Règles du jeu - Warbot

Version Processing 1.2

04/01/2024

SOMMAIRE

I Présentation générale

Contexte
Il s'agit d'un jeu inspiré du jeu Warbot proposé initialement par J. Ferber dans le cadre de la plate-forme Madkit (http://www.madkit.net/warbot/). Dans ce jeu, deux équipes de robots (rouges, verts) s'affrontent dans un environnement commun, et doivent accumuler davantage de ressources que les équipes adverses, soit en développant leurs propres ressources, soit en détruisant ou volant les ressources des adversaires. Les robots ont une conception figée (capacités, vitesse, perception, mémoire, etc.), à l'exception de leur comportement (ou cerveau), qui définit, à chaque itération de jeu, ce que fait chaque agent.

Objectif
Il s'agit donc d'imaginer et d'implémenter une stratégie décentralisée, ***se basant uniquement sur les perceptions et actions locales des agents*** (donc sans la possibilité de disposer de connaissances globales sur le terrain de jeu), pour mettre en œuvre un comportement coopératif entre les robots d'une équipe permettant de maximiser les ressources de l'équipe.

II Le monde de Warbot

II.1. Les entités du jeu

Le jeu se déroule avec deux équipes (rouge, verte), composées de robots de quatre types différents (*Bases*, *Explorers*, *Harvesters*, *RocketLaunchers*), dans un environnement torique peuplé également d'obstacles (*Walls*) et de ressources énergétiques (*Burgers*).

a) Les robots

Les agents-robots sont de 4 types :

Les **« explorateurs » (Explorers)** : les robots explorateurs sont à la fois rapides et avec un grand rayon de perception. Par ailleurs, ils peuvent récupérer et transporter la nourriture collectée par les moissonneurs ;

Les **« lanceurs de missiles » (RocketLaunchers)** : les robots lanceurs de missiles peuvent tirer sur un adversaire, mais leur rayon de perception, ainsi que leur vitesse est relativement faible ;

Les **« moissonneurs » (Harvesters)** : les moissonneurs sont encore plus lents et avec un rayon de perception très faible, mais ils sont les seuls à pouvoir collecter la nourriture dans l'environnement ou déplacer les murs ;

Les **« bases » (Bases)** : les bases sont fixes, mais ont des « super-capacités ». Elles peuvent créer des robots de tous types ou des missiles, et sont les seules à pouvoir convertir la nourriture en énergie. Par ailleurs, elles ont un grand rayon de perception, peuvent lancer des missiles et sont plus résistantes que les autres robots.

b) Les ressources

Les robots peuvent interagir par ailleurs avec 2 types d'objets, qui peuvent devenir des ressources, soit pour se protéger (*Walls*), soit pour gagner en énergie (*Burgers*) :

Les **« murs » (Walls) :** les murs sont des blocs que les agents moissonneurs peuvent déplacer. Ils bloquent les déplacement (un robot ne peut pas traverser un mur) et arrêtent les missiles (mais peuvent quand même être détruits au bout d'un certain nombre de tirs) ;

Les **« burgers » (Burgers)** : les burgers constituent le « carburant » des robots. Ils doivent pour cela être « récoltés » par les *moissonneurs*, soit à l'état « sauvage », soit en les mettant en culture, et ils doivent ensuite être transformés en énergie par les bases.

c) Les patches

L'environnement de simulation est constitué d'une grille de régions appelées « patches ». Ces patches gèrent la présence des murs, des robots, des burgers et des graines. A terme, ces patches pourraient être associés à la possibilité de déposer des phéromones, et auraient alors à gérer la dynamique de dépôt, de diffusion et d'évaporation de signaux chimiques.

III Le moteur de jeu

La gestion d'une simulation est effectuée par une instance de la classe `Simulation`. Le moteur de jeu consiste en une méthode `setup`, qui initialise le jeu et met en place les différentes entités du jeu, et d'une méthode `go`, qui calcule la dynamique du jeu, selon un principe analogue à celui des simulations à pas de temps discret. Le schéma général d'exécution d'un match est le suivant :

```
Méthode match
Début
  setup
  pour tick ← 1 à durée faire
    si non victoire de l'une des équipes alors
      go
      display
    fin_si
  fin_pour
Fin
```

III.1. Initialisation

La méthode `setup` de la classe `Simulation` initialise tous les éléments nécessaires au jeu, selon l'algorithme suivant :

```
Méthode setup
Début
  Création de l'environnement
  pour chaque couleur d'équipe color dans {Red, Green} faire
    Création de l'équipe color
    Création et ajout des bases color
    Initialisation des bases color
  fin_pour
  Création des Walls
  Création des Burgers
Fin
```

III.2. Boucle de simulation

La méthode `go` de la classe `Simulation` calcule ensuite les comportements de tous les robots et met à jour l'environnement de jeu, selon l'algorithme suivant :

```
Méthode go
Début
  Nettoyage des robots morts
  Activation de la dynamique des patches
  Activation des agents-robots en ordre aléatoire
  Guidage des missiles
  Gestion des graines et des burgers
  Mise à jour de l'énergie
Fin
```

Les agents-robots sont activés selon l'algorithme ci-dessous :

```
pour tous les Robots faire
  Décrémenter l'énergie (métabolisme du robot)
  Tester si le robot est toujours vivant (mort)
  Afficher au besoin un label sur le robot
  Appeler la procédure go
fin_pour
```

Les robots *RocketLaunchers* et *Base* doivent en outre respecter un délai d'attente entre deux tirs. Les robots *Base* doivent enfin convertir la nourriture récupérée en énergie.

