Ecole Polytechnique de Bruxelles
Université Libre de Bruxelles

Rapport du projet de Ba2 : Groupe 9

Robots en Essaim : Explorez

Etudiants :
Sacha Alcalde Mangen
Shankar Baba
Rosine Desmet
Nathan Dwek
Bernard Gonda

Superviseur :
Ir. Anh Vu Doan Nguyen

27 juin 2014

Abstract

The goal of this project was to design a swarm-intelligent behaviour for virtual robots using the ARGoS simulator. These robots had to explore an unknown environment with obstacles in order to ultimately loop between spots marked on the ground and a starting area. The fundamental paradigm was that a single set of rules would be followed independently by every robot, which would allow a swarm-intelligent behaviour to emerge through the robot interactions prescribed by these rules. For that to work, exploration, shortest path-finding and obstacle-avoidance algorithms were needed, along with elementary automated decision making, communication and odometry. These concepts were implemented using the ARGoS loop approach, which means that the same sequence of actions takes place at every step, while only events occurring during that step can influence these actions. A working solution was produced using the Lua language, then put to the test and could quickly be enhanced accordingly thanks to the flexible framework. This allowed robots to meet the objectives and information was gathered on parameters meaningful for the experiment along with performance indexes. Future development should be focused on optimizing these parameters and enable the footbots with the latest advances in swarm intelligence, like the usage of simulated pheromones, for example.

Résumé

Le but de ce projet était de créer une intelligence en essaim pour des robots modélisés dans le simulateur ARGoS. Ceux-ci devaient explorer un environnement inconnu qui comportait des obstacles afin d'ensuite faire des allers-retours entre des zones marquées aux sols et leur nid de départ. Le principe de base était que le même ensemble de règles devrait être suivi de manière indépendante par chaque robot ; la caractéristique intelligente de l'ensemble de robots devant émerger à travers les interactions entre robots prescrites par ces règles. Pour cela, des procédures d'exploration, de recherche du plus court chemin et d'évitement furent nécessaires ainsi que des principes de base de prise de décision, d'odométrie et de communication. Ces concepts furent mis en pratique en utilisant l'approche loop d'ARGoS, qui implique que le robot exécute la même séquence d'opérations à chaque pas. Une solution fonctionnelle en Lua fut construite, testée et ensuite améliorée en conséquence grâce au turnaround loop très court offert par ARGoS. Cette solution permet aujourd'hui au robots de remplir les objectifs, et des informations ont été collectées sur des paramètres influençant l'expérience, ainsi que des indicateurs de performances. Des développements futurs pourraient être consacrés à l'optimisation de cette solution et à utiliser les dernières trouvailles de l'intelligence en essaim, comme par exemple les phéromones-messages artificielles, dans le comportement des footbots.

Table des matières

Chapitre 1

Introduction

Le but du projet est de doter un essaim de robots d'un comportement intelligent afin qu'ils soient capables [cah13]

- d'explorer un environnement inconnu afin de localiser des «sources» symbolisées par des taches noires au sol
- d'«exploiter» les sources découvertes en faisant des allers-retours entre celles-ci et le nid
- de partager l'information accumulée afin d'optimiser l'exploitation des sources
- d'accomplir ces tâches dans un environnement comportant des obstacles, tout en gérant leur autonomie limitée

L'essaim de robots et son environnement sont simulés par ARGoS, un simulateur développé par le laboratoire IRIDIA.

1.1 Intérêt du projet

Comme son nom l'indique, la robotique en essaim met en œuvre un nombre élevé de robots afin d'effectuer une tâche. Ceci est très différent de ce qui se fait habituellement en robotique «classique» où un petit nombre de robots extrêmement sophistiqués est déployé afin de résoudre une problématique. Les principes fondamentaux derrière la programmation d'un essaim de robots peu coûteux mais aux capacités plus limitées sont donc aussi différents : chaque individu ne doit plus être considéré comme infaillible, et la perte d'un robot prend moins d'importance, tant qu'elle profite à l'essaim tout entier. Ceci ouvre de nouvelles voies dans de nombreux domaines, par exemple dans le cas très concret du déminage ou lorsqu'il faut opérer dans une zone hautement hostile au sens plus général (intervention en milieu radioactif, en grande profondeur, ...). [Sha07]

1.2 Résultats attendus

L'objectif premier est de développer un comportement qui permet aux robots de survivre et d'exploiter une ressource de manière autonome dans un environnement non connu à l'avance.

Dans un premier temps, les robots seront considérés comme omniscients et connaîtront donc l'environnement à explorer. Cette connaissance leur sera ensuite retirée. Une communication entre les robots pourra être envisagée par la suite et permettra notamment de mieux gérer l'information incomplète.

Même si la tâche à accomplir est au niveau de l'essaim, ce dernier ne sera jamais programmé directement. Le principe même du projet est de développer un comportement qui sera suivi par chaque robot indépendamment. Des interactions locales entre robots, physiques ou non, émergera un comportement global qui devra être étudié et être rendu prévisible.

Des outils de mesure, afin de calculer la qualité du comportement en essaim, devront être élaborés. Ils devront établir la performance des solutions proposées en fonction des objectifs initiaux. [cah13]

Ce rapport a été divisé en quatre parties : une introduction à la notion d'intelligence artificielle et aux outils utilisés, la problématique du déplacement des robots, et enfin la mise en place à un niveau plus élevé d'un comportement «intelligent» permettant de remplir les objectifs décrits plus haut. Pour finir, nous conclurons par l'évaluation des performances et les perspectives d'amélioration.

Chapitre 2

L'intelligence artificielle

Afin de mener à bien l'objectif énoncé dans l'introduction, les robots doivent se comporter de manière intelligente, et le choix de la bonne action est crucial. Il est cependant délicat de définir une seule «meilleure» action. En effet, la rationalité d'un agent peut être difficile à mesurer. D'après [RND10] :

> «La rationalité n'est pas synonyme de perfection, la rationalité maximise la performance espérée tandis que la perfection maximise la performance réelle.»

De plus, dans un système multi-agents, un choix d'action peut s'avérer bénéfique pour un agent mais mauvais pour l'ensemble du groupe. C'est pourquoi il est préférable de concevoir les mesures de performance en fonction de ce que l'on souhaite obtenir dans l'environnement et non en fonction de la façon dont devrait se comporter un agent.

Pour prendre un point de vue plus «haut niveau» sur cette problématique, il est possible d'utiliser le concept de machine à état et de diagramme d'état pour décrire un comportement. L'agent commence dans un état initial et peut à partir de cet état accomplir un certain nombre d'actions qui l'amènent dans d'autres états, à partir desquels d'autres actions et transitions sont possibles, et ainsi de suite. Ceci définit l'espace des états du système, c'est à dire l'ensemble de tous les états accessibles par une séquence d'action à partir de l'état initial. Cet espace des états peut être vu comme un graphe où les nœuds représentent les états et les branches les transitions. Il en découle la notion de chemin (et de coût associé) représentant une séquence d'états reliés par une séquence d'actions. Cette approche est utilisée dans la section 6.3. [PBP+13, RND10]

D'autre part, un agent reçoit des percepts par l'intermédiaire de ses capteurs. L'agent réagit alors en exerçant une action sur l'environnement grâce à ses effecteurs. Il est aussi possible de découper le comportement autour de ce cycle, et c'est ce qui est fait par ARGoS, nous y reviendrons longuement dans la suite. La description des différents capteurs et effecteurs des footbots, ainsi que celle de l'environnement et des outils utilisés pour le

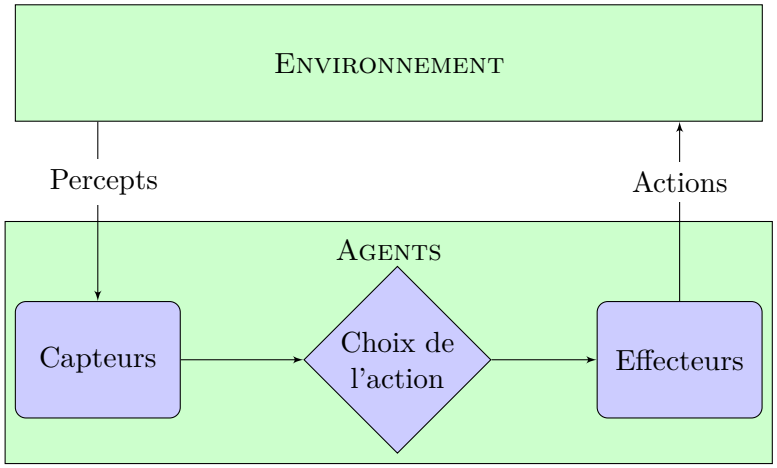


Figure 2.1 – Cycle d'interaction entre l'environnement et les agents

représenter est faite dans le chapitre suivant.

Chapitre 3

Présentation du simulateur ARGoS et des footbots

3.1 Capteurs et actuateurs des footbots

Comme présenté plus haut, les capteurs et actuateurs des agents déterminent bien évidemment les informations qu'ils sont capables de recueillir et les actions qu'ils peuvent effectuer.

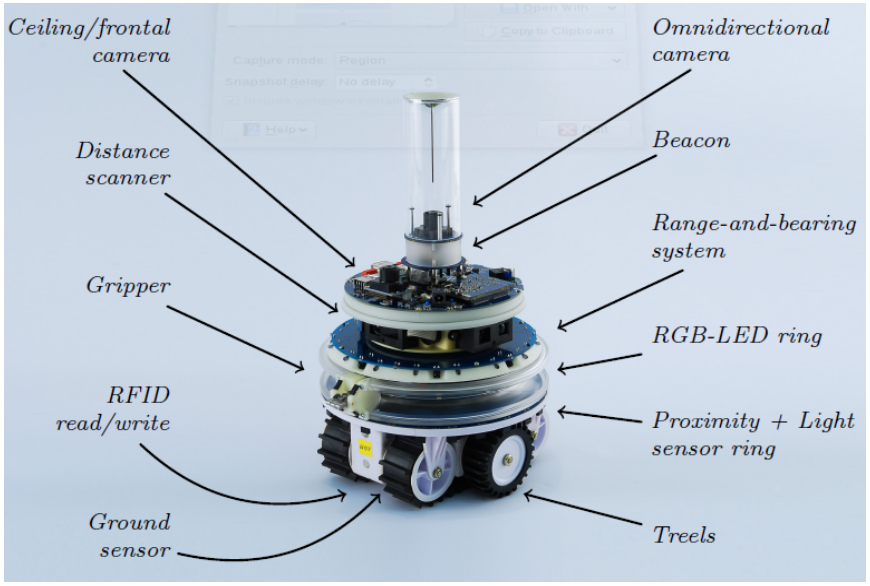


FIGURE 3.1 – Les différents senseurs et actuateurs d'un footbot [Pin]

Dans le cadre du projet, on considère qu'il suffit que le robot passe sur une «source» pour l'exploiter. Les différents capteurs et actuateurs intéressants sont alors les roues, le senseur de proximité et le *distance scanner* pour la partie déplacement, le *ground sensor* pour la partie exploration (car il permet

de lire la couleur du sol), et enfin les différentes LED et le système *range and bearing* ainsi que les capteurs associés pour la partie communication.

