



ÉCOLE 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

16 mars 2014

Abstract

The goal of this project was to design a swarm-intelligent behaviour for virtual robots using the ARGoS software. 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. However, a Dijkstra path-finding algorithm, which would only execute once before every trip while always providing the shortest trajectory, was also considered. A first 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 loop between a starting area and spots of known position while avoiding collisions, and information was gathered on parameters meaningful for the experiment. Future development should be focused on optimizing these parameters and enabling robots to explore a fully unknown environment.

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. Cependant, une recherche du plus court chemin Dijkstra, qui ne serait faite qu'une seule fois avant chaque trajet d'un robot, tout en trouvant toujours le chemin le plus court, fut aussi considéré. Une première 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 robot de faire des allers-retours entre leur nid et une ou plusieurs ressources dont ils connaissent la position à l'avance, tout en évitant les collisions dans un environnement inconnu. De plus, des informations ont été collectées sur des paramètres influençant l'expérience. La deuxième partie du quadrimestre devrait être consacrée à l'optimisation de cette solution et à permettre au robot d'explorer un environnement afin de trouver la position des sources à exploiter.

Table des matières

1	Introduction	3
1.1	Intérêt du projet	3
1.2	Résultats attendus	4
2	L'intelligence artificielle	5
2.1	Algorithme des fourmis	6
3	ARGoS et les footbots	7
3.1	Capteurs et actuateurs des footbots	7
3.2	Choix du langage informatique	8
3.3	Structure d'un comportement en Lua	8
4	Déplacement et évitement d'obstacles	10
4.1	Aspects physiques du déplacement	10
4.2	Déplacement sans obstacles	11
4.3	Évitement <i>greedy</i> d'obstacles lointains	12
4.4	Évitement sûr d'obstacles proches	13
4.5	Évitement intermédiaire	14
4.6	Déplacement selon un chemin précalculé	14
4.6.1	Recherche du plus court chemin	15
Annexe 4.A	Implémentation du déplacement en Lua	18
4.A.1	Évitement <i>greedy</i>	18
4.A.2	Évitement d'obstacles proches	20
Annexe 4.B	Ajout d'une composante aléatoire à l'évitement	22
5	Comportement	23
5.1	Opérations Communes	23
5.2	Exploration	24
5.3	Exploitation	25
5.4	Communication	26
5.5	Gestion de l'autonomie	27
Annexe 5.A	Implémentation des opérations communes en Lua	29
5.A.1	Appels faits par <i>doCommon</i>	29
5.A.2	Détection de nouvelles sources et du nid	29

Annexe 5.B	Implémentation du modèle de diffusion en Lua	31
Annexe 5.C	Implémentation du choix de ressource en Lua	33
Annexe 5.D	Implémentation en Lua de la communication	35
5.D.1	Envoi de message	35
5.D.2	Réception de message	35
6	Résultats obtenus	37
7	Fonctionnement du groupe	38
7.1	Général	38
7.2	Organisation	38
7.3	Communication	38
8	Conclusion	40
	Algorithmes	41
	Listings	42
	Table des figures	43
	Bibliographie	45

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 oeuvre 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 infailible, 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 trois parties : une introduction à la notion d'intelligence artificielle et de comportement en essaim en particulier, la problématique du déplacement des robots et enfin la communication et prise de décision.

Chapitre 2

L'intelligence artificielle

Afin de mener à bien l'objectif énoncé dans l'introduction, les robots doivent se comporter de manière intelligente. Le plus important est l'interaction avec l'environnement. En effet, le choix de la bonne action est cruciale. Il est cependant délicat de définir une seule «meilleure» action. Ceci est développé plus bas, en introduisant le concept de *rationalité*. De manière générale, un robot peut être assimilé à un agent qui reçoit des *percepts* par l'intermédiaire de ses capteurs. L' *agent* réagit alors en exerçant une action sur l'environnement grâce à ses *effecteurs*. La description des différents