III.3. Programmation d'une équipe

Le simulateur de jeu est structuré en plusieurs parties :

- le fichier principal **`Warbot_1_2_1.pde`** qui :
  - crée l'objet de simulation principal (`game`)
  - gère la progression du jeu et la mise à jour de l'affichage
  - gère les interactions clavier/souris
- le fichier **`Parameters.pde`** qui déclare tous les paramètres du jeu
- le fichier **`Message.pde`** qui contient la classe `Message` (qui sert à envoyer/recevoir des messages entre les agents)
- le fichier **`Patches.pde`** qui contient la classe `Patch` (qui sert à définir un découpage de l'environnement, utilisé pour les calculs de distance entre les robots)
- le fichier **`Simulation.pde`** qui contient la classe `Simulation` (qui sert à gérer le fonctionnement de la simulation = ordonnancement des agents, gestion de la physique de l'environnement, etc.)
- le fichier **`Turtles.pde`** qui contient les classes définissant les différents types de robots
- le fichier **`Ressources.pde`** qui contient les classes définissant les différents types de robots
- les fichiers **`Greens.pde`** et **`Reds.pde`** qui contiennent le code des équipes verte et rouge

Pour programmer l'IA d'une équipe, il faut

1. Ouvrir l'onglet `Greens.pde` ou `Reds.pde` selon l'équipe choisie
2. Compléter les classes `*Base`, `*Explorer`, `*Harvester` et `*RocketLauncher` (avec `*` à remplacer par la couleur de l'équipe), en implémentant pour chaque type de robot les méthodes `setup` (exécutée une seule fois à la création du robot) et `go` (exécutée à chaque tour de jeu).
3. Compléter la classe `*Team`, permettant de définir des types de massages personnalisés et de positionner les 2 bases de l'équipes à la position voulue dans la moitié de terrain de son équipe.

IV Les actions principales

Les sections ci-dessous décrivent le fonctionnement des principales actions offertes aux robots. Les variables et primitives sont décrites de manière rapide, une description plus détaillée étant fournie dans les annexes 1 à 3. Les primitives utilisables par les robots sont déclarées sous forme de méthodes dans les classes `Robot` et `Turtle`.

IV.1. Création de nouveaux robots 

La création de nouveaux robots est le propre des bases (`newExplorer`, `newRocketLauncher`, `newHarvester`). La création de robots n'est possible que si :

1. la base possède strictement plus d'énergie (`energy`) que le coût de fabrication du robot concerné (`*Cost`) ;
2. un patch est disponible autour de la base pour déposer le nouveau robot.

Le tableau 1 (voir annexe 1) fournit la quantité d'énergie nécessaire à la fabrication de chaque type de robot. Les robots sont créés avec les paramètres par défaut de leur type et en héritant certains paramètres de la base qui les a créés. Le robot est créé juste à côté de la base, dans une direction aléatoire.

Les procédures correspondantes sont (voir le détail à l'annexe 2) :

- `newExplorer` pour créer un *explorateur*
- `newRocketLauncher` pour créer un *lance-missile*
- `newHarvester` pour créer un *moissonneur*

IV.2. Perception 

Les robots ne peuvent utiliser pour leur perception que les primitives fournies (`perceive*`), qui permettent de limiter leur vision du monde à leur rayon de perception (`detectionRange`). La perception peut être au choix circulaire ou dans un cône orienté dans la direction du robot (`perceive*InCone`). La perception peut concerner la nourriture (`perceiveBurgers*`), les plantations (`perceiveSeeds*`), les murs (`perceiveWalls*`), les missiles guidés (`perceiveFafs*`) ou les autres robots (`perceiveRobots*`).

Les procédures correspondantes sont (voir le détail à l'annexe 2) :

- `perceiveBurgers` pour détecter les *Burgers* dans le rayon de perception du robot
- `perceiveBurgersInCone` pour détecter les *Burgers* dans un cône d'ouverture choisie
- `perceiveSeeds` pour détecter les graines dans le rayon de perception du robot
- `perceiveSeedsInCone` pour détecter les graines dans un cône d'ouverture choisie
- `perceiveWalls` pour détecter les murs dans le rayon de perception du robot
- `perceiveWallsInCone` pour détecter les murs dans un cône d'ouverture choisie
- `perceiveRobots` pour détecter les autres robots dans le rayon de perception du robot ; il est possible d'ajouter un paramètre de couleur et/ou un paramètre de type pour filtrer le type des robots à percevoir
- `perceiveRobotsInCone` pour détecter les robots (hors Bases) d'une couleur choisie dans un cône d'ouverture choisie ; il est possible de préciser la portée de perception choisie et d'ajouter un paramètre de couleur et/ou un paramètre de type pour filtrer le type des robots à percevoir

Les méthodes de perception renvoient une liste d'agents. Pour filtrer ces listes, vous pouvez utiliser les méthodes :

- `oneOf` qui renvoie l'un quelconque des agents de la liste
- `minDist` qui renvoie l'agent le plus proche

IV.3. Déplacement 

A l'exception des *Bases*, tous les agents peuvent s'orienter (`left`, `right`, `towards`) se déplacer (`distance`, `forward`, `backward`, `randomMove`) plus ou moins vite en fonction de leur type (au maximum à la vitesse `speed`), à condition que la voie soit dégagée (`freeAhead`), et que l'agent ne se soit pas déjà déplacé au cours

de l'itération en cours (`fdOK`). Quand un agent *Explorer* ou *RocketLauncher* se déplace sur des graines de *Burgers* (*Seeds*), il écrase les plants ce qui retarde leur croissance de 100 ticks. Quand la procédure `forward` ou `backward` est appelée alors qu'il y avait un obstacle devant (robot ou mur), une collision se produit, ce qui provoque des dommages à la fois sur le robot lui-même (son énergie est diminuée de `botCollisionDamage` points d'énergie) et sur l'obstacle percuté. Si un robot percute une base, il fait des dégâts à la base mais est détruit, quel que soit son niveau d'énergie.