Comme annoncé dans l'introduction, l'essaim de robots devra être simulé dans ARGoS, un simulateur de robots développé notamment par le département IRIDIA de l'ULB. Ce simulateur a besoin de deux fichiers pour lancer une expérience : un fichier XML qui permet de configurer l'arène et l'expérience en général (moteur physique, capteurs et actuateurs disponibles, ...) [PTO+11] d'une part, et un fichier dictant le (même) comportement individuel de chaque robot d'autre part. Pour l'instant, les instructions peuvent être écrites soit en C++, soit en Lua.

3.2 Choix du langage informatique

D'une part, les principaux avantages de C++, sa rapidité d'exécution et ses nombreuses librairies notamment, ne sont pas primordiaux dans le cadre de ce projet. De plus, il nous est moins familier que Lua qui se rapproche fortement de python tant au niveau de la syntaxe que de l'approche fonctionnelle.

Lua étant un langage de scripting, il est plus adapté aux besoins du groupe car la réalisation du projet passe par de nombreuses petites expérimentations mais ne devrait pas aboutir à un comportement final comportant de très nombreuses lignes de code. En effet, ce type de langage permet de concevoir des prototypes de programmes rapidement.

Enfin sa simplicité et sa lisibilité sont des aspects très importants dans un travail de groupe où chacun doit être capable de comprendre et d'améliorer le comportement en développement. [SER13, MAU13]

Au vu des raisons énoncées ci-dessus, Lua a été choisi comme langage de programmation.

3.3 Structure d'un comportement en Lua

Le template de code fourni par ARGoS présenté plus bas montre les deux fonctions les plus importantes parmi les fonctions que le simulateur doit absolument trouver dans le comportement. La fonction *init* qui est exécutée une fois par chaque footbot au début de l'expérience, et la fonction *step* qui est exécutée par chaque footbot à chaque pas de la simulation. C'est bien évidemment cette fonction *step* qui représente presqu'entièrement le comportement du robot. Deux types d'évènements peuvent modifier la façon dont cette fonction step s'exécute : d'une part, les percepts ayant lieu au cours du pas de simulation, et d'autre part un ensemble de variables globales (qui survivent après l'exécution de la fonction *step* et sont donc accessibles par les instances suivante de cette fonction) qui détermine entièrement l'état

du robot lorsque celui-ci entame ce pas de simulation. Enfin, la fonction step peut agir en retour sur ces variables d'état. [Pin]

Listing 3.1 – Structure de base d'un comportement en Lua

```
--[[ This function is executed every time
     you press the 'execute' button ]]
function init()

end

--[[ This function is executed at each time step
     It must contain the logic of your controller ]]
function step()

end
```

3.4 Configuration de l'arène et de l'expérience

L'arène a aussi dû être créée par le groupe selon les spécifications données :
— L'arène est carrée de côté 10m
— Le nid et les sources font 50cm de rayon
— L'arène comporte éventuellement des obstacles
— Les robots ont une batterie de 100 et perdent 0.2 par pas [1]
— Les robots récupèrent leur batterie en passant par le nid

L'arène utilisée comporte entre 50 et 100 obstacles cylindriques de 10cm de rayon disposés au hasard. Différentes répartitions de ressources ont été envisagées. De plus, un script python a été utilisé pour mettre en place aléatoirement de nombreuses expériences et automatiser la collecte de données. Une configuration initiale typique est montrée à la figure 3.2.

1. Ceci impose une vitesse $> 25cms^{-1}$ afin que les foobots puissent raisonnablement faire des allers-retours vers les coins de l'arène, ce qui est largement supérieur à la vitesse de $10cms^{-1}$ utilisée dans des travaux précédents [PBP+13, CGN+10].

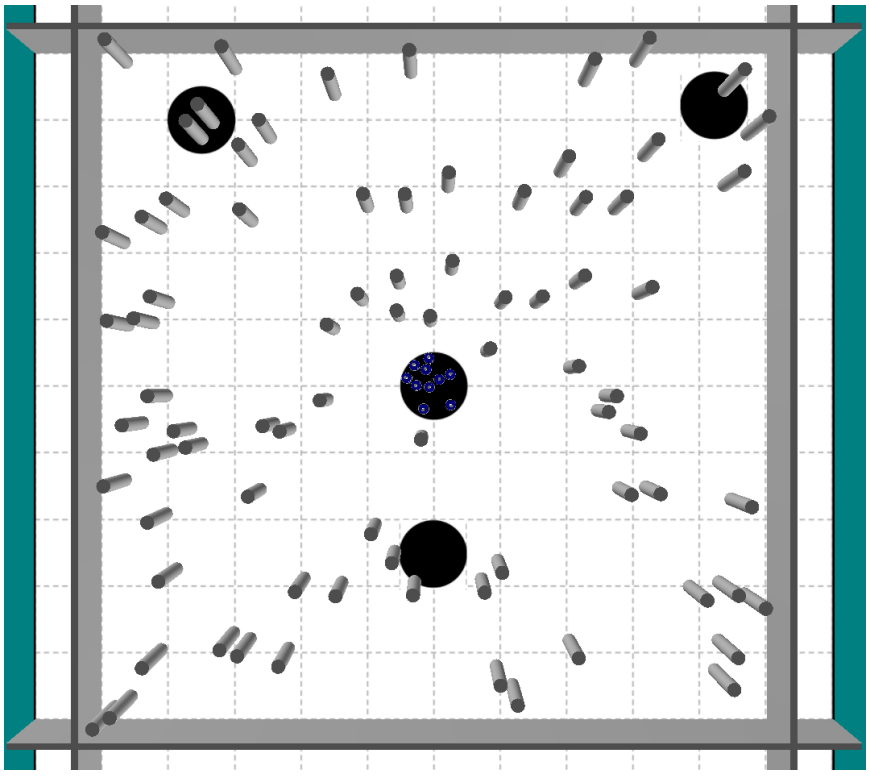


FIGURE 3.2 – Exemple de configuration initiale

Chapitre 4

Déplacement et évitement d'obstacles

Comme il a été présenté dans l'introduction sur l'intelligence artificielle, et vu en pratique dans la structure de base d'un comportement Lua interprétable par ARGoS, un footbot exécute à chaque step une séquence d'opérations. On peut distinguer dans cette séquence trois types d'opérations (cf chapitre 2) : l'écoute des capteurs, la prise de décision et les interactions sur l'environnement par l'intermédiaire des effecteurs, que l'on nommera ici simplement sous le nom d'actions. Ces actions sont donc limitées et déterminées par les effecteurs dont dispose le robot et qui sont décrits au chapitre 3. Parmi ces actions, l'une des plus importantes, accomplie par chaque robot à tout instant sera examinée dans ce chapitre : le déplacement. Même si ce projet met l'accent sur le comportement haut niveau des footbots et leurs interactions, l'efficacité du déplacement a une influence majeure d'abord sur la survie des robots et ensuite sur la performance de leur comportement.

4.1 Aspects physiques du déplacement

Tout d'abord, il est intéressant de se pencher sur le lien entre les paramètres physiques du robot et la manière dont celui-ci peut effectuer l'action simple «se rendre d'un point A à un point B» dans le cas où le robot agit seul dans un environnement sans obstacles. Ensuite, nous verrons comment le robot peut s'accommoder des obstacles (fixes, prévisibles) et autres robots (mobiles, imprévisibles) à partir d'une ou plusieurs de ces actions simples.

L'effecteur dont un footbot dispose afin de se déplacer est une paire de roues dont les vitesses peuvent être fixées de manière indépendante. A chaque instant, les seuls deux mouvements auxquels peut accéder le robot sont donc une translation parallèle aux roues et une rotation autour d'un point au milieu de l'axe des roues. La mécanique [LD06] nous indique que la

composition de ces deux mouvements est suffisante pour permettre au robot de se déplacer librement dans un plan mais surtout, elle nous donne la relation entre les vitesses des deux roues et la vitesse générale ainsi que la vitesse de rotation du robot :

$$\begin{cases} v_g = \frac{v_r + v_l}{2} \\ \omega_g = \frac{v_r - v_l}{l_{axe}} \end{cases} \tag{4.1}$$

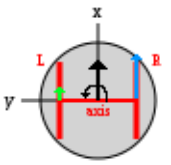


FIGURE 4.1 – Signification physique de v_r, v_l, v_g *et* ω_g [Pin]

4.2 Déplacement sans obstacles

A partir de cette loi des vitesses, il est aisé de construire un algorithme permettant à un robot de converger vers sont but selon une trajectoire souple et à vitesse constante.

Algorithm 4.1 Convergence simple sans évitement d'obstacle

Ensure: footbot converges towards the goal at speed $SPEED$
  while goal not reached **do**
    update footbot position and orientation
    calculate $\theta \equiv$ angle between the direction of the goal from the footbot and footbot orientation
    $v_{right} =$ convergence$(\theta,\ SPEED)$
    $v_{left} = 2 \times SPEED - v_{right}$ ▷*so that overall speed stays equal to SPEED*
  end while

Où convergence$(\theta, SPEED)$ fixe la convergence du robot vers son goal. Elle doit satisfaire :

$$\begin{cases} convergence(0, SPEED) = SPEED \\ convergence(\overbrace{0 < \theta \leq \pi}^{\text{goal à gauche du robot}}, SPEED) > SPEED \\ convergence(\overbrace{0 > \theta \geq -\pi}^{\text{goal à droite du robot}}, SPEED) < SPEED \end{cases} \tag{4.2}$$

Pour une fonction convergence$(\theta, SPEED)$ donnée satisfaisant à cette condition, le footbot peut donc se rendre d'un point A à un point B, tant qu'il ne rencontre pas d'obstacles sur son trajet. Notons que cet algorithme suit la structure «écoute des capteurs - prise de décision - action» évoquée plusieurs fois déjà, ce qui fait qu'il peut s'intégrer presque tel quel dans la fonction *step* demandée par ARGoS. Dans notre projet nous avons choisi

$$convergence(\theta, SPEED) = \begin{cases} \left(\frac{\pi - |\theta|}{\pi}\right)^{\kappa} \times SPEED & \text{si } \theta \geq 0 \\ \left(2 - \left(\frac{\pi - |\theta|}{\pi}\right)^{\kappa}\right) \times SPEED & \text{si } \theta < 0 \end{cases}$$

Où $\kappa > 0$ est un paramètre qui fixe l'intensité de la convergence.

4.3 Évitement *greedy* d'obstacles lointains

Une première manière de permettre au robot d'éviter des obstacles est d'activer le capteur *distance scanner* longue distance qui permet de détecter des obstacles jusqu'à 150cm du footbot. Un évitement consiste alors à chercher à chaque pas la direction la plus proche de la direction du goal parmi les directions pour lesquelles l'obstacle repéré est le plus éloigné du robot (ou non existant) et à converger vers celle-ci. Ceci permet au robot d'éviter plusieurs obstacles à la fois, tout en gardant une trajectoire très optimisée.

