

Règles du jeu - Warbot

Version 3.0.2

19/02/2019

SOMMAIRE

I	Présentation générale.....	2
II	Le monde de Warbot	2
II.1.	Les entités du jeu	2
a)	Les robots	2
b)	Les ressources	2
III	Le moteur de jeu	3
III.1.	Initialisation.....	3
III.2.	Boucle de simulation	3
III.3.	Programmation d'une équipe	4
IV	Les actions principales	5
IV.1.	Création de nouveaux robots	5
IV.2.	Perception	5
IV.3.	Déplacement	6
IV.4.	Récolte et conversion de nourriture en énergie	6
IV.5.	Interaction entre robots	6
IV.6.	Tir	6
IV.7.	Déplacement des blocs	7
V	Contraintes	7
V.1.	Programmation de l'équipe rouge.....	7
V.2.	Fonctionnement en tournoi.....	7
V.3.	Ethique de jeu	8
a)	Perception	8
b)	Gestion de la mémoire.....	8
c)	Action	8
	Constantes caractéristiques des classes de robots.....	9
	Constantes caractéristiques des burgers	10

I Présentation générale

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, trois équipes de robots (rouges, verts, bleus) 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. 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 trois équipes (rouge, verte et bleue), 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 déplacements (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.

III Le moteur de jeu

Comme la plupart des simulations dans Netlogo, le moteur de jeu consiste en une procédure **setup**, qui initialise le jeu et met en place les différentes entités du jeu, et d'une procédure **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 :

```
Procédure match
Début
  setup
  pour i ← 1 à duree faire
    si non victoire de l'une des équipes alors
      go
    fin_si
  fin_pour
fin
```

III.1. Initialisation

La procédure **setup** initialise tous les éléments nécessaires au jeu, selon l'algorithme suivant :

```
Procédure setup
Début
  Initialiser les constantes de jeu
  Créer les bases de chaque couleur
  pour chaque base de couleur Color faire
    Initialiser la variable my-bases
    Appeler InitColorBase (Color à remplacer par Red, Green ou Blue)
  fin_pour
  Création des Walls
  Création des Burgers
  Calcule le guidage des Missiles
fin
```

III.2. Boucle de simulation

La procédure **go** calcule ensuite les comportements de tous les robots et met à jour l'environnement de jeu, selon l'algorithme suivant :

```
Procédure go
Début
  Réinitialise fd-ok? à true (autorise le déplacement des agents)
  Détermine si une équipe a gagné (plus de bases des autres couleurs)
  Active les Explorers de chaque équipe
  Active les RocketLaunchers de chaque équipe
  Active les Harvesters de chaque équipe
  Active les Bases de chaque équipe
  Calcule le guidage des Missiles
  Calcule le guidage des Fafs
  Fait pousser les graines de Burgers
  Affiche au besoin les rayons de perception
  Crée aléatoirement de nouveaux Burgers sauvages
  Met à jour l'affichage de l'énergie des différentes équipes
  Teste l'arrêt du jeu (nombre fixe d'itérations)
Fin
```

Les *Explorers* sont activés selon l'algorithme ci-dessous :

Pour tous les *Explorers* faire

- Décrémenter l'énergie (métabolisme du robot)
- Tester si le robot est toujours vivant (**mort**)
- Afficher au besoin un label sur le robot
- Appeler goColorExplorer (Color à remplacer par Red, Green ou Blue)

Fin_pour

Les *Harvesters* sont activés selon l'algorithme ci-dessous :

Pour tous les *Harvesters* faire

- Décrémenter l'énergie (métabolisme du robot)
- Tester si le robot est toujours vivant (**mort**)
- Afficher au besoin un label sur le robot
- Appeler goColorHarvester (Color à remplacer par Red, Green ou Blue)

Fin_pour

Les *RocketLaunchers* sont activés selon l'algorithme ci-dessous :

Pour tous les *RocketLaunchers* faire

- Décrémenter l'énergie (métabolisme du robot)
- Tester si le robot est toujours vivant (**mort**)
- Décrémenter le délai d'attente avant de pouvoir tirer un nouveau missile
- Afficher au besoin un label sur le robot
- Appeler goColorRocketLauncher (Color à remplacer par Red, Green ou Blue)

Fin_pour

Les *Bases* sont activées selon l'algorithme ci-dessous :