Les procédures correspondantes sont (voir le détail à l'annexe 2) :

- `right` pour faire une rotation à droite d'un certain angle
- `left` pour faire une rotation à gauche d'un certain angle
- `distance` pour connaître la distance d'un autre agent
- `towards` pour connaître la direction pointant vers un autre agent
- `freeAhead` pour détecter les agents ou obstacles devant soi
- `forward` pour avancer devant soi à une certaine vitesse
- `backward` pour reculer à une certaine vitesse
- `randomMove` pour faire un mouvement aléatoire dans un cône d'ouverture 90°

IV.4. Récolte et conversion de nourriture en énergie

La nourriture (les *Burgers*) est récoltée par les *Harvesters* (`takeFood`), mais ceux-ci peuvent ensuite la transmettre à un autre *Harvester*, à un *Explorer* ou directement à la base (`giveFood`). La conversion de la nourriture en énergie s'effectue automatiquement au niveau de chaque base, à chaque tour de jeu (`foodToEnergy`). Plutôt que de donner les *Burgers* à une base, les *Harvesters* ont également la possibilité de les mettre en culture (`plantSeed`), ce qui permet de produire, de manière plus régulière, des *Burgers* plus énergétiques.

Les procédures correspondantes sont (voir le détail à l'annexe 2) :

- `takeFood` pour permettre à un *Harvester* de ramasser des *Burgers*
- `giveFood` pour donner de la nourriture à un autre robot (*Harvester* ou *Explorer*) ou à une *Base*
- `plantSeed` pour planter des *Burgers* afin de les récolter plus tard

IV.5. Communication entre robots

Les robots peuvent avoir différents types d'interaction. Certains peuvent s'échanger de la nourriture (cf. IV.4), mais l'essentiel de l'interaction entre les robots s'effectue sous la forme de communication par échanges de messages. Certains messages sont prédéfinis et correspondent à des demandes d'énergie (`ASK_FOR_ENERGY`) ou de de munitions (`ASK_FOR_BULLETS`) de la part des robots à la base, ou de transmission d'information sur la position de nourriture (`INFORM_ABOUT_FOOD`) ou d'une cible (`INFORM_ABOUT_TARGET` ou `INFORM_ABOUT_XYTARGET`).

Les procédures correspondantes sont (voir le détail à l'annexe 2) :

- `askForEnergy` pour demander de l'énergie à la base
- `askForBullets` pour demander des munitions à la base
- `sendMessage` pour envoyer un message personnalisé
- `flushMessages` pour vider la boîte de réception des messages

En réponse à des demandes d'énergie ou de munitions, seule la base peut transmettre de l'énergie (`giveEnergy`) ou des munitions (`giveBullets`) aux autres robots.

Les robots ont également la possibilité de définir de nouveaux messages personnalisés. Pour cela, il est nécessaire :

- de définir nouveau type de message dans la classe `*Team`
- de définir l'ordre et le nombre des arguments (le contenu d'un message est constitué d'un tableau de flottants)
- d'appeler la fonction `sendMessage` avec l'identifiant du destinataire, le type du message et la liste des arguments

IV.6. Tir 

Les *Bases* et les *RocketLaunchers* ont la possibilité de tirer deux types de missiles. Des missiles « classiques » (`launchBullet`) qui se déplacent en ligne droite à une certaine vitesse (`bulletSpeed`) jusqu'à atteindre un rayon de portée donné (`bulletRange`) ou jusqu'à frapper un obstacle, en infligeant des dégâts égaux à `bulletDamageToRobot` (ou `bulletDamageToBase` si l'obstacle est une base). Des missiles guidés de type « fire and forget » (`launchFaf`) qui s'orientent automatiquement en direction d'une cible donnée (avec les paramètres correspondants `faf*` à la place des paramètres `bullet*`). Il peut être nécessaire au préalable de fabriquer des missiles du bon type (`newBullets`, `newFafs`), avec un coût unitaire de fabrication `bulletCost` ou `fafCost`, et de les transférer au besoin aux rocketLauncher qui en font la demande (cf section précédente). Il est également nécessaire de respecter un temps d'attente entre le lancement de 2 missiles (`baseWaiting`, `launcherWaiting`).

Les procédures correspondantes sont (voir le détail à l'annexe 2) :

- `launchBullet` pour lancer un missile
- `launchFaf` pour lancer un missile de type « fire and forget »
- `newBullets` pour créer de nouvelles munitions (réservé aux bases)
- `newFafs` pour créer de nouveaux missiles de type « fire and forget » (réservé aux bases)

IV.7. Déplacement des blocs 

Les *Harvesters* ont la possibilité de déplacer les blocs (*Walls*). Ils peuvent en transporter jusqu'à cinq à la fois. Pour prendre un bloc (`takeWall`), il faut donc à la fois qu'ils soient à une distance inférieure à deux du bloc qu'ils souhaitent attraper, et qu'ils ne transportent pas déjà cinq bloc (`nbWalls`). Ils peuvent à tout moment déposer un bloc (`dropWall`) à condition qu'un patch soit libre pour le recevoir.

Les procédures correspondantes sont (voir le détail à l'annexe 2) :

- `takeWall` pour prendre un bloc
- `dropWall` pour déposer un bloc

V Contraintes

V.1. Programmation de l'équipe rouge

- L'équipe à compléter est l'équipe rouge, accessible dans l'onglet Reds (ou le fichier `Reds.pde`). Vous ne devez modifier aucun autre onglet ou fichier !
- Il faut pour cela compléter les méthodes `setup` et `go` des différents types de robots (Base, Explorer, Harvester, RocketLauncher), ainsi que la class RedTeam pour ajouter d'autres types de messages ou pour prépositionner vos bases au démarrage.