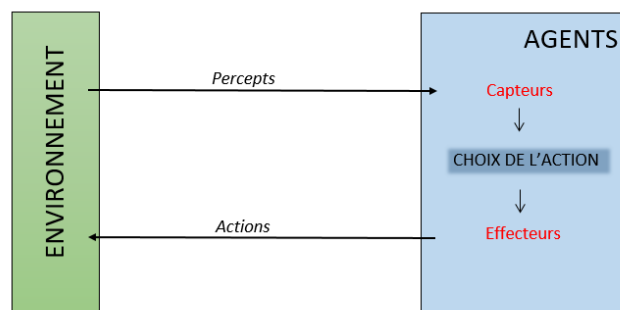


FIGURE 2.1 – Cycle d'interaction entre l'environnement et les agents

capteurs et effecteurs du robot est faite dans le chapitre 3.

Comme énoncé précédemment la rationalité d'un agent peut être délicate 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.»

En effet, dans un groupe composé de plusieurs 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.

A partir de cela, tout problème peut être formellement défini par cinq composantes. Tout d'abord l'état initial dans lequel commence l'agent, puis la description de ses différentes actions, c'est à dire toutes les actions possibles dans un état donné. Ensuite, vient le modèle de transition, il décrit ce que chaque action réalise. Ces trois premières composantes définissent 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.

Dès lors l'espace d'état peut être interprété sous forme d'arbre où les nœuds représentent des états et les branches des séquences d'action. Il en découle la notion de chemin représentant une séquence d'états reliés par une séquence d'action.

La quatrième composante correspond au test but. Celle-ci détermine si un état donné est un état but. Enfin vient la cinquième et dernière composante, le coût du chemin. Elle permet d'attribuer une valeur numérique à un chemin en accord avec la mesure de performance imposée.

Maintenant qu'une présentation générale de l'intelligence artificielle a été faite, le comportement d'essaim d'agents intelligents peut être étudié. Pour ce faire, il s'est avéré très intéressant d'en comprendre le comportement à partir d'exemples se trouvant dans la nature. Les fourmis et les abeilles illustrent bien cela. Deux algorithmes mettant en avant leur comportement sont présentés ci-dessous.

2.1 Algorithme des fourmis

Cet algorithme est basé sur le comportement des fourmis dont une des particularités est la communication au travers de l'environnement par dépôts de phéromones.

L'algorithme se présente de la manière suivante. Pour commencer, une exploration de l'environnement est faite par les fourmis. Si l'une d'entre elles trouve une source, elle déposera, lors de son retour au nid, des phéromones tout au long du chemin qu'elle emprunte. Dès lors, lorsque que d'autres fourmis partiront à la recherche de nourriture, elles auront tendance à suivre le chemin marqué de phéromones. Et à leur retour à la colonie, elles renforceront cette piste.

Cet algorithme illustre une communication indirecte d'un essaim et la manière dont ce dernier est influencé [DG97]. L'autre algorithme, illustrant le comportement d'une structure organisée, est l'algorithme des abeilles.

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.