Algorithm 4.2 Convergence avec évitement *greedy* d'obstacles lointains

while goal not reached **do**
  update footbot position and orientation
  find the direction which is closest to goal direction amongst directions with the furthest or no obstacles
  find $\theta \equiv$ angle between footbot orientation and this optimal direction
  $v_{right} =$ convergence(θ, $SPEED$)
  $v_{left} = 2 \times SPEED - v_{right}$
end while

La fonction convergence doit satisfaire aux mêmes contraintes que précédemment et reste inchangée dans notre projet. Pour que l'évitement soit efficace, il faut que κ soit assez élevé pour que le footbot s'oriente rapidement vers la direction optimale (puisque celle-ci est réévaluée à chaque pas et change donc fréquemment).

Cet évitement est *greedy* car il sélectionne toujours la direction la plus proche de celle du goal parmi les directions acceptables (voir plus haut) ce qui signifie qu'un choix momentanément optimal est fait à chaque pas. Ceci assure que le goal est atteint en un temps «raisonnablement proche» du temps optimal [MS08] mais peut cependant mener à des collisions à cause de différentes sources d'erreurs et approximations (footbot de taille non nulle, rayons du *distance scanner* parfois parfaitement tangents à un obstacle, *distance scanner* ne faisant pas une mesure par direction à chaque pas, nombre de directions mesurées bien évidemment fini, ...). Une première manière de pallier cela est de regrouper plusieurs mesures voisines en gardant systématiquement la mesure la moins optimiste. Malgré cela, un évitement plus sûr d'obstacles proches est aussi nécessaire pour limiter le plus possible les collisions.

Cet évitement d'obstacles est directement utilisé dans notre projet. Son implémentation est détaillée dans l'annexe 4.A.1.

4.4 Evitement sûr d'obstacles proches

La manière la plus directe de permettre au footbot d'éviter des obstacles ou autres robots lorqu'ils sont dangereusement proches est d'alors exécuter une routine d'évitement à la place de la routine de convergence précédente.

Algorithm 4.3 Convergence avec évitement d'obstacles proches

```
while goal not reached do
  update footbot position and orientation
  read proximity sensor                    ▷or whatever other sensor used
  if no obstacles too close then
    do previous convergence
  else
    v_right = avoidance(proximity sensor reading, SPEED)
  end if
  v_left = 2 × SPEED − v_left
end while
```

Où avoidance(proximity sensor reading, $SPEED$) fixe la routine d'évitement du robot. Son implémentation est très libre et peut fortement varier en fonction du capteur utilisé pour détecter les obstacles. On peut par exemple utiliser le senseur *proximity* du footbot, qui associe aux 24 directions autour du robot une valeur entre zéro et un : une valeur zéro indique qu'aucun obstacle n'est perçu à moins de 10cm dans la direction donnée tandis qu'une valeur supérieure indique qu'un objet a été détecté. Cette valeur augmente au fur et à mesure que le robot se rapproche de l'obstacle. [Pin]

Dans notre projet nous avons choisi

$$avoidance(dir, prox) = \begin{cases} \frac{-\alpha+(1-prox)^{\beta}\cdot dir}{11} SPEED & \text{si } dir \leq 12 \\ \frac{(22+\alpha)-(1-prox)^{\beta}\cdot(25-dir)}{11} SPEED & \text{si } dir \geq 12 \end{cases}$$

Où $1 \leq dir \leq 12$ est la direction de l'obstacle perçu le plus proche et $0 \leq prox \leq 1$ donne la proximité de cette obstacle. Comme présenté plus haut, ce sont les deux informations dont on dispose si l'on utilise le capteur de proximité. $1 \leq \alpha \leq 12$ est un paramètre qui fixe l'influence de la direction de l'obstacle le plus proche et $0 \leq \beta$ fixe l'influence de la proximité de cette obstacle. Cet évitement est partiellement tiré des exemples fournis sur le site du cours présentant ARGoS. [Pin]

4.5 Evitement intermédiaire

Additionnellement, il est possible d'utiliser le capteur *short range* du *distance scanner* pour que le robot puisse déjà commencer à dévier sa trajectoire s'il détecte des obstacles proche de moins de 30cm [Pin]. Ceci

permet de déclencher plus tôt le même évitement que ci-dessus aux constantes numériques près (évitement moins brusque). Cet évitement est beaucoup plus sûr que le premier évitement présenté, tout en étant forcément plus souple que l'évitement «d'urgence» précédent.

De plus, les capteurs *short range* et *long range* sont des capteurs rotatifs, ce qui signifie que la table des mesures n'est pas complètement renouvelée à chaque pas. [1] Les deux capteurs étant orientés perpendiculairement, il est donc avantageux de les utiliser tous les deux afin de ne pas avoir de direction pour laquelle la dernière mesure est «trop vieille».

Grâce aux différentes routines élémentaires présentées ci-dessus, il est donc possible de construire une solution pratique à la partie déplacement du cahier des charges. L'implémentation détaillée des routines d'évitement supplémentaire est donnée dans l'annexe 4.A.2.

4.6 Déplacement selon un chemin précalculé

Il est cependant possible d'améliorer cette solution en fonction de la connaissance de son environnement dont dispose le robot. Ainsi, dans le cas omniscient ou si le robot est capable de construire une carte de son environnement reprenant la position des différents obstacles, il peut-être judicieux d'utiliser un algorithme de recherche du plus court chemin. La manière la plus directe de faire est de donner au footbot une liste de points successifs qui le mèneront au goal final. Ceci permet de réutiliser facilement les algorithmes déjà présentés tout en étant parfaitement compatible avec les valeurs de retour typiques d'un algorithme de recherche du plus court chemin. En effet, la plupart de ces algorithmes utilisent une représentation en graphe d'un environnement. La valeur de retour d'une telle recherche est donc une liste des nœuds qu'il faut parcourir dans le graphe afin d'arriver au but final, ce qui est précisément ce que cet algorithme fait.

1. D'où la nécessité de rafraîchir les tables de mesures de la manière qui est faite au listing 4.3.

Algorithm 4.4 Convergence avec chemin précalculé

```
Require: intermediate goals list ≡ list of points which lead to the goal while
  avoiding the obstacles
  for intermediate goal in intermediate goals list do
    while intermediate goal not reached do
      if no obstacles too close then
        do greedy avoidance towards intermediate goal
      else
        do close obstacle avoidance
      end if
    end while
  end for
```

L'implémentation de l'algorithme de recherche du plus court chemin qui fournit *intermediate goals list* est un problème à part entière. Avant de l'examiner plus en détails dans le chapitre suivant, il faut noter que malgré l'utilisation d'une recherche du plus court chemin qui devrait a priori permettre d'éviter les obstacles, le test d'obstacle est toujours présent, ainsi que la possibilité d'évitement. Il est évident que ceci est fait pour permettre d'éviter des objets inattendus tels que d'autres robots. Cependant, on peut dès lors se demander s'il ne faudrait pas aussi chercher à nouveau un plus court chemin après un évitement imprévu ou si le robot a dévié d'une distance significative de sa trajectoire prévue.

Annexe 4.A Implémentation du déplacement en Lua

4.A.1 Evitement *greedy*

Listing 4.1 – Initialisation

```
function init()
  robot.distance_scanner.enable()
  robot.distance_scanner.set_rpm(SCANNER_RPM)
  obstaclesTable={}
  for i=-PI+PI/DIR_NUMBER, PI-PI/DIR_NUMBER, 2*PI/DIR_NUMBER do
    obstaclesTable[i]=151
  end
  goalX=SOURCEX
  goalY=SOURCEY
end
```

Listing 4.2 – Structure générale

```
function step()
  odometry()
  obstaclesTable=updateObstaclesTable(obstaclesTable)
  move(goalX,goalY,obstaclesTable)
end
```

Listing 4.3 – Rafraîchir la table des mesures à chaque pas

```
function updateObstaclesTable(tabl, which)
  --tabl: the obstacleTable to refresh
  --(obstaclesTable or shortObstaclesTable)
  --which (string): which sensor to use
  --("short_range" or "long_range")
  local sensor, reading, angle, value, rAngle, rDistance
  for angle, value in pairs(tabl) do
    newValue=false
    for sensor, reading in pairs(robot
                                 .distance_scanner
                                 [which]) do
      rAngle = reading.angle
      rDistance=reading.distance
      if rDistance == -2 then rDistance=151 end
      if rDistance == -1 then rDistance=0 end
      if abs(angle-rAngle)<PI/DIR_NUMBER then
        if value>rDistance or not newValue then
          tabl[angle]=rDistance
          newValue = true
        end
      end
    end
  end
  return tabl
end
```

Listing 4.4 – Fonction move

```
function move(goalX,goalY,obstaclesTable)
  local goalDirection=findGoalDirection(posX,posY,goalX,goalY)
  local goalAngle=findGoalAngle(goalDirection, alpha)
  obstacleAvoidance(goalAngle, obstaclesTable)
end
```

Listing 4.5 – Trouver la direction du goal vu du footbot

```
function findGoalDirection(posX, posY, goalX, goalY)
  local deltaX=goalX-posX
  local deltaY=goalY-posY
  local goalDirection=math.atan(deltaY/deltaX)
  if deltaX<0 then
    goalDirection=goalDirection+PI
  end
  if goalDirection<0 then
    goalDirection=goalDirection+2*PI
  end
  return goalDirection
end

function findGoalAngle(goalDirection,alpha)
  local goalAngle=goalDirection-alpha
  if goalAngle>PI then
    goalAngle=goalAngle-2*PI
  end
  return goalAngle
end
```

Listing 4.6 – Trouver la direction optimale et la suivre

```
function obstacleAvoidance(goalAngle, obstaclesTable)
  local bestAngle, bestDistance, angle, distance
  bestDistance = -1
  for angle, distance in pairs(obstaclesTable) do
    if distance>bestDistance
       or (distance==bestDistance and not bestAngle)
       or (distance==bestDistance
         and abs(angle-goalAngle)<abs(bestAngle-goalAngle)) then
      bestDistance = distance
      bestAngle = angle
    end
  end
  getToGoal(bestAngle, CONVERGENCE)
end
```

Listing 4.7 – Fonction getToGoal

```
function getToGoal(angle, conv)
   if angle>=0 then --goal is to the left
      vLeft=speed*((PI-angle)/PI)^conv
      vRight = 2*speed-vLeft
   else --goal is to the right
      vRight=speed*((PI+angle)/PI)^conv
      vLeft = 2*speed - vRight
   end
   robot.wheels.set_velocity(vLeft, vRight)
end
```

4.A.2 Evitement d'obstacles proches

Listing 4.8 – Capteur supplémentaire

```
function step()
  ...
  emerProx, emerDir=readProxSensor()
  if emerProx>0 then
    emergencyAvoidance(emerProx, emerDir)
  else
  ...
end
```

Listing 4.9 – lecture du *proximity sensor*

```
function readProxSensor()
  local emerDir = 1
  local emerProx = robot.proximity[1].value
  for i=2,24 do
    if emerProx < robot.proximity[i].value
       or (emerProx == robot.proximity[i].value
           and abs(12-emerDir)<abs(12-i)) then
      emerDir = i
      emerProx = robot.proximity[i].value
    end
  end
  return emerProx, emerDir
end
```

Listing 4.10 – Fonction emergencyAvoidance

```
function emergencyAvoidance(emerProx,emerDir)
  local vLeft, vRight
  if emerDir <= 12 then --Obstacle is to the left
    vRight=((1-emerProx)^EMER_PROX_DEP*emerDir
            -EMER_DIR_DEP)
           *speed/11
    vLeft=2*speed-vRight
  else --Obstacle is to the right
    vLeft=((1-emerProx)^EMER_PROX_DEP*(25-emerDir)
           -EMER_DIR_DEP)
          *speed/11
    vRight=2*speed-vLeft
  end
  robot.wheels.set_velocity(vLeft, vRight)
end
```

Annexe 4.B Ajout d'une composante aléatoire à l'évitement

Le déplacement présenté jusqu'à présent est purement déterministe. Dans un système multi-agents où tous les agents suivent le même comportement déterministe, un phénomène peut apparaître où après une première collision, les collisions se répètent un grand nombre de fois, puisqu'après la première rencontre les deux agents impliqués suivent exactement le même comportement avec des conditions initiales proches (au moment de la collision). Ici, collision est à prendre au sens large puisque ce phénomène est par exemple présent dans les protocoles de télécommunication.