Pour toutes les *Bases* faire

- Tester si la base est toujours vivante (**mort**)
- Décrémenter le délai d'attente avant de pouvoir tirer un nouveau missile
- Convertir la nourriture récupérée en énergie
- Afficher au besoin un label sur le robot
- Appeler goColorBase (Color à remplacer par Red, Green ou Blue)

Fin_pour

III.3. Programmation d'une équipe

Le simulateur de jeu est structuré en deux parties :

- le fichier principal **warbot.nlogo** qui contient
 - o le moteur de jeu (setup, go, procédures de gestion du jeu, voir annexe 3)
 - o les déclarations des constantes et variables globales (voir annexe 1)
 - o la définition de la structure « physique » des robots (voir annexe 1)
 - o les procédures utilisateurs (voir annexe 2)
- les fichiers d'extension :
 - o blues.nls : l'IA de l'équipe bleu
 - o greens.nls : l'IA de l'équipe verte
 - o reds.nls : l'IA de l'équipe rouge

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

1. Aller dans l'onglet Code puis sélectionner l'équipe souhaitée dans « Included Files »
2. Compléter les procédures **initColor*** (Color à remplacer par Red, Green ou Blue). Pour l'équipe rouge : **initRedBase, initRedExplorer, initRedRocketLauncher, initRedBase**
3. Compléter les procédures **goColor*** (Color à remplacer par Red, Green ou Blue). Pour l'équipe rouge : **goRedBase, goRedExplorer, goRedRocketLauncher, goRedBase**

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.

IV.1. Création de nouveaux robots

La création de nouveaux robots est le propre des bases (**new-Explorer**, **new-RocketLauncher**, **new-Harvester**). La création de robots n'est possible que si la base possède suffisamment d'énergie (**energy**). Le tableau 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 :

- **new-Explorer** pour créer un ou plusieurs *explorateurs*
- **new-RocketLauncher** pour créer un ou plusieurs *lance-missiles*
- **new-Harvester** pour créer un ou plusieurs *moissonneurs*

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 (**detection-range**). La perception peut être au choix circulaire ou dans un cône orienté dans la direction du robot (**perceive-*-in-cone**). La perception peut concerner la nourriture (**perceive-food***), les plantations (**perceive-seeds***), les murs (**perceive-walls***), ou les autres robots (**perceive-robot***, **perceive-specific-robot***, **perceive-base***).

Les procédures correspondantes sont :

- **perceive-food** pour détecter les *Burgers* dans le rayon de perception du robot
- **perceive-food-in-cone** pour détecter les *Burgers* dans un cône d'ouverture choisie
- **perceive-seeds** pour détecter les graines dans le rayon de perception du robot
- **perceive-seeds-in-cone** pour détecter les graines dans un cône d'ouverture choisie
- **perceive-walls** pour détecter les murs dans le rayon de perception du robot
- **perceive-walls-in-cone** pour détecter les murs dans un cône d'ouverture choisie
- **perceive-robots** pour détecter les robots (*hors Bases*) d'une couleur choisie dans le rayon de perception du robot
- **perceive-robots-in-cone** pour détecter les robots (*hors Bases*) d'une couleur choisie dans un cône d'ouverture choisie
- **perceive-robots2** pour détecter tous les robots (*hors Bases*) dans le rayon de perception du robot
- **perceive-robots2-in-cone** pour détecter tous les robots (*hors Bases*) dans un cône d'ouverture choisie
- **perceive-specific-robots** pour détecter les robots d'une couleur et d'un type choisis dans le rayon de perception du robot
- **perceive-specific-robots-in-cone** pour détecter les robots d'une couleur et d'un type choisis dans un cône d'ouverture choisie
- **perceive-specific-robots2** pour détecter tous les robots d'un type choisis dans le rayon de perception du robot
- **perceive-specific-robots2-in-cone** pour détecter tous les robots d'un type choisis dans un cône d'ouverture choisie
- **perceive-base** pour détecter les Bases d'une couleur choisie dans le rayon de perception du robot
- **perceive-base-in-cone** pour détecter les Bases d'une couleur choisie dans un cône d'ouverture choisie
- **perceive-base2** pour détecter les Bases dans le rayon de perception du robot
- **perceive-base2-in-cone** pour détecter les Bases dans un cône d'ouverture choisie

to-report

IV.3. Déplacement

A l'exception des *Bases*, tous les agents peuvent se déplacer (**forward-move**), plus ou moins vite en fonction de leur type (au maximum à la vitesse **speed**), à condition que la voie soit dégagée (**free-ahead?**), et que l'agent ne se soit pas déjà déplacé au cours de l'itération en cours (**fd-ok?**). 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-move** est appelée alors qu'il y avait un obstacle devant (robot ou mur), une collision se produit, ce qui coûte **collision-damage** points d'énergie (**energy**) à la fois au robot et à l'obstacle percuté. Si un robot percute une base, il fait des dégâts à la base mais est détruit, quelque soit son niveau d'énergie.