V.2. Fonctionnement en tournoi

Les différentes équipes programmées sont opposées au cours d'un tournoi. Pour cela, 2 équipes sont sélectionnées, et l'une d'entre elles est transformée en équipe verte. Pour cela, le fichier `Reds.pde` est recopié en `Greens.pde` en remplaçant systématiquement chaque occurrence de la chaîne de caractères « Red » par la chaîne de caractères « Green », transformant ainsi les classes `RedTeam` en `GreenTeam`, `RedBase` en `GreenBase`, etc. Il faut donc veiller à **respecter absolument** les règles suivantes :

- du fait que l'équipe rouge peut participer à un match du tournoi en tant qu'équipe verte, il ne faut pas faire d'hypothèses dans le code relatives à la couleur des robots. Pour faire référence à la couleur des robots « amis », utiliser la variable `colour` (c'est-à-dire la même couleur que soi), ou `friend`. Pour faire référence à la couleur des robots « ennemis », utiliser la variable `ennemy` (mise à jour automatiquement à la création des agents) ;

V.3. Ethique de jeu

L'intérêt du projet est de se focaliser sur les stratégies locales et décentralisées. Il est donc important de respecter un certain nombre de règles « éthiques » de manière à assurer l'équité entre les équipes. La règle générale, quand on programme le comportement d'un agent, est de toujours s'assurer d'une part que l'agent fait appel exclusivement à des informations locales dans son processus de décision, d'autre part que ses actions ont un effet uniquement local également. Pour cela, il est demandé de respecter les règles ci-dessous et, en cas de doute, de demander au préalable si tel ou tel comportement ou pratique est admise ou non.

a) Perception

- Chaque robot possède une distance de perception maximale (donnée par la variable `detectionRange`), et ne peut donc percevoir les robots (ni amis ni ennemis) au-delà de cette portée. Il faut utiliser pour cela les procédures `perceive*` mises à disposition dans les classes `Turtle` et `Robot` (voir également référence).
- Concernant toujours la perception, il n'est pas autorisé d'accéder à la mémoire des autres robots (et de manière générale à leurs informations : position, énergie, etc.) dans son rayon de perception, ni en consultation, ni a fortiori en modification. Si vous souhaitez échanger de l'information entre les robots, vous devez utiliser pour cela l'envoi de messages entre robots.
- Du fait de la perception uniquement locale des robots (y compris de la base), il n'est pas autorisé de compter les robots de son équipe pour savoir combien il en reste de chaque type, mais il faudra se baser sur des indices indirects liés aux agents que l'on a perçus.

b) Gestion de la mémoire

- Il est interdit de créer des variables globales (cela permettrait une communication globale entre les agents). De même, ne pas créer pas de variables d'agents (ce qui permettrait d'étendre la « mémoire » des agents). Utiliser en guise de mémoire des agents les variables `mem0` à `mem5` (`mem0` à `mem9` pour les bases). Chacune de ces variables peut contenir au maximum une liste de 3 valeurs (par exemple pour stocker les coordonnées d'un agent).
- Mémoriser un agent ou un identifiant d'agent (qu'il soit ami ou ennemi) pourrait permettre d'avoir des informations à son sujet ou d'agir sur lui, même s'il est hors de portée de perception. Une telle pratique est donc à manier avec précaution, sachant qu'il n'est pas autorisé d'accéder aux informations d'un agent dont on aurait mémorisé l'identifiant quand cet agent est sorti de sa sphère de perception.

c) Action

- Un seul déplacement est autorisé par itération de jeu. Pour cela, il faut utiliser les méthodes `forward` et `backward` mises à disposition celle-ci s'assure alors que l'agent qui essaye d'avancer ne s'est pas déjà déplacé au cours de l'itération en cours, qu'il ne se déplace pas plus vite que ses capacités lui permettent, et gère les problèmes de collision éventuels.
- De même, un seul tir de missile est autorisé par itération de jeu. Pour cela, il faut utiliser les méthodes `launchBullet` et `launchFaf` pour tirer des missiles.

Annexe 1 – Constantes caractéristiques des entités du jeu

Constantes caractéristiques des classes de robots

Nous énumérons ci-dessous les principales caractéristiques des différents types de robots (regroupées dans le tableau 1). Ces constantes sont déclarées dans le fichier `Parameters.pde`.

<breed>Speed

La vitesse maximale d'un agent de type <breed>

baseSpeed = vitesse maximale des Bases
explorerSpeed = vitesse maximale des Explorers
launcherSpeed = vitesse maximale des RocketLaunchers
harvesterSpeed = vitesse maximale des harvesters

<breed>Perception

Le rayon de perception d'un agent de type <breed>

basePerception = rayon de perception des Bases
explorerPerception = rayon de perception des Explorers
launcherPerception = rayon de perception des RocketLaunchers
harvesterPerception = rayon de perception des Harvesters

<breed>Cost

Le coût de création d'un agent de type <breed>

explorerCost = coût de création d'un robot Explorer
launcherCost = coût de création d'un robot RocketLauncher
harvesterCost = coût de création d'un robot Harvester

<breed>Nrj

La quantité d'énergie d'un robot « neuf »

baseNrj = énergie initiale des Bases
explorerNrj = énergie initiale des Explorers
launcherNrj = énergie initiale des RocketLaunchers
harvesterNrj = énergie initiale des Harvesters

<breed>Burgers

Le nombre de burgers relâchés dans l'environnement quand un robot est détruit

baseBurgers = burgers relâchés quand une Base est détruite
explorerBurgers = burgers relâchés quand un Explorer est détruit
launcherBurgers = burgers relâchés quand un RocketLauncher est détruit
harvesterBurgers = burgers relâchés quand unHarvester est détruit

<breed>Metabolism

Le métabolisme d'un robot, c'est-à-dire la quantité d'énergie consommée à chaque cycle

baseMetabolism = métabolisme des Bases
explorerMetabolism = métabolisme des Explorers
launcherMetabolism = métabolisme des RocketLaunchers
harvesterMetabolism = métabolisme des Harvesters

Le tableau ci-dessous récapitule les principales caractéristiques des agents.

Type de robot	Base	Explorer	Launcher	Harvester
Vitesse de déplacement (Speed)	0	1	0.5	0.25
Rayon de perception (Perception)	10	10	5	3
Coût de fabrication (Cost)	---	3 000	6 000	4 000
Energie initiale (Nrj)	50 000	1 000	4 000	2 000
Burgers relâchés en cas de destruction (Burgers)	20	2	2	2
Métabolisme (Metabolism)	1	0.1	0.1	0.1

Tableau 1 – Principales caractéristiques des différents types de robots

Constantes caractéristiques des « tireurs »

Les constantes énumérées ci-dessous ne concernent que les robots avec des capacités de tir (regroupées dans le tableau 2), à savoir les Bases et les RocketLaunchers. Ces constantes sont déclarées dans le fichier `Parameters.pde`.