Les deux solutions les plus couramment utilisées sont l'instauration de priorités, où l'introduction d'une composante aléatoire comme par exemple l'*exponential backoff* en télécommunication, où, après i collisions, les deux agents tentent d'envoyer le paquet après un temps tiré aléatoirement dans $[0, t_i^{max}]$ avec t_i^{max} grandissant exponentiellement en fonction de i. [Bac14]

Dans notre projet, après une collision, les deux footbots tirent aléatoirement une nouvelle vitesse inférieure ou égale à la vitesse de base et gardent cette nouvelle vitesse pendant un bref temps après la collision, avant de reprendre leur vitesse originale, ce qui empêche la collision de se répéter.

4.B.1 Implémentation en Lua

Listing 4.11 – Vitesse aléaoire temporaire après évitement

```
function move(obstaclesTable, obstacleProximity,
             obstacleDirection, goalX,goalY)
  if obstacleProximity >= 30 then
    if not lastHit
       or currentStep-lastHit < RANDOM_SPEED_TIME then
      speed=BASE_SPEED
    end
    ...
  else
    speed, lastHit = newRandomSpeed(BASE_SPEED, lastHit)
    --/!\Global: survives beyond this step
    ...
  end
end

function newRandomSpeed(lastHit)
  if not lastHit
     or currentStep-RANDOM_SPEED_TIME > lastHit then
    local speed=robot.random.uniform(MIN_SPEED_COEFF,1)
                *BASE_SPEED
  end
  lastHit=currentStep
  return speed, lastHit
end
```

Chapitre 5

Recherche du plus court chemin

5.1 Algorithmes déterministes classiques

Dans le cas où les footbots disposent réellement d'une description complète de leur environnement, il est envisageable d'utiliser les algorithmes classiques de recherche du plus court chemin comme A* et Dijkstra. Ceux-ci fonctionnent en représentant l'environnement sous forme d'un graphe dont les nœuds représentent les positions accessibles et dont les branches associent à deux nœuds joignables le coût (ou la distance) qui les sépare. Ces algorithmes associent à un chemin en cours d'évaluation un coût total et sélectionnent au final le chemin au coût le moins élevé.

Dijkstra évalue le coût du chemin courant en calculant simplement la somme des coûts depuis le nœud initial. Pour cette raison, il ne privilégie a priori aucune direction d'exploration et son exécution ne se termine que si tous les nœuds accessibles ont été évalués ou si le nœud but a été «visité» [1].

A* rajoute au coût «connu» du chemin reliant l'origine au nœud courant l'évaluation heuristique du coût du chemin reliant le point courant au but. Ceci permet de guider la recherche et de l'arrêter plus tôt si on a confiance en l'heuristique utilisée. [MS08, Pat07]

Les algorithmes classiques de recherche du chemin ne sont pas immédiatement adaptables à un environnement partiellement connu ou en cours d'exploration. Des alternatives existent cependant, parmi lesquelles l'utilisation de l'intelligence en essaim, qui s'inspire en partie des comportements originaux et très efficaces déjà présents dans la nature.

1. Dans le cas de Dijkstra, «visité» signifie que tous les nœuds adjacents ont été évalués.

5.2 Algorithme des fourmis

Cet algorithme est basé sur le comportement de certaines colonies de fourmis qui communiquent indirectement par dépôts de phéromones. Il se présente ainsi : lorsqu'une fourmi trouve une source de nourriture, elle dépose, lors de son retour vers la fourmilière, des phéromones tout au long du chemin qu'elle emprunte. Les autres fourmis suivent cette piste de phéromones pour exploiter la source de nourriture et déposent elles aussi des phéromones sur le chemin du retour, qui peut parfois être différent de la piste suivie initialement.

Comme les fourmis suivent préférentiellement les pistes comportant le plus de phéromones, et puisque les phéromones s'évaporent après un certain laps de temps, le chemin le plus court devient rapidement la piste dominante, car plus de fourmis le traversent et déposent des phéromones par unité de temps. De plus, comme les autres pistes sont alors délaissées, cet algorithme converge très fortement car le chemin le plus court devient rapidement l'unique chemin marqué. [DG97]

Cette approche a été utilisée par [CGN+10], ce qui montre qu'il est possible d'appliquer cet algorithme aux footbots dans ARGoS. Ici, ce sont des messages transmis de robot à robot qui jouent le rôle des phéromones, et la rapidité avec laquelle un seule message traverse toute une chaîne de robot qui indique l'«intensité» d'une piste ; mais les principes fondamentaux restent les mêmes.

Par manque de temps, nous n'avons pas pu implémenter une recherche du chemin efficace en non-omniscient comme ci-dessus, et nous avons choisi de ne pas perdre de temps à implémenter une recherche “classique” car elle serait inutile en non-omniscient, qui est le but final du projet.

Chapitre 6

Construction d'un comportement intelligent

Avec la méthode de déplacement développée au chapitre précédent, nous disposons de la principale «brique» élémentaire nécessaire à la construction d'un comportement intelligent. Pour ce faire, nous allons dans ce chapitre assembler ces briques à un niveau d'abstraction plus élevé, même s'il faudra encore développer quelques outils de plus bas niveau.

6.1 Vue d'ensemble du comportement

Après s'être initialisé, chaque footbot exécute a chaque pas une séquence d'opérations communes à tous les états possibles, suivie d'une séquence d'opérations propre à l'état courant.

Listing 6.1 – Fonction step

```
function step()
  local obstacleProximity, obstacleDirection, onSource,
        foundSource, backHome, gotSource,
        emerProx, emerDir
  obstacleProximity, obstacleDirection, onSource,
  foundSource, backHome, gotSource,
  emerProx, emerDir = doCommon()
  if emerProx>0 then
    emergencyAvoidance(emerProx, emerDir)
  elseif explore then
    doExplore(obstacleProximity, obstacleDirection,
              foundSource, gotSource)
  else
    doMine(obstacleProximity, obstacleDirection,
           onSource, backHome, foundSource)
  end
end
```

6.1.1 Opérations Communes

Les opérations communes à tous les états sont

— L'écoute des capteurs «vitaux» c'est-à-dire qu'ils peuvent complètement écraser l'état courant : capteur de proximité (déclenche l'évitement d'urgence) et batterie (force le footbot à rentrer au nid)

— L'écoute des autres capteurs qui par essence doivent toujours être consultés : capteur de position, de couleur du sol (permet au footbot d'enregistrer des nouvelles sources à tout moment) et communication entre robots (idem).

— Lecture du capteur *distance scanner (short range)* parce qu'il est utilisé en pratique quel que soit l'état du robot.

A part pour l'évitement d'urgence, toutes les décisions sont donc uniquement prises par les fonctions *doMine* et *doExplore*. La fonction *doCommon* se contente de leur envoyer des signaux sous la forme des diverses variables *onSource, backHome, ...* qu'elle renvoie.

L'implémentation détaillée de la fonction doCommon est donnée dans l'annexe 6.A. L'évitement d'urgence est celui présenté dans la section 4.4, tandis que pour retourner le plus rapidement possible au nid, le footbot utilise simplement le déplacement présenté au chapitre 4 avec pour goal le centre du nid.

6.1.2 Exploration

Tous les robots sont initialisés dans l'état *explore*. Dans cet état, ils se déplacent selon une méthode d'exploration donnée jusqu'à recevoir le signal *foundSource* ou *gotSource*. Le premier signal signifie que le footbot a lui-même trouvé une nouvelle source, tandis que le second signal indique qu'une nouvelle source lui a été transmise. C'est la fonction *doCommon* qui s'assure qu'une source potentielle est effectivement originale et la stocke alors dans la liste des sources connues. Cette liste est partagée avec les autres robots (voir section 6.2.1) et utilisée lorsque le robot est dans l'état *exploite*.

Dans le cas où le robot a trouvé une source, il passe directement dans l'état exploite. Par contre, dans le cas où le robot a reçu une nouvelle source, il décide aléatoirement s'il continue à explorer ou s'il se met à exploiter la ou les ressources qu'il connaît. La diffusion de l'information entre robots que nous avons mis en place étant très rapide, ceci permet d'éviter que tous les robots se contentent d'une unique source dès qu'un seul individu en a fait la découverte. Une amélioration possible serait de permettre au robot de passer de l'état *explore* à l'état *exploite* même en l'absence de découverte originale s'il estime avoir déjà passé trop de temps à explorer.

Le parcours d'exploration pris par le robot est une composante importante et complexe de l'efficacité de la phase d'exploration. Des travaux précédents montrent cependant qu'une marche aléatoire (voir [PBP+13] par exemple où

la marche d'exploration est décrite en détail) peut suffire lorsque le nombre de robots est assez élevés, et leurs interactions suffisament développées. Nous nous limitons donc à un simple modèle de diffusion : les robots se déplacent en ligne droite et rebondissent sur tous les obstacles qu'ils rencontrent à la manière de simples particules. Ceci permet d'assurer que les robots migrent des zones les plus concentrées en footbots vers les zones les moins concentrées et que, plus globalement, les robots occupent tout l'espace disponible.

Le modèle utilisé est assez satisfaisant en théorie et en pratique mais n'est pas tout à fait adapté pour plusieurs raisons. Tout d'abord, la gestion de l'autonomie entraîne bien évidemment des retours fréquents des robots vers le nid, ce qui annule en partie l'effet désiré d'homogénéiser la concentration de robots. Ceci devrait être mieux pris en compte. D'autre part, ce modèle permet d'éviter relativement bien l'exploration d'une même zone par un nombre élevé de robots, mais ne permet pas de réellement privilégier d'éventuelles «zones d'ombres» qui n'ont encore jamais été explorées. Cependant, le principal facteur limitant semble bien être la faible portée des capteurs de couleur de sol.

L'implémentation du modèle de diffusion est détaillée en annexe 6.B.

6.1.3 Exploitation

Une fois passé dans l'état *exploite*, le footbot fait simplement des allers-retours entre une des ressources connues et le nid en utilisant les méthodes de déplacement développées au chapitre 4. Les opérations communes ont toujours lieu et le footbot continue donc à communiquer avec les autres robots et à lire la couleur du sol. Il reste donc à traiter du choix de la source à exploiter.