Les procédures correspondantes sont :

- **free-ahead?** pour détecter les agents ou obstacles devant soi
- **forward-move** pour avancer devant soi à une certaine vitesse
- **random-move** 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* (**take-food**), mais ceux-ci peuvent ensuite la transmettre à un autre *Harvester*, à un *Explorer* ou directement à la base (**give-food**). La conversion de la nourriture en énergie s'effectue automatiquement au niveau de chaque base, à chaque tour de jeu (**convert-food-into-energy**). Plutôt que de donner les *Burgers* à une base, les *Harvesters* ont également la possibilité de les mettre en culture (**plant-seeds**), ce qui permet de produire, de manière plus régulière, des *Burgers* plus énergétiques.

Les procédures correspondantes sont :

- **take-food** pour permettre à un *Harvester* de ramasser des *Burgers*
- **give-food** pour donner de la nourriture à un autre robot (*Harvester* ou *Explorer*) ou à une *Base*
- **plant-seeds** pour planter des *Burgers* afin de les récolter plus tard

IV.5. Interaction entre robots

Les robots peuvent avoir différents types d'interaction. Certains peuvent s'échanger de la nourriture (cf. III.4). Seule la base peut transmettre de l'énergie (**give-energy**). Ils peuvent enfin transmettre de l'information en modifiant la mémoire d'un autre robot à proximité (mem0 à mem5).

Les procédures correspondantes sont :

- **give-energy** pour donner de l'énergie à un autre robot

IV.6. Tir

Les *Bases* et les *RocketLaunchers* ont la possibilité de tirer deux types de missiles. Des missiles « classiques » (**launch-rocket**) qui se déplacent en ligne droite à une certaine vitesse (**missile-speed**) jusqu'à atteindre un rayon de portée donné (**missile-range**) ou jusqu'à frapper un obstacle, en infligeant des dégâts égaux à **missile-robot-damage** (ou **missile-base-damage** si l'obstacle est une base). Des missiles guidés de type « fire and forget » (**launch-faf**) qui s'orientent automatiquement en direction d'une cible donnée (avec les paramètres correspondant **faf-** à la place des paramètres **missile-**). Il peut être nécessaire de recharger au préalable le robot en missiles du bon type (**new-missiles**, **new-fafs**), avec un coût unitaire de fabrication de **missile-cost**. Il est également nécessaire de respecter un temps d'attente entre le lancement de 2 missiles (**base-waiting**, **rocket-launcher-waiting**).

Les procédures correspondantes sont :

- **launch-rocket** pour lancer un missile
- **launch-faf** pour lancer un missile de type « fire and forget »
- **new-missiles** pour créer de nouveaux missiles
- **new-fafs** pour créer de nouveaux missiles de type « fire and forget »

IV.7. Déplacement des blocs

Les *Harvesters* ont la possibilité de déplacer les blocs (*Walls*). Ils ne peuvent en transporter qu'un seul à la fois. Pour prendre un bloc (**take-wall**), il faut donc à la fois qu'ils soient à une distance inférieure à 2 du bloc qu'ils souhaitent attraper, et qu'ils ne transportent pas déjà un bloc (**carrying-wall?**). Ils peuvent à tout moment déposer le bloc (**drop-wall**).

Les procédures correspondantes sont :