<breed>NbBullets

Le nombre de munitions initialement détenues par le robot

baseNbBullets = le nombre de munitions d'une Base
launcherNbBullets = le nombre de munitions d'un RocketLauncher

<breed>MaxBullets

Le nombre maximum de munitions qu'un robot peut transporter

baseMaxBullets = le nombre maximum de munitions d'une Base
launcherMaxBullets = le nombre maximum de munitions d'un RocketLauncher

<breed>NbFafs

Le nombre de missiles « Fire and forget » initialement détenues par le robot

baseNbFafs = le nombre de missiles de type « fire and forget » d'une Base
launcherNbFafs = le nombre de missiles de type « fire and forget » d'un RocketLauncher

<breed>MaxFafs

Le nombre maximum de missiles de type « fire and forget » qu'un robot peut transporter

baseMaxFafs = le nombre maximum de missiles de type « fire and forget » d'une Base
launcherMaxFafs = le nombre maximum de missiles de type « fire and forget » d'un RocketLauncher

<breed>Waiting

Le délai d'attente entre deux tirs qu'un robot peut effectuer

baseWaiting = le délai d'attente entre deux tirs pour une Base
launcherWaiting = le délai d'attente entre deux tirs pour un RocketLauncher

Le tableau ci-dessous récapitule les principales caractéristiques des agents tireurs.

Type de robot	Base	Launcher
Nombre initial de munitions (NbBullets)	1000	1000
Nombre maximal de munitions (MaxBullets)	1000	1000
Nombre initial de « fire and forget » (nbFafs)	20	0
Nombre maximal de « fire and forget » (maxFafs)	100	0
Délai entre deux tirs (Waiting)	1	5

Tableau 2 – Principales caractéristiques des robots avec des capacités de tir

Les constantes énumérées ci-dessous concernent les munitions de type Bullet et Faf (regroupées dans le tableau 3). Ces constantes sont déclarées dans le fichier `Parameters.pde`.

<ammo>Cost

Le coût de fabrication du type de munition

bulletCost = le coût de fabrication d'une munition de type Bullet
fafCost = le coût de fabrication d'une munition de type Faf

<ammo>Speed

La vitesse de déplacement du type de munition

bulletSpeed = la vitesse de déplacement d'une munition de type Bullet
fafSpeed = la vitesse de déplacement d'une munition de type Faf

<ammo>Range

La portée du type de munition

bulletRange = la portée d'une munition de type Bullet
fafRange = la portée d'une munition de type Faf

<ammo>DamageToRobot

Les dégâts infligés par le type de munition sur les robots (hors Base)

bulletDamageToRobot = les dégâts infligés par une munition de type Bullet sur les robots hors Bases
fafDamageToRobot = les dégâts infligés par une munition de type Faf sur les robots hors Bases

<ammo>DamageToBase

Les dégâts infligés par le type de munition sur les bases

bulletDamageToBase = les dégâts infligés par une munition de type Bullet sur les Bases
fafDamageToBase = les dégâts infligés par une munition de type Faf sur les Bases

Le tableau ci-dessous récapitule les principales caractéristiques des munitions.

Type de munition	Bullet	Faf
Vitesse de déplacement (Speed)	1	1
Portée (Range)	10	20
Coût de fabrication (Cost)	1	50
Dommages sur les robots (DamageToRobot)	50	200
Dommages sur les bases (DamageToBase)	20	40

Tableau 3 – Principales caractéristiques des différents types de munitions

Constantes caractéristiques des burgers

Les constantes énumérées ci-dessous sont en relation avec les Burgers (regroupées dans le tableau 4). Ces constantes sont déclarées dans le fichier `Parameters.pde`.

maxSeeds

Le nombre maximal de graines de *Burgers* que l'on peut planter dans un patch

seedCost

Le coût pour planter une graine de *Burgers*, en unités de `carryingFood`

maturationTime

Le temps nécessaire, en nombre de ticks, pour qu'une graine de Burger arrive à maturité et produise un nouveau burger, dont l'énergie est comprise entre `domesticBurgerMinNrj` et `domesticBurgerMaxNrj`.

(voir aussi `maturationCrush`, `domesticBurgerMinNrj`, `domesticBurgerMaxNrj`)

maturationCrush

Le retard de croissance, en nombre de ticks, quand un burger domestique est piétiné

(voir aussi `maturationTime`, `domesticBurgerMinNrj`, `domesticBurgerMaxNrj`)

domesticBurgerMinNrj

L'énergie minimale d'un Burger « de culture »

(voir aussi `domesticBurgerMaxNrj`, `wildBurgerMinNrj`, `wildBurgerMaxNrj`)

domesticBurgerMaxNrj

L'énergie maximale d'un Burger « de culture »

(voir aussi `domesticBurgerMinNrj`, `wildBurgerMinNrj`, `wildBurgerMaxNrj`)

wildBurgerMinNrj

L'énergie minimale d'un Burger « sauvage »

(voir aussi `wildBurgerMaxNrj`, `domesticBurgerMinNrj`, `domesticBurgerMaxNrj`)

wildBurgerMaxNrj

L'énergie maximale d'un *Burger* « sauvage »

(voir aussi `wildBurgerMinNrj`, `domesticBurgerMinNrj`, `domesticBurgerMaxNrj`)

burgerPeriodicity

Avec une probabilité de 1 sur `burgerPeriodicity`, un gisement de `burgerQuantity` Burgers sauvages est produit

(voir aussi `wildBurgerMinNrj`, `wildBurgerMaxNrj`, `burgerQuantity`)

burgerQuantity

Avec une probabilité de 1 sur `burgerPeriodicity`, un gisement de `burgerQuantity` Burgers sauvages est produit

(voir aussi `wildBurgerMinNrj`, `wildBurgerMaxNrj`, `burgerPeriodicity`)

burgerDecay

A chaque tick, un burger mature perd une quantité d'énergie égale à `burgerDecay`.