Ce choix est important : il est possible d'augmenter l'efficacité de l'exploitation en incitant les robots à exploiter les sources fournissant le meilleur rendement. Différentes méthodes peuvent être utilisées à cet effet, nous utilisons l'algorithme ε-greedy car il est simple, compréhensible, et utilise une seule variable qui représente directement le degré d'exploration [PBP+13]. Cet algorithme sélectionne avec une probabilité $1 - \varepsilon$ la source au meilleur rendement et avec une probabilité ε une source au hasard parmi les sources moins rentables. Pour ce faire, nous devons pour chaque source mettre en place un indice de qualité qui est réévalué à chaque trajet :

Algorithm 6.1 Mise en place d'un score de qualité des sources

Require: List of available ressources with their quality score
  loop
    Pick ressource to mine using ε-greedy algorithm
    Go to ressource
    Update ressource quality score
    Go to nest
  end loop

Il est en fait plus sensé de mettre à jour l'indice de qualité non pas après un simple aller, mais après un aller-retour complet. Cependant, ceci aurait été plus difficile à mettre en pratique dans notre cas, et nous avons donc choisi la solution présentée ci-dessus. Le score de qualité est simplement donné par le nombre de trajets faits vers une source divisé par la quantité totale de batterie dépensée pour faire ces trajets.

L'utilisation d'une décision ε-greedy nécessitant déjà de trouver le meilleur indice de qualité, une amélioration possible serait de donner la possibilité au footbot de repasser dans l'état *explore* si le meilleur score dans la liste passe en dessous d'une certaine valeur critique. Cependant, il faudrait alors aussi permettre au footbot de changer d'état dans l'autre sens à d'autres occasions et pas uniquement s'il trouve ou reçoit une nouvelle source, comme il a été évoqué dans la section 6.1.2.

L'implémentation détaillée de l'évaluation et du choix des ressources est présentée dans l'annexe 6.C.

6.2 Derniers outils bas niveau

6.2.1 Communication

Vu la structure en boucle demandée par ARGoS, et grâce au pas d'éxécution assez court, les footbots peuvent entièrement partager leur connaissance des ressources sans que la limite de dix Bytes imposée par le système *range and bearing* ne pose problème.

Algorithm 6.2 Partage total de la connaissance des ressources

Require: Known ressources list
Ensure: Total sharing of the ressource knowledge
  loop
    broadcast a different source
  end loop

Dans notre projet, les ressources étant stockées sous forme de liste, les footbots diffusent à chaque pas la i^e source dans la liste où i est le numéro

du pas courant modulo la longueur de la liste des ressources connues.

Une source étant représentée par un couple de nombres à virgule flottante compris entre -500 et 500, la conversion suivante est utilisée pour transmettre l'information :

<table>
<tr><td>Byte 1</td><td>Byte 2</td><td>Byte 3</td><td>Byte 4</td><td>Byte 5</td><td>Byte 6</td><td>Byte 7</td></tr>
<tr><td rowspan="3">type message</td><td rowspan="2">Signe x</td><td rowspan="2">Signe y</td><td>Centaines x</td><td>Centaines y</td><td>Reste x</td><td>Reste y</td></tr>
<tr><td>$\left\lfloor \frac{\lvert x \rvert}{100} \right\rfloor$</td><td>$\left\lfloor \frac{\lvert y \rvert}{100} \right\rfloor$</td><td>round($\lvert x \rvert$ mod 100)</td><td>round($\lvert x \rvert$ mod 100)</td></tr>
<tr><td colspan="5">coordonnées</td></tr>
</table>

Le premier Byte indique comment décoder et interpréter le reste du message. La table ci-dessus n'est donc valide que pour les messages dont les 6 Bytes après le type sont alloués à une position dans l'arène. Dans notre projet, les seuls messages échangés sont de ce type et transmettent la position d'une nouvelle source, mais cette manière de faire est flexible et permet de facilement rajouter d'autres échanges : il suffirait par exemple de changer le type en gardant le même encodage de position pour implémenter la partie communication du ravitaillement entre robots.

Lorsqu'un robot reçoit un message, il le décode en se servant du type du message et de l'encodage ci-dessus, et, dans le cas d'une nouvelle source, vérifie son originalité et la stocke alors en mémoire. Si le robot est déjà en train d'exploiter des ressources, le traitement s'arrête là. Sinon, le footbot doit alors choisir s'il commence à exploiter les ressources ou non, comme décrit dans la section 6.1.2.

L'implémentation en Lua de la communication est détaillée dans l'annexe 6.D.

6.2.2 Gestion de l'autonomie

L'autonomie limitée des footbots leur impose de revenir au nid à intervalles réguliers. Ceci est immédiat à implémenter, à l'exception de la problématique principale : quel indicateur utiliser pour décider de rentrer au nid. Dans notre projet, les footbots rentrent au nid dès qu'ils estiment ne plus avoir assez de batterie pour pouvoir continuer à s'éloigner du nid sans risque. Pour ce faire, nous évaluons simplement la quantité de batterie nécessaire pour rentrer à vol d'oiseau, munie de deux coefficients de sécurité : un coefficient multiplicatif et une réserve de batterie que le robot doit avoir lorsqu'il est de retour au nid. L'indicateur utilisé est donc :

$$\mu_{scrity} \times batt_{step} \times \frac{\sqrt{x^2 + y^2}}{speed} = \delta_{scrity}$$

Afin que les footbots puissent s'adapter à l'environnement dans lequel ils sont placés, μ_{scrity} est constamment réévalué durant l'expérience : à la baisse si un footbot a atteint le nid avec un surplus de batterie $> \delta_{scrity}$ et à

la hausse si le surplus $< \delta_{scrity}$. L'implémentation en Lua de la gestion de l'autonomie est détaillée dans l'annexe 6.5.

6.3 Retour sur le comportement général

Pour conclure ce chapitre, reprenons un point de vue plus large sur le comportement que nous avons construit. La figure 6.1 résume les différents états et transitions traversés par chaque footbot tout au long d'une expérience. Les transitions en pointillés représentent les deux axes d'amélioration que nous avons évoqués aux paragraphes 6.1.2 et 6.1.3.

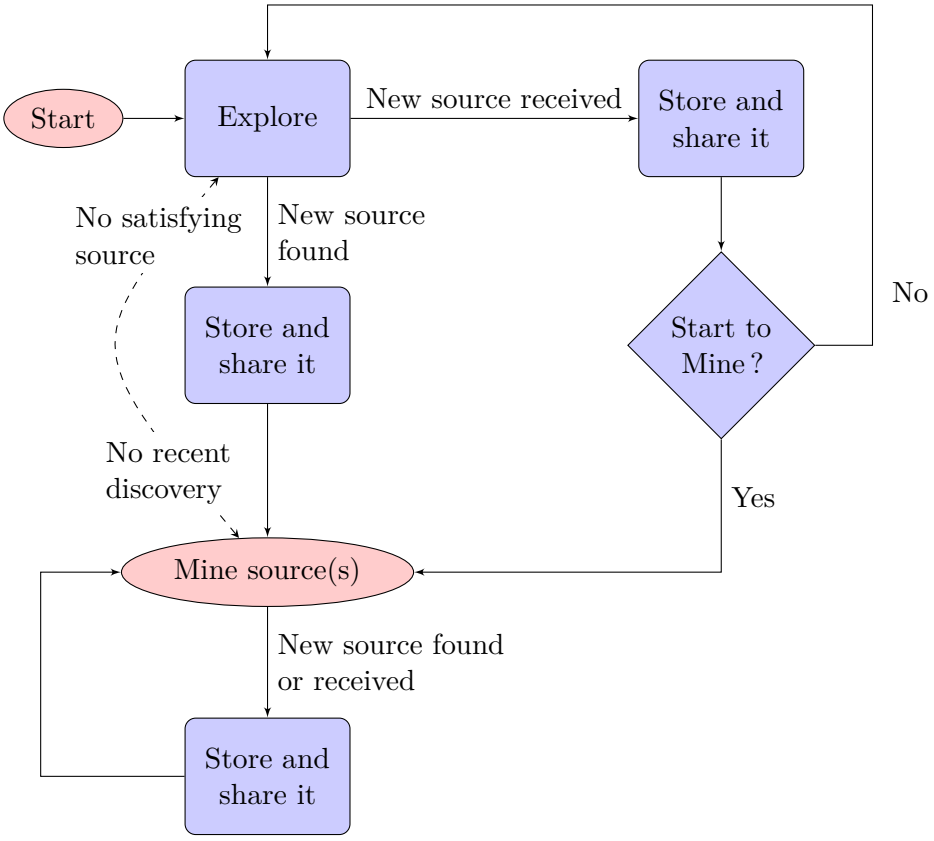


FIGURE 6.1 – Comportement haut niveau de chaque robot [PBP+13].

Il s'agit maintenant de mettre ce comportement à l'épreuve, d'évaluer ses performances, d'examiner quelles possibilités d'optimisation et d'adaptation sont déjà offertes et d'identifier les défauts plus fondamentaux qui sont présents.

Annexe 6.A Implémentation des opérations communes en Lua

6.A.1 Appels faits par *doCommon*

L'implémentation des fonctions appelées est donnée dans les sections traitant de ce que font ces fonctions (fonction *listen* → communication, fonction *closestObstacleDirection* → évitement d'obstacles, ...) à l'exception de *checkGoalReached*, qui détecte de nouvelles sources et qui est présentée ci-dessous.

Listing 6.2 – fonction doCommon

```
function doCommon()
  local obstacleProximity, obstacleDirection,
        onSource, foundSource, backHome, gotSource,
        emerProx, emerDir

  odometry()
  onSource, foundSource, backHome = checkGoalReached()
  if #ressources>=1 then
    broadcastSource(chooseSourceToBroadcast())
  end
  gotSource = listen()
  shortObstaclesTable = updateObstaclesTable("short_range",
                                             shortObstaclesTable)
  obstacleProximity, obstacleDirection
  =closestObstacleDirection(shortObstaclesTable)

  battery=battery-BATT_BY_STEP
  backForBattery = backForBattery
                or battery-batterySecurity
                          *BATT_BY_STEP
                          *math.sqrt(posX^2+posY^2)
                          /BASE_SPEED<IDEAL_NEST_BATT

  emerProx, emerDir=readProxSensor()
  if battery==0 then
    BASE_SPEED=0
    logerr("batt␣empty")
  end
  return obstacleProximity, obstacleDirection,
         onSource, foundSource, backHome,
         gotSource, emerProx, emerDir
end
```

6.A.2 Détection de nouvelle sources et du nid

Listing 6.3 – fonction checkGoalReached

```
function checkGoalReached()
   local foundSource, onSource, backHome=false,false,false
   local insideBlack, seeBlack = floorIsBlack()
   if seeBlack and math.sqrt((posX)^2+(posY)^2)>=90 then
      if insideBlack and sourceIsOriginal(posX,posY, ressources) then
         ressources[#ressources+1]={math.floor(posX),math.floor(posY),
                                    score=1, travels=0,bSpent=0}
         --store source. Initial score=1 which is way above average
         --(Forces footbots to try new sources out)
         foundSource=true
      end
      onSource=true
   elseif  seeBlack and math.sqrt((posX)^2+(posY)^2)<=70 then
      if goalX==0 and goalY==0 then
         backHome=true
      end
      if backForBattery then
         batterySecurity=updateBattCoeff(battery,batterySecurity)
         backForBattery=false
      end
      battery=100
   end
   return onSource, foundSource, backHome
end
```