- **take-wall** pour prendre un bloc
- **drop-wall** pour le déposer

V Contraintes

V.1. Programmation de l'équipe rouge

- Le projet est structuré en 5 fichiers distincts :
 - le fichier **warbot.nlogo** contient le code de base définissant la structure des robots, les procédures de gestion du jeu, et un certain nombre de procédures à disposition des robots pour la programmation de leur comportement ;
 - le fichier **parameters.nls** contenant les valeurs initiales des paramètres ;
 - le fichier **reds.nls** contenant le comportement des robots de l'équipe rouge ;
 - le fichier **greens.nls** contenant le comportement des robots de l'équipe verte ;
 - le fichier **blues.nls** contenant le comportement des robots de l'équipe bleue ;
- L'équipe à compléter est l'équipe rouge, accessible dans le fichier « reds.nls » : dans l'onglet « Code », cliquer sur le bouton « Included Files » puis sur « reds.nls »
- Il faut pour cela compléter les procédures suivantes :
 - **initRedExplorer** : initialisation des explorateurs rouges
 - **initRedRocketLauncher** : initialisation des lanceurs de missiles rouges
 - **initRedHarvester** : initialisation des moissonneurs rouges
 - **initRedBase** : initialisation des bases rouges
 - **goRedExplorer** : activation des explorateurs rouges
 - **goRedRocketLauncher** : activation des lanceurs de missiles rouges
 - **goRedHarvester** : activation des moissonneurs rouges
 - **goRedBase** : activation des bases rouges

V.2. Fonctionnement en tournoi

Les différentes équipes programmées sont opposées au cours d'un tournoi. Pour cela, 3 équipes sont sélectionnées, et deux d'entre elles sont transformées en équipes verte et bleue. Pour cela, les en-têtes des procédures **initRed*** sont transformées en **initGreen*** ou **initBlue*** (de même pour **goRed***). Il faut donc veiller à **respecter absolument** les règles suivantes :

- ne pas modifier les lignes d'en-tête des procédures **initRed*** et **goRed***, sinon le logiciel de tournoi risque de ne pas les identifier correctement ;
- du fait que l'équipe rouge peut participer à un match du tournoi en tant qu'équipe verte ou bleue, 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 **color** (c'est-à-dire la même couleur que soi), ou **friend**. Pour faire référence à la couleur des robots « ennemis », utiliser les variables **enemy1** et **enemy2** (mises à jour automatiquement à la création des agents) ;
- du fait que Netlogo n'autorise pas plusieurs procédures ou fonctions avec le même nom, il est nécessaire :
 - de choisir un nom d'équipe unique ;
 - de préfixer toutes les procédures et fonctions de l'équipe rouge avec ce nom d'équipe.

- du fait que votre équipe ne sera pas toujours opposée aux équipes greens.nls et blues.nls, vous devez faire attention à ne pas utiliser dans le code de votre équipe, des appels aux procédures des autres équipes !

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 `detection-range`), et ne peut donc percevoir les robots (ni amis ni ennemis) au-delà de cette portée. Il faut utiliser pour cela les procédures **perceive-xxx** mises à disposition dans l'onglet code (voir également référence).
- Concernant toujours la perception, il est autorisé d'accéder à la mémoire des autres robots (et de manière générale à leurs informations : position, énergie, etc.) dans son rayon de perception. Il est de même autorisé de modifier la mémoire des robots de son équipe situés dans son rayon de perception (ce qui équivaut à envoyer un message à un autre robot). En revanche, il n'est pas autorisé de modifier ni la mémoire des robots adverses, ni les autres informations les concernant.
- 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 perçu. Par exemple, les burgers créés lors de la mort d'un robot portent la trace de l'espèce (**burger-breed**) et de l'identifiant (**who-id**) du robot mort.

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**. Chacune de ces variables peut contenir au maximum une liste de 2 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 ne faut pas appeler directement la procédure **forward** (ou **fd**) pour avancer mais utiliser à la place la procédure **forward-move** : 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 la procédure **launch-rocket** pour lancer un missile.

Annexe 1 - Caractéristiques des entités du jeu

Constantes caractéristiques des classes de robots

speed

<breed>-speed

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

base-speed = vitesse maximale des Bases

explorer-speed = vitesse maximale des Explorers

rocket-launcher-speed = vitesse maximale des RocketLaunchers

harvester-speed = vitesse maximale des harvesters

perception

<breed>-perception

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

base-perception = rayon de perception des Bases

explorer-perception = rayon de perception des Explorers

rocket-launcher-perception = rayon de perception des RocketLaunchers

harvester-perception = rayon de perception des harvesters

cost

<breed>-cost

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

explorer-cost = rayon de perception des Explorers

rocket-launcher-cost = rayon de perception des RocketLaunchers

harvester-cost = rayon de perception des harvesters

nrj

<breed>-nrj

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

base-nrj = énergie initiale des Bases

explorer-nrj = énergie initiale des Explorers

rocket-launcher-nrj = énergie initiale des RocketLaunchers

harvester-nrj = énergie initiale des harvesters

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

Type de robot	Base	Explorer	RocketLauncher	Harvester
Vitesse de déplacement (speed)	0	1	0.5	0.25
Rayon de perception (perception)	10	10	5	3
Coût de fabrication (cost)	---	2 000	6 000	3 000
Energie initiale (nrj)	50 000	1 000	4 000	2 000
Burgers relâchés en cas de destruction	100	5	10	5
Métabolisme (metabolism)	---	0.01	0.01	0.01
Nombre initial de missiles	100	0	30	0
Nombre initial de « fire and forget »	20	0	2	0

