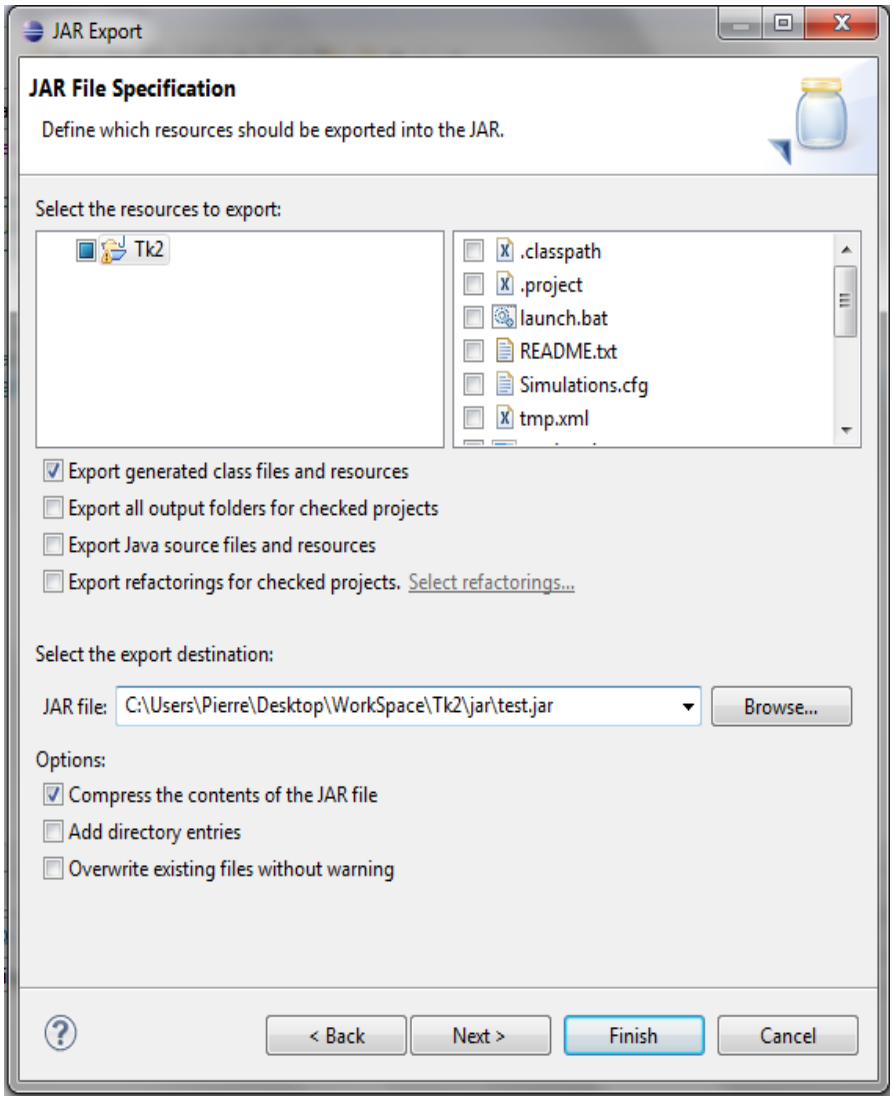
Enfin, Il faudra un fichier structurant l'ensemble des précédents, sous forme XML et nommé « config.xml ». La syntaxe doit obligatoirement respecter celle qui suit :

```
<?xml version="1.0" encoding="UTF-8"?>
<config team="Mac Gyver" color="E6E2AF">
	<icon name="gyver.png"/>
	<sound name="MacGyver.wav"/>
	<waragents>
		<warexplorer name="BrainExplorer.class"/>
		<warrocketlauncher name="BrainRocketLauncher.class"/>
		<warbase name="BrainBase.class"/>
	</waragents>
</config>
```

Une fois ces fichiers complétés, il vous suffit de générer le JAR à l'aide d'un IDE, puis de le placer dans le dossier jar présent à la racine du dossier Tk2. De cette façon, vous verrez votre équipe apparaître lors du lancement de WARBOT II. Une fois que le JAR est créé, il ne sera plus utile de le regénérer à chaque fois que vous effectuerez une modification dans votre code, cela sera détecté automatiquement.

X. Génération d'un JAR

Voici une méthode simple et rapide pour générer un JAR sous Eclipse. Clique droit sur votre package, puis export / Java / JAR File. Voici un petit screenshot pour les options à cocher.


JAR Export
JAR File Specification
Define which resources should be exported into the JAR.
Select the resources to export:
Tk2
.classpath
.project
launch.bat
README.txt
Simulations.cfg
tmp.xml
Export generated class files and resources
Export all output folders for checked projects
Export Java source files and resources
Export refactorings for checked projects. Select refactorings...
Select the export destination:
JAR file: C:\Users\Pierre\Desktop\WorkSpace\Tk2\jar\test.jar
Browse...
Options:
Compress the contents of the JAR file
Add directory entries
Overwrite existing files without warning
< Back
Next >
Finish
Cancel