Listing 6.4 – Détection de la couleur du sol

```
function floorIsBlack()
  local insideBlack = true
  --will tell if robot is fully inside a source or the nest
  local seeBlack = false
  --will tell if robot is touching a source or the nest
  local clr,i
  for i=1,8 do
    clr = robot.base_ground[i].value
    seeBlack = seeBlack or clr== 0
    insideBlack = insideBlack and clr == 0
  end
  return insideBlack, seeBlack
end
```

Listing 6.5 – Vérifier l'originalité de la source

```
function sourceIsOriginal(x, y, rsc)
  local i=1
  local orgn=true
  while i<=#ressources and orgn do
    orgn=(math.sqrt((rsc[i][1]-x)^2 + (rsc[i][2]-y)^2)>ORGN_SRC_DST)
    --just check if it's far enough from sources you already know
    i=i+1
  end
  return orgn
end
```

Annexe 6.B Implémentation de la marche d'exploration en Lua

Listing 6.6 – fonction doExplore

```
function doExplore(obstacleProximity, obstacleDirection,
                   foundSource, gotSource)
  if foundSource then
    explore=false
    goalX,goalY=0,0
  end
  if gotSource then
    if robot.random.uniform()<MINE_PROB_WHEN_SRC_RECVD then
    --choose whether or not you start mining
      explore,goalX,goalY=false,0,0
      --refill battery before starting to mine
    end
  end
  if not backForBattery then
    gasLike(obstacleProximity, obstacleDirection)
  else
    obstaclesTable = updateObstaclesTable("long_range",obstaclesTable)
    move(obstaclesTable, obstacleProximity, obstacleDirection,0,0)
  end
end
```

Listing 6.7 – Déplacement *gaslike*

```
function gasLike(obstacleProximity, obstacleDirection)
  local goalAngle
  if obstacleProximity < 30
     and not(obstacleDirection<-PI/2 or obstacleDirection>PI/2)
     and not wasHit then
    wasHit = true --/!\Not local: allows footbot to remember
                  --if it's already trying to reach an angle
    newDirection = alpha+rebound(alpha,obstacleDirection)
    newDirection=setCoupure(newDirection) --sets angle in [-PI,PI]
  end
  if wasHit then
    if abs(alpha-newDirection)<0.2 then --checks if angle is reached
      wasHit=false
    else
      goalAngle=newDirection-alpha
      goalAngle=setCoupure(goalAngle) --sets angle in [-PI,PI]
      getToGoal(goalAngle, EXPL_CONV)
      --EXPL_CONV is really high because we want to do sharp turns
    end
  else
    robot.wheels.set_velocity(BASE_SPEED,BASE_SPEED)
  end
end
```

Listing 6.8 – Calcul de l'angle "réfléchi"

```
function rebound(alpha, obstacleDirection)
  if obstacleDirection<=12 then --obstacle is to the left
    newAngle = -2*(PI/2-obstacleDirection)
  else
    newAngle = 2*(PI/2-obstacleDirection)
  end
  newAngle=setCoupure(newAngle) --sets angle in [-PI,PI]
  return newAngle
end
```

Annexe 6.C Implémentation du choix de ressource à exploiter en Lua

Listing 6.9 – fonction doMine

```
function doMine(obstacleProximity, obstacleDirection, onSource, backHome)
   if onSource then
      if not hasMined then
         ressources[sourceId].travels=ressources[sourceId].travels +1
         travels=travels+1
         evalSource(sourceId, battery)
         goalX,goalY=0,0
      end
      hasMined=true
   elseif backHome then
      if hasMined then
         hasMined=false
      end
      sourceId,goalX,goalY=chooseNewSource(ressources)
   elseif backForBattery then
      if goalX~=0 and goalY~=0 then
         goalX,goalY=0,0
         evalSource(sourceId, 0)
      end
   end
   obstaclesTable = updateObstaclesTable("long_range",obstaclesTable)
   move(obstaclesTable, obstacleProximity, obstacleDirection,goalX,goalY)
end
```

Listing 6.10 – Choix de la ressource à exploiter

```
function chooseNewSource(rsc)
   local newSourceId
   if #rsc>1 then
      placeMaxAtOne(rsc)
      local pickBest=robot.random.uniform()
      if pickBest<(1-EPSILONGREED) then
         newSourceId=1
      else
         newSourceId=robot.random.uniform_int(2,#rsc+1)
      end
   else
      newSourceId=1
   end
   local x=rsc[newSourceId][1]
   local y=rsc[newSourceId][2]
   return newSourceId, x, y
end
```

Listing 6.11 – Evaluation de la qualité de la source

```
function evalSource(sourceId, battery)
   ressources[sourceId].bSpent=ressources[sourceId].bSpent+(100-battery)
   ressources[sourceId].score=ressources[sourceId].travels
                            /ressources[sourceId].bSpent
end
```

Annexe 6.D Implémentation de la communication en Lua

6.D.1 Envoi de message

Listing 6.12 – Envoi d'un message différent à chaque pas

```
function doCommon()
  ...
  if #ressources>=1 then
    broadcastSource(chooseSourceToBroadcast())
  end
  ...
end

function broadcastSource(x,y)
  local msg=sourceIn(x,y)
  robot.range_and_bearing.set_data(msg)
end

function chooseSourceToBroadcast()
  local i=(currentStep%#ressources)+1
  return ressources[i][1],ressources[i][2]
end
```

Listing 6.13 – Encodage du message

```
function sourceIn(x,y)
  local msgOut={1,sgnIn(x),sgnIn(y),
                math.floor(abs(x)/100),math.floor(abs(y)/100),
                math.floor(abs(x)%100),math.floor(abs(y)%100),
                0,0,0}
  return msgOut
end

function sgnIn(n)
   local sgn
   if n==0 then
      sgn=0
   else
      if n==abs(n) then
         sgn=1
      else
         sgn=2
      end
   end
   return sgn
end
```

6.D.2 Réception de message

Listing 6.14 – Ecoute du capteur à chaque pas

```
function doCommon()
  ...
  gotSource = listen()
  ...
end

function listen()
  local gotSource=false
  for i=1,#robot.range_and_bearing do
    if robot.range_and_bearing[i].data[1]==1 then
      local source = sourceOut(robot.range_and_bearing[i].data)
      if sourceIsOriginal(source[1],source[2],ressources) then
      --Check for originality
        source.score=1
        source.travels=0
        source.bSpent=0
        ressources[#ressources+1]=source
        gotSource=true
      end
    end
  end
  return gotSource
end
```

Listing 6.15 – Décodage du message

```
function sourceOut(msg)
   local x,y
   x=100*msg[4]+msg[6]
   if msg[2]==2 then
      x=-x
   end
   y=100*msg[5]+msg[7]
   if msg[3]==2 then
      y=-y
   end
   return {x,y}
end
```

6.5 Implémentation de la gestion de l'autonomie en Lua

Listing 6.16 – appel par doCommon

```
function doCommon()
  ...
  battery=battery-BATT_BY_STEP
  backForBattery = backForBattery
                or battery-batterySecurity
                          *BATT_BY_STEP
                          *math.sqrt(posX^2+posY^2)
                          /BASE_SPEED<IDEAL_NEST_BATT
  --We use the "or" so that backForBattery stays true once
  --it has been triggered. Only checkGoalReached can set
  --it back to false
  ...
end
```

Listing 6.17 – Réévaluation de batterySecurity

```
function checkGoalReached()
  ...
  elseif seeBlack and math.sqrt((posX)^2+(posY)^2)<=70 then
  ... --You're back in the nest
    if backForBattery then
      batterySecurity=updateBattCoeff(battery,batterySecurity)
      backForBattery=false
    end
    battery=100
  end
  ...
end

function updateBattCoeff(battery, batterySecurity)
  if battery>IDEAL_NEST_BATT then
    batterySecurity=batterySecurity-(battery-IDEAL_NEST_BATT)*.07
  else
    batterySecurity=batterySecurity-(battery-IDEAL_NEST_BATT)*.1
  end
  return batterySecurity
end
```

Chapitre 7

Résultats obtenus

Le comportement créé permet de remplir tous les objectifs énoncés par le cahier des charges, à savoir l'exploration d'un environnement inconnu dans le but d'y repérer des «sources» marquées par des taches noires au sol et l'exploitation de ces sources, tout en évitant les obstacles éventuels et en gérant adéquatement l'autonomie des robots. De plus, l'arène a aussi dû être créée afin de convenir à l'expérience ainsi qu'aux spécifications données. Enfin, comme demandé, une évaluation des performances du comportement a été faite. Cette évaluation est présentée ci-dessous.

7.1 Evaluation des performances

7.1.1 Efficacité de l'exploitation

Nous avons d'abord évalué les performances de l'exploitation seule, en donnant aux robots la connaissance des sources présentes. Les indicateurs utilisés sont le nombre moyens de trajets totaux et par source après 10.000 steps, en fonction du nombre de robots, et le nombre de trajets totaux par robot après 10.000 steps. Ceci permet de juger de l'efficacité du déplacement mais aussi du choix de ressource à exploiter. A cet effet, nous utilisons volontairement une répartition de source assez déséquilibrée.

On voit que les robots sont effectivement capables de distinguer la source la plus rentable, mais que l'indice de qualité d'une source utilisé par la décision ε-greedy est peut-être inadapté. En effet, lorsqu'on augmente le nombre de robots, on assiste à une meilleure répartition des robots sur les trois sources mais ce n'est pas suffisant parce que les robots continuent à utiliser massivement la source A la plus proche, au point que celle-ci n'est visiblement plus rentable car les robots embouteillent complètement la zone entre le nid et cette source. La conséquence directe est que le rendement par robot décroît énormément lorsqu'on augmente le nombre de footbots.