Tableau 1 – Principales caractéristiques des différents types de robots

Constantes caractéristiques des burgers

max-seeds

max-seeds

Le nombre maximal de graines de *Burgers* que l'on peut planter dans un patch (voir tableau 2)

seed-cost

seed-cost

Le coût pour planter une graine de *Burgers*, en unités de *carrying-food?* (voir tableau 2)

maturation-time

maturation-time

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 *seeded-burger-min-nrj* et *seeded-burger-max-nrj*.

(voir aussi [seeded-burger-min-nrj](#), [seeded-burger-max-nrj](#))

seeded-burger-min-nrj

seeded-burger-min-nrj

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

(voir aussi [seeded-burger-max-nrj](#), [wild-burger-min-nrj](#), [wild-burger-max-nrj](#), [grow-seed](#))

seeded-burger-max-nrj

seeded-burger-max-nrj

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

(voir aussi [seeded-burger-min-nrj](#), [wild-burger-min-nrj](#), [wild-burger-max-nrj](#), [grow-seed](#))

wild-burger-min-nrj

wild-burger-min-nrj

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

(voir aussi [wild-burger-max-nrj](#), [seeded-burger-min-nrj](#), [seeded-burger-max-nrj](#), [new-random-burgers](#))

wild-burger-max-nrj

wild-burger-max-nrj

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

(voir aussi [wild-burger-min-nrj](#), [seeded-burger-min-nrj](#), [seeded-burger-max-nrj](#), [new-random-burgers](#))

burger-periodicity

burger-periodicity

Avec une probabilité de 1 sur *burger-periodicity*, un gisement de *burger-quantity* Burgers sauvages est produit

(voir aussi [wild-burger-min-nrj](#), [wild-burger-max-nrj](#), [burger-quantity](#), [new-random-burgers](#))

burger-quantity

burger-quantity

Avec une probabilité de 1 sur *burger-periodicity*, un gisement de *burger-quantity* Burgers sauvages est produit

(voir aussi [wild-burger-min-nrj](#), [wild-burger-max-nrj](#), [burger-periodicity](#), [new-random-burgers](#))

burger-decay

burger-decay

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

(voir aussi [wild-burger-min-nrj](#), [wild-burger-max-nrj](#), [burger-periodicity](#), [new-random-burgers](#))

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

Constante	Valeur
max-seeds	5
seed-cost	20
dégâts quand le plant est écrasé	100
seeded-burger-min-nrj	100
seeded-burger-max-nrj	150
maturation-time	1 000
wild-burger-min-nrj	50
wild-burger-max-nrj	100
burger-periodicity	2 000
burger-quantity	100
burger-decay	0.1

Tableau 2 – Principales constantes caractéristiques liées aux burgers

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

Type de missile	missile	faf
Vitesse de déplacement (speed)	1	1
Portée (range)	10	20
Coût de fabrication (cost)	10	50
Dommages sur les robots (robot-damage)	100	200
Stock initial des rocket-launchers (rocket-launcher-nb-missiles)	100	2
Capacité maximum des rocket-launchers (rocket-launcher-max-missiles)	1 000	5
Stock initial des bases (base-nb-missiles)	1 000	20
Capacité maximum des bases (base-launcher-max-missiles)	1 000 000	100
Temps d'attente des rocket-launchers (rocket-launcher-waiting)	5	5
Temps d'attente des bases (base-waiting)	1	1
Dommages sur les bases (robot-damage)	20	40

Tableau 3 – Principales caractéristiques des différents types de missiles

Annexe 2 - Manuel de référence des procédures de robots

drop-wall

drop-wall



Si le robot *Harvester* transportait bien un bloc (**carrying-wall?** vaut true), le dépose immédiatement devant le robot et réinitialise **carrying-wall?** à false.

(voir aussi [take-wall](#), [carrying-wall?](#))

forward-move

forward-move *spd*



Si l'agent ne s'est pas encore déplacé au cours de l'itération courante (**fd-ok?** vaut true), il inhibe son déplacement pour le restant de cette itération (**fd-ok?** passe à false). S'il n'y a pas d'obstacle devant (**free-ahead?** renvoie nobody), alors l'agent avance d'une distance *spd* (ou *speed* si *spd* est supérieur à *speed*), sinon il entre en collision avec l'obstacle devant et perd une quantité d'énergie égale à **collision-damage**, de même que l'obstacle percuté.