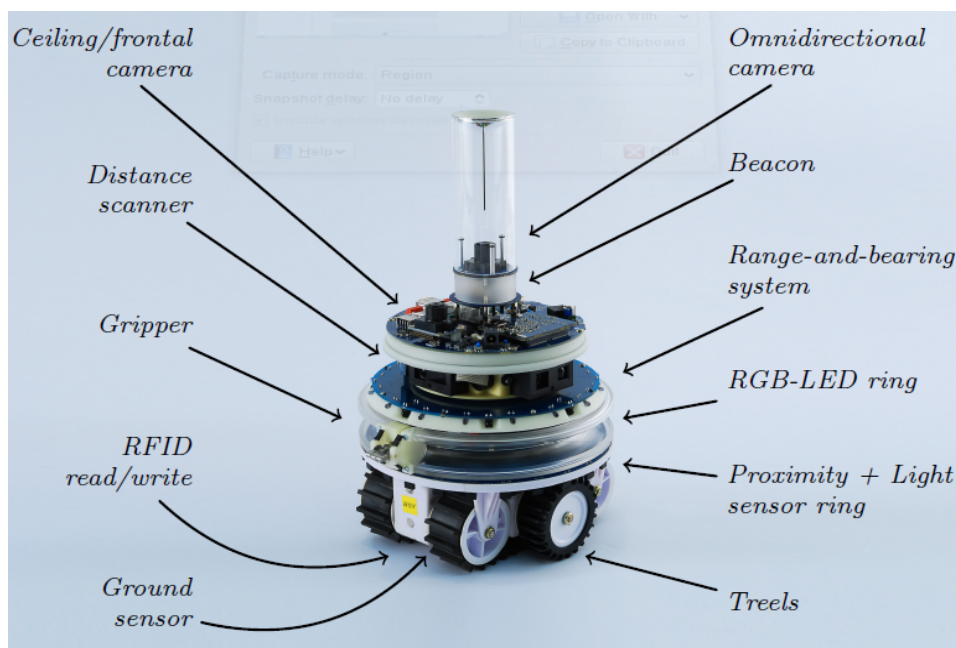


FIGURE 3.1 – Les différents senseurs et actuateurs d'un footbot [Pin]

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 intéres-

sants sont donc 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 actionneurs 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++, notamment, sa rapidité d'exécution et ses nombreuses bibliothèques, 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 celles exigées par ARGoS. 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 presque entièrement le comportement du robot. Deux types d'événements 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
```

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. Dans ce chapitre-ci, l'un des effecteurs primordiaux d'un footbot sera examiné : son déplacement.

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

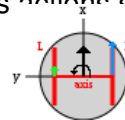


FIGURE 4.1 – Signification physique de v_r , v_l , v_g et ω_g [Pin]

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 :

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

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 son but selon une trajectoire souple et à vitesse constante.

Algorithm 4.1 Convergence simple sans évitement d'obstacle

Require: $SPEED \equiv$ Fixed speed of the footbot > 0

Require: Goal in arena

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

$right\ velocity \leftarrow convergence(\theta, SPEED)$

$left\ velocity \leftarrow 2 \times SPEED - right\ velocity$ \triangleright 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 :

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

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 s'intègre parfaitement 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 très efficace mais parfois trop gourmand consiste alors à chercher à chaque pas la direction la plus proche de la direction du goal et pour laquelle 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
  right velocity  $\leftarrow$  convergence( $\theta$ , SPEED)
  left velocity  $\leftarrow 2 \times \textit{SPEED} - \textit{right velocity}$   $\triangleright$ so that overall speed
  stays equal to SPEED
end while

```

La fonction convergence doit satisfaire au 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. Malgré cela, cet évitement reste très *greedy*, dans le sens où chercher absolument la direction la plus proche de celle du but peut mener à des collisions à cause de différents facteurs et approximations (footbot de taille non nulle, rayons du *distance scanner* parfaitement tangent 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 à 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 lorsqu'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

Ensure: footbot converges towards the goal at speed *SPEED* while avoiding obstacles
while goal not reached **do**
 update footbot position and orientation
 read proximity sensors ▷ or whatever other sensor in use
 if no obstacles too close **then**
 do previous convergence
 else
 $right\ velocity \leftarrow avoidance(\text{proximity sensor reading}, SPEED)$
 end if
 $left\ velocity \leftarrow 2 \times SPEED - right\ velocity$ ▷ so that overall speed stays equal to *SPEED*
end while

Où $avoidance(\text{proximity sensor reading}, SPEED)$ 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 à 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 fourni 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 aux constantes numériques près (évitement moins brusque) que ci-dessus. Cet évitement est beaucoup moins *greedy* 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 pas¹. 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 à aux différentes routines élémentaires présentées ci-dessus, il est donc possible de résoudre 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 goal 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 des recherches du plus court chemin 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.