Le tableau ci-dessous récapitule les principales constantes caractéristiques des *Burgers*.

Constante	Valeur
maxSeeds	5
seedCost	20
domesticBurgerMinNrj	100
domesticBurgerMaxNrj	150
Temps de maturation (maturationTime)	1 000
dégâts quand le plant est écrasé (maturationCrush)	100
wildBurgerMinNrj	50
wildBurgerMaxNrj	100
burgerPeriodicity	2 000
burgerQuantity	100
burgerDecay	0.1

Tableau 4 – Principales constantes caractéristiques liées aux burgers

Annexe 2 - Manuel de référence des procédures de robots

askForBullets

`askForBullets(Robot bob, float qty)`

`askForBullets(int bobID, float qty)`

Envoie un message de type `ASK_FOR_BULLETS` à la base `bob` (ou d'identifiant `bobID`) pour lui demander de transférer `qty` munitions de type Bullets au robot appelant.

(voir aussi `ASK_FOR_BULLETS`, `askForEnergy`, `giveEnergy`, `sendMessage`)

askForEnergy

`askForEnergy(Robot bob, float qty)`

`askForEnergy(int bobID, float qty)`

Envoie un message de type `ASK_FOR_ENERGY` à la base `bob` (ou d'identifiant `bobID`) pour lui demander de transférer `qty` unités d'énergie au robot appelant.

(voir aussi `ASK_FOR_ENERGY`, `askForBullets`, `giveEnergy`, `sendMessage`)

backward

`backward(float dist)`

Fonctionne de la même manière que `forward` mais le déplacement s'effectue vers l'arrière au lieu de s'effectuer vers l'avant.

(voir aussi `collisionDamage`, `fdOK`, `forward`, `randomMove`, `speed`)

distance

`distance(Turtle bob)`

`distance(PVector position)`

Renvoie la distance entre l'agent appelant et l'agent `bob` ou l'emplacement `position`.

(voir aussi `towards`)

dropWall

`dropWall()`

Si le robot *Harvester* transportait bien un bloc (`nbWalls` strictement positif), et si un patch est libre autour du robot, le dépose immédiatement dans le patch vide et décrémente `nbWalls`.

(voir aussi `takeWall`, `nbWalls`, `freePatch`)

flushMessages

`flushMessages()`

Vide la file de messages du robot.

(voir aussi `sendMessage`, `informAbout*`, `AskFor*`)

forward

`forward(float dist)`

Si l'agent ne s'est pas encore déplacé au cours de l'itération courante (`fdOK` vaut true), il inhibe son déplacement pour le restant du cycle (`fdOK` passe à false). S'il n'y a pas d'obstacle (robot ou mur) devant, alors l'agent avance d'une distance *dist* (ou *speed* si *dist* est supérieur à *speed*), sinon il entre en collision avec l'obstacle et perd une quantité d'énergie égale à `collisionDamage`, de même que l'obstacle percuté. Si des graines de burgers sont présentes dans le patch, le robot les écrase, sauf si c'est un Harvester.

(voir aussi `backward`, `collisionDamage`, `fdOK`, `randomMove`, `speed`)

freeAhead

`freeAhead(float dist)`

`freeAhead(float dist, float angle)`

Renvoie vrai si aucun agent n'est présent devant l'agent dans un cône d'ouverture `collisionAngle` (ou `angle` s'il est spécifié) et jusqu'à une distance `dist` (sans prendre en compte les burgers, les graines, les agents de perception, et les missiles).

(voir aussi `collisionAngle`, `fdOK`, `forward`, `randomMove`, `speed`)

freePatch

`freePatch()`

Essaye de trouver un patch libre autour de l'agent. Pour cela, la méthode choisit un angle aléatoire et teste si le patch dans cette direction à une distance de 1 est libre. Si c'est le cas, renvoie la position du patch, et sinon refait un nouvel essai avec un nouvel angle (10 tentatives au maximum).

(voir aussi `patchesInRadius`)

giveEnergy

`giveEnergy(int bobID, float nrj)`

Donne une quantité d'énergie `nrj` au robot d'identifiant `bobID`. Il faut pour cela que l'autre robot soit suffisamment proche (distance inférieure à 2) et que le donneur ait un niveau d'énergie (`energy`) au moins égal à `nrj`.

(voir aussi `energy`, `giveFood`)

giveBullet

`giveBullets(int bobID, float qty)`

Si l'agent d'identifiant `bobID` est à une distance inférieure ou égale à 2 de l'agent appelant, qu'il est de type *RocketLauncher*, et qu'il a la capacité de stocker les munitions demandées, et si la base a l'énergie suffisante, cette dernière transfère une quantité `qty` de munitions vers le robot `bobID`.

(voir aussi launchBullet, launcherMaxBullets)

giveFood

giveFood(Robot bob, float qty)

Si l'agent bob est à une distance inférieure ou égale à 2 de l'agent appelant, et qu'il est de type *Harvester*, *Explorer* ou *Base*, le *Harvester* ou l'*Explorer* qui fait l'appel transfère une quantité qty de nourriture vers le robot bob. Pour cela, la quantité qty ne doit pas être supérieure à la quantité transportée par l'agent appelant (carryingFood).

(voir aussi takeFood, carryingFood)

informAboutFood

informAboutFood(Robot bob, PVector position)

informAboutFood(int bobID, PVector position)

Envoie un message de type INFORM_ABOUT_FOOD au robot bob (ou d'identifiant bobID) pour lui indiquer la position de nourriture à l'emplacement position. Bob doit être à une distance inférieure à messageRange.

(voir aussi INFORM_ABOUT_FOOD, informAboutTarget, informAboutXYTarget, sendMessage)

informAboutTarget

informAboutTarget(Robot bob, Robot target)

informAboutTarget(int bobID, Robot target)

Envoie un message de type INFORM_ABOUT_TARGET au robot bob (ou d'identifiant bobID) pour lui indiquer l'identifiant target d'une cible. Bob doit être à une distance inférieure à messageRange.