(voir aussi [collision-damage](#), [fd-ok?](#), [free-ahead?](#), [random-move](#), [speed](#))

free-ahead?

free-ahead? *dist*



Renvoie un des agents présent devant dans un cone d'ouverture 135° et jusqu'à une distance *dist* (sans prendre en compte les burgers, les graines, les agents de perception, et les missiles).

(voir aussi [collision-damage](#), [fd-ok?](#), [forward-move](#), [random-move](#), [speed](#))

give-energy

give-energy *agt nrj*



Donne une quantité d'énergie **nrj** au robot **agt**. 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](#))

give-food

give-food *agent food*



Si l'agent **agent** est à une distance inférieure ou égale à 2 de l'agent appelant, le *Harvester* ou l'*Explorer* qui fait l'appel transfère une quantité **food** de nourriture vers le robot *agent*. Pour cela, la quantité **food** ne doit pas être supérieure à la quantité transportée par l'agent appelant (**carrying-food?**).

```
let b min-one-of my-bases [ distance myself ]
if (b != nobody) and (distance b <= 2)
[ give-food b carrying-food? ]
```

(voir aussi [take-food](#), [carrying-food?](#))

launch-faf

launch-faf **target**



Envoie un nouveau missile de type « fire and forget » vers la cible **target**, à condition d'une part d'avoir encore des missiles de ce type en réserve (**nb-fafs**) et d'autre part de respecter un délai d'attente (**waiting**) entre deux tirs successifs.

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

launch-rocket

launch-rocket **dir**



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](#))

new-Explorer

new-Explorer **nb agt**



Demande la création de *nb* nouveaux *Explorers* par la base *agt*. Les nouveaux *Explorers* sont créés avec les paramètres par défaut des *Explorers* 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, couleurs ennemies), la connaissance des bases de son équipe et une mémoire initialement vide. Les nouveaux robots sont créés avec une direction aléatoire, à une distance de 1 de la base. L'énergie de la base est décrétementée de la quantité d'énergie associée à la création d'un *Explorer* (voir tableau 1)

```
if (energy > 12000) [ new-Explorer 1 self ]
```

(voir aussi [new-Harvester](#), [new-RocketLauncher](#))

new-faf

new-faf **nb**



Demande la création de *nb* nouveaux missiles de type « fire and forget » (*Fafs*). Cela n'est possible que si cela n'entraîne pas un dépassement du nombre maximal de missiles de ce type autorisé (**max-fafs**). L'énergie du robot (*Base* ou *RocketLauncher*) est décrétementée de la quantité d'énergie associée à la création d'un *Faf* (**faf-cost**)

(voir aussi [launch-faf](#), [launch-missile](#), [max-fafs](#), [max-missiles](#), [new-missile](#))

new-Harvester

new-Harvester **nb agt**



Demande la création de *nb* nouveaux *Harvesters* par la base *agt*. Les nouveaux *Harvesters* sont créés avec les paramètres par défaut des *Harvesters* (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, couleurs ennemies), la connaissance des bases de son équipe et une mémoire initialement vide.

Les nouveaux robots sont créés avec une direction aléatoire, à une distance de 1 de la base.

L'énergie de la base est décrétementée de la quantité d'énergie associée à la création d'un *Harvester* (voir tableau 1)

```
if (energy > 12000) [ new-Harvester 1 self ]
```

(voir aussi [new-Explorer](#), [new-RocketLauncher](#))

new-missile

new-missile nb



Demande la création de *nb* nouveaux missiles « classiques » (*Missiles*). Cela n'est possible que si cela n'entraîne pas un dépassement du nombre maximal de missiles de ce type autorisé (**max-missiles**). L'énergie du robot (*Base* ou *RocketLauncher*) est décrétementée de la quantité d'énergie associée à la création d'un *Faf* (**faf-cost**)

(voir aussi [launch-faf](#), [launch-rocket](#), [max-missiles](#), [max-fafs](#), [new-faf](#))

new-RocketLauncher

new-RocketLauncher nb agt



Demande la création de *nb* nouveaux *RocketLaunchers* par la base *agt*. Les nouveaux *RocketLaunchers* sont créés avec les paramètres par défaut des *RocketLaunchers* (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, couleurs ennemies), la connaissance des bases de son équipe et une mémoire initialement vide. Les nouveaux robots sont créés avec une direction aléatoire, à une distance de 1 de la base.