Le pourcentage de robots tombés en panne de batterie au bout de 10.000 steps nous aide à mieux mettre ce phénomène en évidence. Les pannes qui

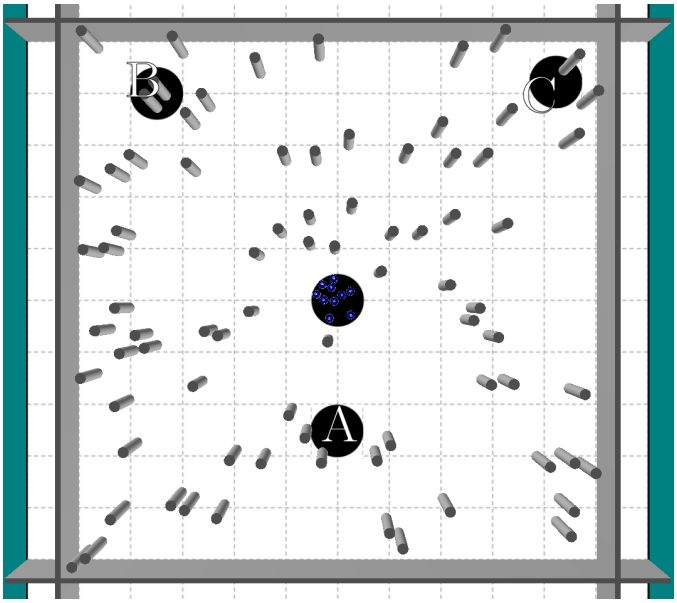
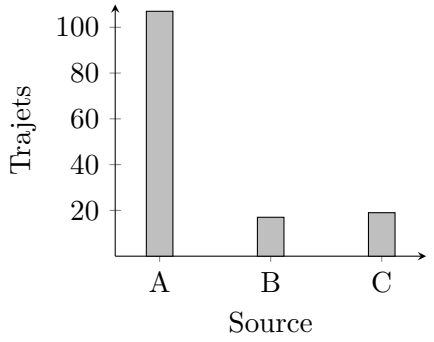
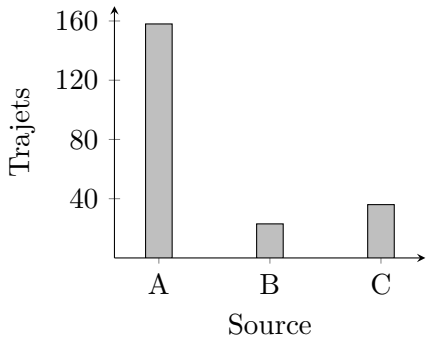


FIGURE 7.1 – Configuration utilisée et référence des sources

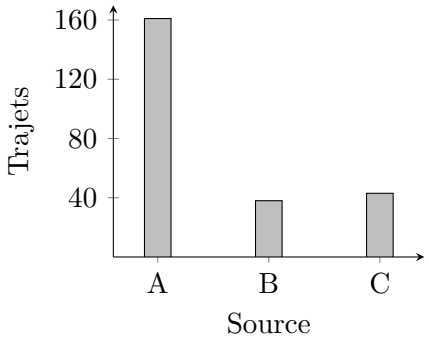
apparaissent lorsqu'on augmente le nombre de footbots concernent majoritairement des robots qui cherchaient à exploiter la source A et qui se sont retrouvé complètement bloqués sur le chemin.



(a) 5 footbots



(b) 10 footbots



(c) 15 footbots

FIGURE 7.2 – Nombre de trajets effectués par source au bout de 10.000 steps, selon le nombre de robots

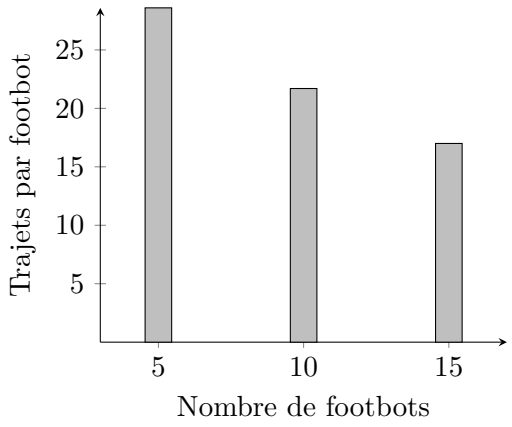


FIGURE 7.3 – Nombre de trajets effectués par footbot au bout de 10.000 steps, selon le nombre de footbots

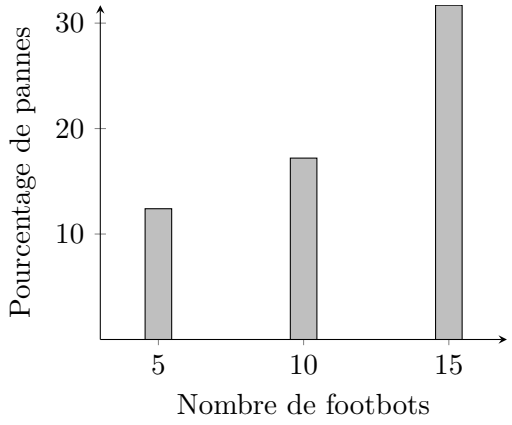
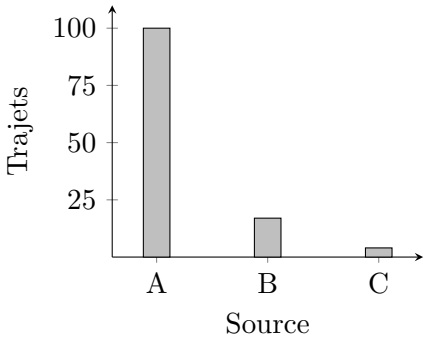


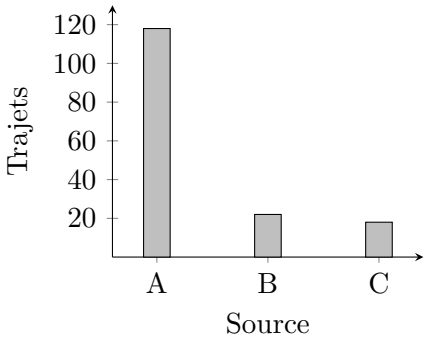
FIGURE 7.4 – Pourcentage de pannes de batterie après 10.000 steps, selon le nombre de robots

7.1.2 Performances Générales

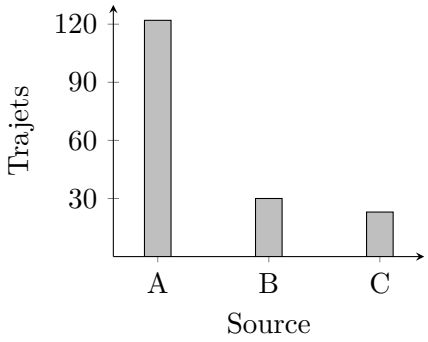
On voit sur la dernière figure qu'à la fin de l'expérience, les footbots atteignent des rendements comparables à ceux qu'ils avaient en omniscient. On remarque aussi que deux phénomènes se contrebalancent : d'une part, le fait que l'exploration s'améliore avec le nombre de robots, et d'autre part le fait que le rendement individuel moyen diminue avec le nombre de robots. Ceci explique probablement pourquoi le rendement maximum est atteint pour 10 footbots et diminue ensuite.



(a) 5 footbots



(b) 10 footbots



(c) 15 footbots

FIGURE 7.5 – Nombre de trajets effectués par source au bout de 10.000 steps, selon le nombre de robots

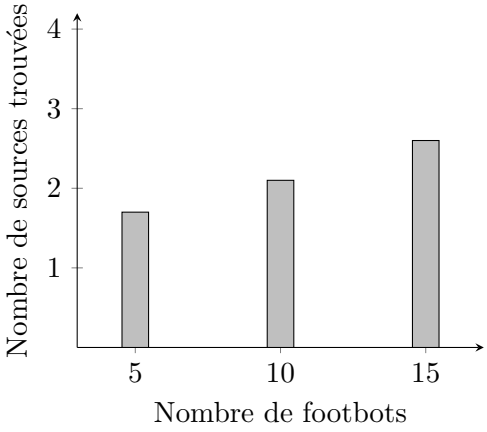


FIGURE 7.6 – Nombre moyens de source trouvées après 10.000 steps, selon le nombre de robots [a]

a. Les footbots peuvent trouver deux sources «différentes» sur la même tache réelle si ces sources sont assez espacées (voir 6.5)

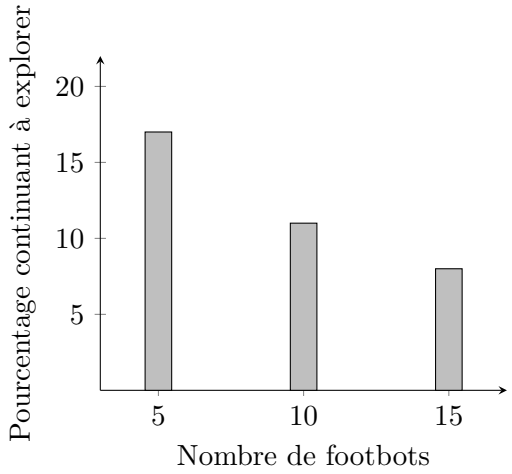


FIGURE 7.7 – Pourcentage moyen de footbots continuant à explorer l'environnement après 10.000 steps, selon le nombre de robots

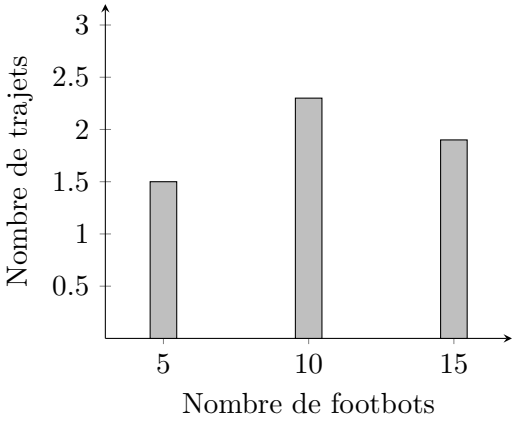


FIGURE 7.8 – Nombre moyens de trajets par footbot faits sur les 1000 steps suivant, après 10.000 steps, selon le nombre de footbots

7.2 Perspectives d'amélioration

7.2.1 Optimisation des constantes utilisées

Tout au long de ce rapport, différents paramètres numériques ont été mis en avant et le comportement Lua a aussi été écrit de manière à ce que ces valeurs soient facilement accessibles. Il en résulte un ensemble de variables globales qui régissent toutes un aspect du comportement du robot dans l'expérience.

Listing 7.1 – Définitions des variables globales

```
--Specs received
BASE_SPEED=30
BATT_BY_STEP =.2

--Low-level
SCANNER_RPM=75
DIR_NUMBER = 15
EXPL_DIR_NUMBER = 20
EXPL_CONV = 3

--"Mid"-level:Movement
CONVERGENCE=1
OBSTACLE_PROXIMITY_DEPENDANCE=.25
OBSTACLE_DIRECTION_DEPENDANCE=.25
EMER_DIR_DEP=1
EMER_PROX_DEP=1
MIN_SPEED_COEFF = 0.6
--When a footbot "hits" something, he will pick a
--temporary speed between this coeff and 1 times BASE_SPEED
RANDOM_SPEED_TIME = 30
--The number of steps during which
--the footbot keeps this new random speed

--High-level:Decision Making
ORGN_SRC_DST=80
--Minimal distance between two sources considered "different"
MINE_PROB_WHEN_SRC_RECVD=.2
--Probability of starting mining upon receiving a new source
INIT_BATT_SEC=25
--Initial battery handling security coeff
IDEAL_NEST_BATT=20
--Leftover battery a footbot should have when returning to the nest
EPSILONGREED=0.1
--epsilon for epsilon-greedy choice algorithm
```

L'étape suivante consisterait à trouver une combinaison de ces variables qui maximise la performance des robots. Ceci est bien évidemment une problématique très vaste, d'une part parce que cette combinaison optimale de valeurs dépend de la manière dont la performance est mesurée et de la configuration d'expérience utilisée, et d'autre part parce que le nombre

de ces variables est trop déraisonnablement élevé pour se lancer dans une telle optimisation sans d'abord essayer de simplifier le problème. Dans cette section, nous allons brièvement rappeler les paramètres qui sont apparus, montrer quelles sont les constantes qui jouent les rôles les plus importants et quelles possibilités immédiates d'adaptation notre comportement offre à travers la modification de ces variables globales.