(voir aussi INFORM_ABOUT_TARGET, informAboutFood, informAboutXYTarget, sendMessage)

informAboutXYTarget

informAboutXYTarget(Robot bob, PVector position)

informAboutXYTarget(int bobID, PVector position)

Envoie un message de type INFORM_ABOUT_XYTARGET au robot bob (ou d'identifiant bobID) pour lui indiquer l'emplacement position d'une cible. Bob doit être à une distance inférieure à messageRange.

(voir aussi INFORM_ABOUT_XYTARGET, informAboutFood, informAboutTarget, sendMessage)

launchBullet

launchBullet(float angle)

Lance une munition de type Bullet dans la direction angle, à condition d'une part d'avoir encore des missiles en réserve (nbBullets) et d'autre part de respecter un délai d'attente (*Waiting) entre deux tirs successifs. Il s'agit de munitions « standards » qui progressent en ligne droite jusqu'à toucher un obstacle ou atteindre la portée maximale (bulletRange).

(voir aussi launchFaf)

Envoie un nouveau missile dans la direction **dir**, à condition d'une part d'avoir encore des missiles en réserve (**nb-missiles**) et d'autre part de respecter un délai d'attente (**waiting**) entre deux tirs successifs. Il s'agit de missiles « standards » qui progressent en ligne droite jusqu'à toucher un obstacle ou atteindre la portée maximale (**my-range**).

(voir aussi base-waiting, faf-range, go-faf, go-missile, launch-faf, missile-range, my-range, new-faf, new-missile, rocket-launcher-waiting)

launchFaf

launchFaf(Robot bob)

Lance un missile de type « fire and forget » en direction du robot `bob`, à condition d'une part d'avoir encore des missiles de ce type en réserve (`nbFafs`) et d'autre part de respecter un délai d'attente (`baseWaiting`) entre deux tirs successifs.

(voir aussi `baseWaiting`, `fafRange`, `fafSpeed`, `launchBullet`, `newFaf`)

left

left(float angle)

Effectue une rotation sur la gauche d'un angle égal à angle (spécifié en degrés).

(voir aussi `right`)

minDist

minDist(ArrayList agentsSet)

Sélectionne celui des agents de la liste transmise en argument qui est le plus proche de l'agent appelant, et le renvoie.

(voir aussi `oneOf`)

new<breed>

newExplorer()

newHarvester()

newRocketLauncher()

La base ne doit pas avoir déjà fabriqué un robot au cycle courant, doit avoir l'énergie suffisante, et un patch libre doit être disponible autour de la base pour placer le nouveau robot. La base crée un nouveau robot, avec les paramètres par défaut présentés dans le tableau 1 (taille, vitesse, distance de perception, énergie initiale, nombre de burgers relâchés en cas de destruction), les caractères propres à la base qui les crée (couleur amie, couleur ennemie), la connaissance des bases de son équipe et une mémoire initialement vide.
Le nouveau robot est créé dans un patch libre adjacent à la base. L'énergie de la base est décrémentée de la quantité d'énergie associée à la création du robot (voir tableau 1)

(voir aussi `newBullets`, `newFafs`, `newWall`)

newBullets

newBullets(int qty)

Demande la création de `qty` nouvelles munitions de type Bullets. Cela n'est possible que si cela n'entraîne pas un dépassement du nombre maximal de missiles de ce type autorisé (`baseMaxBullets`). L'énergie de la base est décrémentée de la quantité d'énergie associée à la création d'un Bullet (`bulletCost`) multipliée par le nombre de munitions créées.

(voir aussi `launchFaf`, `launchBullet`, `baseMaxBullets`, `baseMaxFafs`, `newFafs`)

newFafs

`newFafs(int qty)`

Demande la création de `qty` nouvelles munitions de type Fafs. Cela n'est possible que si cela n'entraîne pas un dépassement du nombre maximal de missiles de ce type autorisé (`baseMaxFafs`). L'énergie de la base est décrémentée de la quantité d'énergie associée à la création d'un Faf (Faf`Cost`) multipliée par le nombre de munitions créées.

(voir aussi `launchFaf`, `launchBullet`, `baseMaxBullets`, `baseMaxFafs`, `newBullets`)

newWall

`newWall`

Crée un nouveau mur. L'énergie de la base est décrémentée de la quantité d'énergie associée à la création d'un mur. La base ne peut pas à la fois créer de nouveaux robots et de nouveaux murs au même cycle.

(voir aussi `newBullets`, `newFafs`, `new<breed>`)

oneOf

`oneOf(ArrayList agentsSet)`

Sélectionne aléatoirement l'un des agents de la liste transmise en argument, et le renvoie.

(voir aussi `minDist`)

patchesInRadius

`patchesInRadius()`

Renvoie la liste des patchs libre autour de l'agent dans un rayon égal à la distance de perception de l'agent.

(voir aussi `freePatch`)

perceiveBurgers

`perceiveBurgers()`

Renvoie la liste des agents de type `Burger` dans le rayon de perception du robot (`detectionRange`).

(voir aussi `perceiveBurgersInCone`, `perceive*`)

perceiveBurgersInCone

`perceiveBurgersInCone(float angle)`

Renvoie la liste des agents de type `Burger` dans un cône d'ouverture `angle` et dans le rayon de perception du robot (`detectionRange`).

(voir aussi `perceiveBurgers`, `perceive*`)

perceiveFafs

`perceiveFafs()`

Renvoie la liste des agents de type `Faf` (missiles guidés de type « Fire and forget ») dans le rayon de perception du robot (`detectionRange`).

(voir aussi `perceiveFafsInCone`, `perceive*`)

perceiveFafsInCone

`perceiveFafsInCone(float angle)`

Renvoie la liste des agents de type `Faf` dans un cône d'ouverture `angle` et dans le rayon de perception du robot (`detectionRange`).