L'énergie de la base est décrétementée de la quantité d'énergie associée à la création d'un *RocketLauncher* (voir tableau 1)

```
if (energy > 12000) [ new-RocketLauncher 1 self ]
```

(voir aussi [new-Explorer](#), [new-Harvester](#))

new-wall

new-wall nb agt



Demande la création de *nb* murs par la base *agt*. L'énergie de la base est décrétementée de la quantité d'énergie associée à la création d'un mur.

perceive-base

perceive-base clr



Renvoie les agents de type *Base* et de couleur **clr** dans le rayon de perception du robot (**detection-range**).

(voir aussi [detection-range](#), [enemy1](#), [enemy2](#), [friend](#), [perceive-base-in-cone](#), [perceive-*](#))

perceive-base2

perceive-base2 *clr1 clr2*



Renvoie les agents de type *Base* et de couleur **clr1** ou **clr2** dans le rayon de perception du robot (**detection-range**).

(voir aussi [detection-range](#), [enemy1](#), [enemy2](#), [friend](#), [perceive-base-in-cone](#), [perceive-*](#))

perceive-base-in-cone

perceive-base-in-cone *clr angle*



Renvoie les agents de type *Base* et de couleur **clr** dans un cône d'ouverture **angle** et dans le rayon de perception du robot (**detection-range**).

(voir aussi [detection-range](#), [enemy1](#), [enemy2](#), [friend](#), [perceive-base](#), [perceive-*](#))

perceive-base2-in-cone

perceive-base2-in-cone *clr1 clr2 angle*



Renvoie les agents de type *Base* et de couleur **clr1** ou **clr2** dans un cône d'ouverture **angle** et dans le rayon de perception du robot (**detection-range**).

(voir aussi [detection-range](#), [enemy1](#), [enemy2](#), [friend](#), [perceive-base](#), [perceive-*](#))

perceive-food

perceive-food

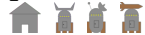


Renvoie les agents de type *Burger* dans le rayon de perception de l'agent (**detection-range**).

(voir aussi [detection-range](#), [perceive-food-in-cone](#), [perceive-*](#))

perceive-food-in-cone

perceive-food-in-cone *angle*



Renvoie les agents de type *Burger* dans un cône d'ouverture **angle** et dans le rayon de perception du robot (**detection-range**).

(voir aussi [detection-range](#), [perceive-food](#), [perceive-*](#))

perceive-robots

perceive-robots *clr*



Renvoie les agents de type *Explorer*, *Harvester* ou *RocketLauncher* et de couleur **clr** dans le rayon de perception du robot (**detection-range**).

(voir aussi [detection-range](#), [enemy1](#), [enemy2](#), [friend](#), [perceive-robots-in-cone](#), [perceive-*](#))

perceive-robots2

perceive-robots2 *clr1 clr2*



Renvoie les agents de type *Explorer*, *Harvester* ou *RocketLauncher* et de couleur **clr1** ou **clr2** dans le rayon de perception du robot (**detection-range**).

(voir aussi [detection-range](#), [enemy1](#), [enemy2](#), [friend](#), [perceive-robots2-in-cone](#), [perceive-*](#))

perceive-robots-in-cone

perceive-robots-in-cone *clr angle*



Renvoie les agents de type *Explorer*, *Harvester* ou *RocketLauncher* et de couleur **clr** dans un cône d'ouverture **angle** et dans le rayon de perception du robot (**detection-range**).

(voir aussi [detection-range](#), [enemy1](#), [enemy2](#), [friend](#), [perceive-robots](#), [perceive-*](#))

perceive-robots2-in-cone

perceive-robots-in-cone *clr1 clr2 angle*



Renvoie les agents de type *Explorer*, *Harvester* ou *RocketLauncher* et de couleur **clr1** ou **clr2** dans un cône d'ouverture **angle** et dans le rayon de perception du robot (**detection-range**).

(voir aussi [detection-range](#), [enemy1](#), [enemy2](#), [friend](#), [perceive-robots2](#), [perceive-*](#))

perceive-seeds

perceive-seeds *clr*



Renvoie les agents de type *Seed* et de couleur **clr** dans le rayon de perception du robot (**detection-range**). Les graines « ennemies » (**enemy1**, **enemy2**) ne sont perceptibles que quand elle ont un âge > 500, alors que toutes les graines amies (**friend**) sont perceptibles.