Les constantes sont réparties en trois groupes : les constantes «bas-niveau», les constantes utilisées par les différents algorithmes de déplacement, et les constantes «haut-niveau» qui interviennent dans la prise de décision. Parmi les constantes «bas-niveau», on trouve la vitesse angulaire du *distance scanner*, le nombre de directions dans lesquelles les mesures de ce capteur sont regroupées ainsi que la convergence κ utilisée pour les «rebonds» lors de l'exploration [1]. Ces constantes jouent un rôle mineur et sont plutôt liées à la physique des footbots.

Dans les constantes liées au mouvement, on retrouve la convergence κ utilisée dans le reste des situations ainsi que les deux paires de paramètres α et β qui sont apparus dans l'évitement d'obstacles proches en 4.4 et l'évitement intermédiaire en 4.5 [2]. On retrouve aussi les paramètres utilisés pour introduire une composante aléatoire à l'évitement dans l'annexe 4.B. Ces constantes influent sur la qualité générale du mouvement à travers toute l'expérience, et il existe probablement une plage de valeurs qui optimisent le mouvement dans l'écrasante majorité des configurations. En d'autres termes, elles ne permettent pas d'adapter le comportement des footbots d'une situation à l'autre.

Dans les constantes «haut niveau» liées à la prise de décision, on retrouve la distance minimale entre deux sources pour qu'elles soient considérées comme originales, la probabilité qu'un footbot commence à miner les ressources qu'il connaît lorsqu'il reçoit une nouvelle source, le coefficient de sécurité initial μ_{scrity} utilisé dans la gestion de l'autonomie, le résidu de batterie idéal δ_{scrity} que doit avoir le robot lorsqu'il rentre au nid pour se ravitailler, ainsi que la valeur du ε utilisé dans la décision ε-greedy. Ces variables ont une influence majeure sur le comportement final des robots. Tout travail d'optimisation ultérieur devrait se pencher en priorité sur ces constantes-ci.

Parmi ces variables, il est cependant possible de mettre `INIT_BATT_SEC` à part car comme son nom l'indique il ne s'agit que du coefficient de sécurité initial puisqu'il est ensuite réévalué constamment durant l'expérience. Il suffit donc de lui donner une valeur largement supérieure à la normale et celle-ci sera abaissée au cours de l'éxécution. Pour jouer sur la prise de risque liée à la gestion de la batterie, il faut utiliser `IDEAL_NEST_BATT`.

1. ▲voir listing 6.7.
2. ▲Pour rappel, ces deux évitements sont les mêmes aux constantes et à la portée des capteurs utilisés près

`ORGN_SRC_DST` et `MINE_PROB_WHEN_SRC_RECVD` sont assez étroitement liées. L'influence de `ORGN_SRC_DST` est assez claire, mais lorsque cette constante est modifiée, il faut penser à modifier `MINE_PROB_WHEN_SRC_RECVD` en parallèle. En effet, pour un même nombre de sources «réellement différentes» abaisser la première de ces constantes augmente le nombre de sources «différentes selon les robots» à découvrir et il est donc judicieux d'augmenter par la même occasion la tendance à l'exploration par le biais de `MINE_PROB_WHEN_SRC_RECVD`. De même, il peut être intéressant de jouer sur cette dernière constante si l'on dispose d'informations sur le nombre de sources réelles ou selon le but de l'expérience : rendement court-terme, rendement long-terme, découverte d'une proportion maximale des ressources, ... Cependant, l'importance de ces deux variables peut-être réduite si l'on implémente les transitions supplémentaires proposées dans le diagramme 6.1.

Enfin, l'influence de ε dans la décision ε-greedy est assez claire et bien documentée (voir par exemple [PBP+13].). Etant donné les résultats des études statistiques, une valeur plus élevée que celle utilisée (0.1) semble peut-être judicieuse, à moins qu'une refonte totale du choix de la source à exploiter ne soit nécessaire.

7.2.2 Améliorations fondamentales

De nombreuses améliorations et fonctionnalité supplémentaires ont déjà été mentionnée au cours de ce rapport. Ainsi, le passage de l'état *explore* à *exploite* semble le plus problématique, et nous avons évoqué la possibilité de transitions supplémentaires, notamment vers *explore* si l'indice de qualité des sources semble trop mauvais (voir sections 6.1.2 et 6.1.3).

Dans le but de créer un comportement intelligent, il faudrait essayer d'éliminer le plus possible les variables globales mentionnées ci-dessus et rajouter des possibilités d'adaptation par lui-même du footbot à son environnement. D'autre part, il serait bien sûr intéressant d'adapter des méthodes issues de la swarm-intelligence au problème, comme par exemple l'algorithme des fourmis présenté en 5.2.

Enfin, les tests ont montré que les algorithmes de choix de la source à exploiter et de gestion de l'autonomie pouvaient être nettement améliorés. Ainsi, la gestion de l'autonomie demande peut-être une approche plus «subtile» pour tenir compte des obstacles qu'une simple utilisation de coefficients de sécurité.

Chapitre 8

Fonctionnement du groupe

Le groupe est composé de 5 membres. Chaque membre possède sa manière de travailler, de comprendre, de communiquer. Une des difficultés d'un travail d'équipe est de pouvoir combiner tous ces caractères pour que le projet se déroule dans les meilleures conditions et que chacun puisse trouver sa place. Donc pour comprendre le fonctionnement de chacun, chaque membre a dû présenter ses points forts et ses points faibles. Le but de cette démarche est de pouvoir répartir les tâches au mieux tout en privilégiant le transfert de compétences. Ceci a été fait par exemple lors de la présentation orale formative de mi-parcours, où les membres qui se sentaient le moins à l'aise dans l'exercice ont choisi de faire cette présentation afin de pouvoir profiter de l'opportunité.

Organisation et communication

La répartition des tâches se fait spontanément en fonction des besoins pour l'avancée du projet ainsi que des membres qui ont travaillé sur des tâches similaires dans le passé. Ceci offre l'avantage d'une meilleure gestion du temps lors des réunions car les membres sont au courant des problèmes à régler. Cependant un diagramme de Gantt (cf. annexe) a été créé pour aussi avoir une vision plus globale de l'avancement du projet et également se fixer des dates butoir.

A côté des moyens de communication habituellement utilisés tels que l'utilisation de mails et de réseau sociaux, des outils plus spécifiques ont été mis en place afin d'améliorer l'implication des membres et l'efficacité du travail collectif. Tout d'abord, un dépôt git a été créé [1] afin de pouvoir profiter de tous les avantages qu'offre un contrôleur de version distribué : sécurité, travail simultané sur plusieurs aspects du code tout en maintenant une version parfaitement fonctionelle, possibilité de consulter l'historique, ... Nous avons essayé de passer systématiquement par l'usage de branches afin de nouveau

1. https://github.com/nathdwek/projetBa2 et https://github.com/nathdwek/rapportProjetBa2

de gagner en sécurité, ce qui permet à n'importe quel membre d'essayer sans crainte et sans perte de temps d'implémenter une fonctionnalité ou de corriger une erreur éventuelle. D'autre part, nous avons utilisé Zotero pour la mise en commun, l'uniformisation et l'export en format Bibtex des ressources bibliographiques.

FIGURE 8.1 – Réseau de commits récents du projet

Algorithmes

Listings

Figures

Bibliographie

[Bac14] Exponential Backoff. Exponential backoff — Wikipedia, the free encyclopedia, feb 2014. Available from : http://en.wikipedia.org/w/index.php?title=Exponential_backoff&oldid=594243345.

[cah13] Cahier des charges project BA 2 – robots en essaim : explorez !, October 2013.

[CGN+10] Alexandre Campo, Álvaro Gutiérrez, Shervin Nouyan, Carlo Pinciroli, Valentin Longchamp, Simon Garnier, and Marco Dorigo. Artificial pheromone for path selection by a foraging swarm of robots. *Biological Cybernetics*, 103(5) :339–352, 2010. Available from : http://link.springer.com/article/10.1007%2Fs00422-010-0402-x.

[DG97] Marco Dorigo and Luca M. Gambardella. Ant colony system : A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1) :53–66, 1997.

[Dor07] Marco Dorigo. Swarm intelligence, oct 2007. Available from : http://www.scholarpedia.org/article/Swarm_Intelligence.

[LD06] Pierre Lambert and Alain Delchambre. *Mécanique rationelle II*. Presses Universitaire de Bruxelles, Bruxelles, 5ème edition, 2006.

[MAU13] Michel MAUNY. Langages de scripts. *Techniques de l'ingénieur Langages de programmation*, base documentaire : TIB304DUO.(ref. article : h3118), 2013. fre. Available from : http://www.techniques-ingenieur.fr/base-documentaire/technologies-de-l-information-th9/langages-de-programmation-42304210/langages-de-scripts-h3118/.

[MS08] Kurt Mehlhorn and Peter Sanders. Graph traversal and shortest paths. In *Algorithms and Data Structures*, pages 175–215. Springer Berlin Heidelberg, January 2008. Available from : http://link.springer.com/chapter/10.1007/978-3-540-77978-0_10.

[Pat07] Srikanta Patnaik. Path planning and navigation using a genetic algorithm. In *Robot Cognition and Navigation*, Cognitive Technologies, pages 39–76. Springer Berlin Heidelberg, January 2007.

Available from : http://link.springer.com/chapter/10.1007/978-3-540-68916-4_4.

[PBP+13] Giovanni Pini, Arne Brutschy, Carlo Pinciroli, Marco Dorigo, and Mauro Birattari. Autonomous task partitioning in robot foraging : an approach based on cost estimation. *Adaptive Behavior*, 21(2) :118–136, 2013. Available from : http://iridia.ulb.ac.be/~cpinciroli/pdf/Pini:AB2013.pdf.

[Pin] Carlo Pinciro. Swarm intelligence course (INFO-H-414). Available from : http://iridia.ulb.ac.be/~cpinciroli/extra/h-414/.

[PTO+11] Carlo Pinciroli, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle, Timothy Stirling, Álvaro Gutiérrez, Luca Maria Gambardella, and Marco Dorigo. ARGoS : a modular, multi-engine simulator for heterogeneous swarm robotics. Technical Report TR/IRIDIA/2011-009, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium, 2011. Available from : http://iridia.ulb.ac.be/~cpinciroli/pdf/Pinciroli:TechRep2011009.pdf.

[RND10] Stuart J Russell, Peter Norvig, and Ernest Davis. *Artificial intelligence : a modern approach*. Prentice Hall, Upper Saddle River, NJ, 2010.

[SER13] Manuel SERRANO. Langage c++. *Techniques de l'ingénieur Langages de programmation*, base documentaire : TIB304DUO.(ref. article : h3078), 2013. fre. Available from : http://www.techniques-ingenieur.fr/base-documentaire/technologies-de-l-information-th9/langages-de-programmation-42304210/langage-c-h3078/.

[Sha07] Amanda J. C. Sharkey. Swarm robotics and minimalism. *Connection Science*, 19(3) :245–260, 2007. Available from : http://www.tandfonline.com/doi/abs/10.1080/09540090701584970.