(voir aussi `perceiveFafs`, `perceive*`)

perceiveRobots

`perceiveRobots()`

`perceiveRobots(color clr)`

`perceiveRobots(color clr, int breed)`

Renvoie la liste des agents de type *Explorer*, *Harvester* ou *RocketLauncher* dans le rayon de perception du robot (`detectionRange`). Lorsque la couleur `clr` est précisée, ne renvoie que les robots de la couleur demandée. Lorsque `breed` est précisé, ne renvoie que les robots du type demandé.

(voir aussi `detectionRange`, `ennemy`, `friend`, `perceiveRobotsInCone`, `perceive*`)

perceiveRobotsInCone

`perceiveRobotsInCone(float angle)`

`perceiveRobotsInCone(float angle, color clr)`

`perceiveRobotsInCone(float angle, color clr, int breed)`

`perceiveRobotsInCone(float angle, float dist)`

`perceiveRobotsInCone(float angle, float dist, color clr)`

`perceiveRobotsInCone(float angle, float dist, color clr, int breed)`

Renvoie la liste des agents de type *Explorer*, *Harvester* ou *RocketLauncher* dans un cône d'ouverture `angle` et dans le rayon de perception du robot (`detectionRange`) ou jusqu'à une distance `dist`. Lorsque la couleur `clr` est précisée, ne renvoie que les robots de la couleur demandée. Lorsque `breed` est précisé, ne renvoie que les robots du type demandé.

(voir aussi `detectionRange`, `ennemy`, `friend`, `perceiveRobots`, `perceive*`)

perceiveSeeds

`perceiveSeeds(color clr)`

Renvoie les agents de type `Seed` et de couleur `clr` dans le rayon de perception du robot (`detectionRange`). Les graines « ennemies » (`ennemy`) ne sont perceptibles que quand elles ont un âge > 500, alors que toutes les graines amies (`friend`) sont perceptibles.

(voir aussi `detectionRange`, `ennemy`, `friend`, `perceiveSeedsInCone`, `perceive*`)

perceiveSeedsInCone

`perceiveSeedsInCone(float angle, color clr)`

Renvoie les agents de type `Seed` et de couleur `clr` dans un cône d'ouverture `angle` et dans le rayon de perception du robot (`detectionRange`). Les graines « ennemies » (`ennemy`) ne sont perceptibles que quand elles ont un âge > 500, alors que toutes les graines amies (`friend`) sont perceptibles.

(voir aussi `detectionRange`, `ennemy`, `friend`, `perceiveSeeds`, `perceive*`)

perceiveWalls

`perceiveWalls()`

Renvoie la liste des agents de type `Wall` dans le rayon de perception du robot (`detectionRange`).

(voir aussi `perceiveWallsInCone`, `perceive*`)

perceiveWallsInCone

`perceiveWallsInCone(float angle)`

Renvoie la liste des agents de type `Wall` dans un cône d'ouverture `angle` et dans le rayon de perception du robot (`detectionRange`).

(voir aussi `perceiveWalls`, `perceive*`)

plantSeed

`plantSeed`

Plante une graine de *Burger* dans le patch courant. Le nombre maximum de graines dans un patch est donné par `maxSeeds`, chaque graine coûte `seedCost` unités de `carryingFood` à planter. Au bout d'un temps égal à `maturationTime`, chaque graine produira un Burger dont l'énergie est comprise entre `domesticBurgerMinNrj` et `domesticBurgerMaxNrj`. Si le plan de burger est écrasé par un robot autre qu'un *Harvester*, l'âge de maturation est diminué de 100.

(voir aussi `maxSeed`, `seedCost`, `maturationTime`, `domesticBurgerMinNrj`, `domesticBurgerMaxNrj`)

randomMove

`randomMove(float angle)`

Choisit une orientation aléatoire dans un cône d'ouverture `angle` par rapport à la direction courante puis appelle la méthode `forward` pour avancer.

(voir aussi backward, collisionDamage, fdOK, forward, freeAhead, speed)

recycle

recycle(Robot bob)

Recycle le robot bob en récupérant son énergie augmentée de 1000 puis le supprime.

(voir aussi nbw<breed>, newBullets, newFafs, newWall)

right

right(float angle)

Effectue une rotation sur la droite d'un angle égal à angle (spécifié en degrés).

(voir aussi left)

sendMessage

sendMessage(Robot bob, int type, float[] args)

sendMessage(int bobID, int type, float[] args)

Envoie un message personnalisé de type type au robot bob (ou d'identifiant bobID), avec la liste d'arguments args. Le type de message personnalisé doit être déclaré dans la classe *Team.

(voir aussi flushMessages, informAbout*, AskFor*)

takeFood

takeFood(Burger zorg)

Si le burger zorg est à une distance inférieure ou égale à 2, le *Harvester* augmente sa quantité de nourriture transportée (carryingFood) de la valeur d'énergie (energy) du *Burger*. Le *Burger* est ensuite supprimé.

(voir aussi giveFood, carryingFood, energy)

takeWall

takeWall(Wall bloc)

Si bloc est bien un mur et qu'il est à une distance inférieure ou égale à 2, et si le *Harvester* transporte un nombre de blocs strictement inférieur à harvesterWallCapacity, ramasse le bloc et l'enlève de l'environnement.

(voir aussi dropFood, nbWalls, harvesterWallCapacity)

towards

towards(Turtle bob)

towards(PVector position)

Renvoie l'angle pour pointer de l'agent appelant vers l'agent bob ou l'emplacement position.

(voir aussi distance)

Annexe 3 - Manuel de référence des procédures de gestion du jeu

compute-energy
(à compléter)

convert-food-into-energy
(à compléter)

display-label
(à compléter)

go-faf
(à compléter)

go-missile
(à compléter)

go-perception
(à compléter)

grow-seed
(à compléter)

init-burgers
(à compléter)

mort
(à compléter)

new-base
(à compléter)

new-burgers
(à compléter)

new-random-burgers
(à compléter)

new-walls
(à compléter)

update_energy_watches
(à compléter)

Annexe 4 – Interface graphique