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 détail, 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, par exemple. Cependant, on peut dès lors se demander

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 \equiv list of points which lead to the goal while avoiding the obstacles

Ensure: footbot goes to goal while avoiding obstacles

```

for intermediate goal in intermediate goals list do
  while intermediate goal not reached do
    update footbot position and orientation
    read proximity sensors ▷ or whatever other sensor in use
    if no obstacles too close then
      do greedy avoidance
    else
      do close obstacle avoidance
    end if
  end while
end for

```

s'il ne faudrait pas aussi chercher de nouveau un plus court chemin après un évitement imprévu ou si le robot a dévié d'une distance significative de sa trajectoire prévue.

4.6.1 Recherche du plus court chemin**Algorithme A***

[wik13a]

En informatique, A* est un algorithme informatique qui consiste à mettre en place un processus de traçage d'un chemin traversable efficace entre des points. Les points sont considérés comme des noeuds.

A* utilise une «best-first» recherche et trouve le chemin possédant le moindre coût à partir d'un nœud initial donné à un nœud but.

Pour cela, il utilise une fonction de coût pour déterminer l'ordre dans lequel les visites de recherche des nœuds de l'arbre vont s'effectuer. Cette fonction de coût est la somme de deux autres fonctions. La fonction coût de la trajectoire passée qui est la distance connue à partir du nœud de départ au dernier parcouru lors de la recherche. Et la fonction coût du chemin futur qui est une estimation heuristique de la distance entre le dernier nœud parcouru et le nouveau nœud à atteindre .

Cet algorithme possède quand même des limites. Il est effectivement efficace dans le cas où l'on considère les robots comme omniscient, connaissant l'environnement et, donc, connaissant la position de la source et des obstacles se trouvant dans l'environnement. Dans le cas de non-

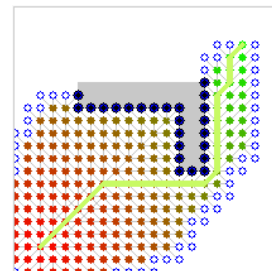


FIGURE 4.2 – Représentation d'une exécution de A*[wik13a]

omniscience, l'arène est inconnue et on ne possède aucunes données à propos de celles-ci. Et c'est ici que se trouve le plus grand défaut de l'algorithme A*.

A* détermine un chemin complet de nœuds pour arriver d'un nœud départ à un nœud but. Lorsqu'il ne possède pas de données complètes à propos de l'arène, il est bloqué lors de son exécution et ne peut donc pas déterminer le chemin que doit suivre le robot. Il faudra adapter l'algorithme A* afin de remédier à ce problème.

Algorithme de Dijkstra

[wik13b]

L'algorithme de Dijkstra est un algorithme servant à résoudre le problème du plus court chemin. Le principe de l'algorithme est le suivant :

Il s'agit de mettre en place progressivement un sous-graphe dans lequel sont classés les différents sommets. Un ordre croissant est établi entre les sommets et il est fixé en fonction de la distance minimale qui éloignent ces sommets à celui de départ. Cette distance correspond à la somme des nœuds parcourus.

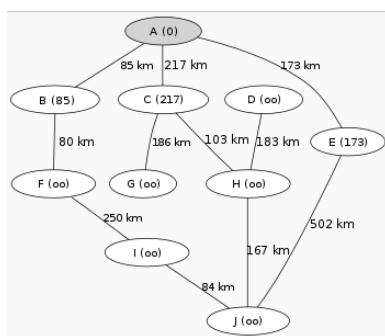
Au début, les distances de chaque sommet par rapport au sommet de départ sont considérées comme infinie et on attribue à celui-ci une distance de 0.

Ensuite, au cours de chaque itération, les distances des sommets reliés par un nœud au dernier du sous-graphe sont mis à jour. Cette mise à jour consiste à ajouter la valeur du nœud à la distance séparant le sommet de départ à ce dernier sommet. Après cette mise à jour, l'ensemble des sommets, ne faisant pas partis du sous-graphe, sont examinés et celui qui possède la distance minimale y est ajouté.