(voir aussi [detection-range](#), [enemy1](#), [enemy2](#), [friend](#), [perceive-seeds-in-cone](#), [perceive-*](#))

perceive-seeds-in-cone

perceive-seeds-in-cone *angle 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 (**detection-range**). Les graines « ennemies » (**enemy1**, **enemy2**) ne sont perceptibles que quand elle ont un âge > 500, alors que toutes les graines amies (**friend**) sont perceptibles.

(voir aussi [detection-range](#), [enemy1](#), [enemy2](#), [friend](#), [perceive-seeds](#), [perceive-*](#))

perceive-specific-robots

perceive-specific-robots *clr br*



Renvoie les agents dont l'espèce est **br**, et de couleur **clr** dans le rayon de perception du robot (**detection-range**).

(voir aussi [detection-range](#), [enemy1](#), [enemy2](#), [friend](#), [perceive-specific-robots-in-cone](#), [perceive-*](#))

perceive-specific-robots2

perceive-specific-robots2 *clr1 clr2 br*



Renvoie les agents dont l'espèce est **br**, et de couleur **clr1** ou **clr2** dans le rayon de perception du robot (**detection-range**).

(voir aussi [detection-range](#), [enemy1](#), [enemy2](#), [friend](#), [perceive-specific-robots-in-cone](#), [perceive-*](#))

perceive-specific-robots-in-cone

perceive-specific-robots-in-cone *clr br angle*



Renvoie les agents dont l'espèce est **br**, et de couleur **clr** dans un cône d'ouverture **angle** et dans le rayon de perception du robot (**detection-range**).

(voir aussi [detection-range](#), [enemy1](#), [enemy2](#), [friend](#), [perceive-specific-robots](#), [perceive-*](#))

perceive-specific-robots2-in-cone

perceive-specific-robots2-in-cone *clr1 clr2 br angle*



Renvoie les agents dont l'espèce est **br**, et de couleur **clr1** ou **clr2** dans un cône d'ouverture **angle** et dans le rayon de perception du robot (**detection-range**).

(voir aussi [detection-range](#), [enemy1](#), [enemy2](#), [friend](#), [perceive-specific-robots](#), [perceive-*](#))

perceive-walls

perceive-walls



Renvoie les agents de type *Wall* dans le rayon de perception de l'agent (**detection-range**).

(voir aussi [detection-range](#), [perceive-walls-in-cone](#), [perceive-*](#))

perceive-walls-in-cone

perceive-walls-in-cone



Renvoie les agents de type *Wall* dans un cône d'ouverture **angle** et dans le rayon de perception du robot (**detection-range**).

(voir aussi [detection-range](#), [perceive-walls](#), [perceive-*](#))

plant-seeds

plant-seeds *color nb*



Plante *nb* graines de *Burgers* dans le patch courant. Le nombre maximum de graines dans un patch est donné par *max-seeds*, chaque graine coûte *seed-cost* unités de *carrying-food?* à planter. Au bout d'un temps égal à *maturation-time*, chaque graine produira un Burger dont l'énergie est comprise entre *seeded-burger-min-nrj* et *seeded-burger-max-nrj*. Si le plan de burger est écrasé par un robot autre qu'un Harvester, l'âge de maturation est diminué de 100.

(voir aussi [max-seeds](#), [seed-cost](#), [maturation-time](#), [seeded-burger-min-nrj](#), [seeded-burger-max-nrj](#))

random-move

random-move



Choisit une orientation aléatoire dans un cône d'ouverture 90° par rapport à la direction courante puis appelle la fonction `forward-move` pour avancer.

(voir aussi [`collision-damage`](#), [`fd-ok?`](#), [`forward-move`](#), [`free-ahead?`](#), [`speed`](#))

take-food

take-food *food*



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

```
let b min-one-of perceive-food [ distance myself ]
if (b != nobody) and (distance b <= 2)
  [ take-food b ]
```

(voir aussi [`give-food`](#), [`carrying-food?`](#), [`energy`](#))

take-wall

take-wall *bloc*



Si le mur **bloc** est à une distance inférieure ou égale à 2, et si le *Harvester* ne transporte pas déjà de bloc (**carrying-wall?** à false), ramasse le bloc et passe **carrying-wall?** à true.

(voir aussi [`drop-food`](#), [`carrying-wall?`](#))

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)