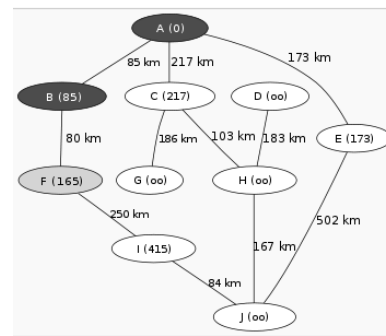
Enfin, on répète l'exécution jusqu'à l'épuisement des sommets ou jusqu'à la sélection du sommet d'arrivée. Voici 3 figures qui représentent un exemple du principe utilisé. Le but est de trouver le plus court chemin entre le point A et le point J. Comme dit précédemment, après chaque mise à jour, le sommet possédant la distance minimale est rajouté au sous graphe. La figure 4.3 représente bien cela.

En effet, le sommet E est rajouté au sous graphe et non le sommet I car la distance à parcourir entre A et E est plus petite que entre A et I.

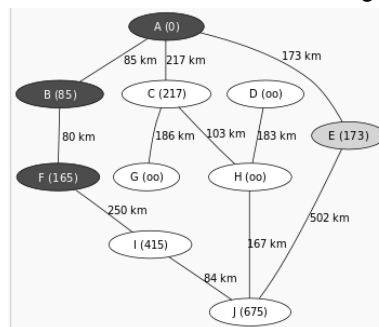
Il faut souligner que cet algorithme possède le même inconvénient qu'A*.



(a) Mise à jour initiale



(b) Mise à jour et construction du sous-graphe



(c) Mise à jour et graphe final

FIGURE 4.3 – Représentation d'une exécution de Dijkstra [wik13b]

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
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

```

end

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

Chapitre 5

Comportement

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

Listing 5.1 – 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
```

5.1 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 renvoyées *onSource*, *backHome*,...

L'implémentation détaillée de la fonction *doCommon* est donnée dans l'annexe 5.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.

5.2 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 5.4) 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 reçoit 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 communication entre robot étant relativement efficace, 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. Malheureusement, par manque de temps, nous avons dû nous limiter 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.

Ce modèle 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 portée extrêmement limitée des capteurs de couleur de sol.

L'implémentation du modèle de diffusion est détaillée en annexe 5.B

5.3 Exploitation

Une fois passé dans l'état *exploite*, le footbot fait simplement des allers-retours entre une des ressources connues et le nid en suivant le modèle de déplacement présenté 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.

Il est possible d'améliorer l'efficacité de l'exploitation en incitant les robots à exploiter les sources fournissant le meilleur rendement. Une possibilité est d'associer un score de qualité à chaque source connue et d'utiliser l'algorithme suivant :

Algorithm 5.1 Mise en place d'un score de qualité des sources

Require: List of available ressources with their quality score

loop

Pick ressource to mine while taking into account their quality score

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 aller, mais après un aller-retour. Cependant, ceci aurait été plus difficile à mettre en pratique dans notre cas, et nous avons donc choisi la solution présentée ci-dessus. Pour réévaluer le score d'une source, le pourcentage de batterie restant à ce moment est utilisé, ce qui indique directement le temps mis par le robot pour atteindre la source. Le nouveau score de qualité est donné par

$$newScore(battery, oldScore) = \begin{cases} oldScore + (1 - oldScore) \times \frac{battery}{100} & \text{si } battery > \beta \\ oldScore - oldScore \times \frac{100 - battery}{100} & \text{si } battery \leq \beta \end{cases}$$

Où $0 < \beta < 100$ indique le niveau minimum de batterie à l'arrivée pour qu'une source soit considérée comme rentable. Ceci permet une bonne convergence tout en gardant l'indice de qualité dans l'intervalle $[0, 1]$.

L'implémentation détaillée de l'évaluation et du choix des ressources est présentée dans l'annexe 5.C.

5.4 Communication

Vu la structure en boucle demandée par ARGoS, et grâce au pas d'exécution 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 5.2 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 nombre à virgule flottante compris entre -500 et 500, la conversion suivante est utilisée pour transmettre l'information :

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
type message	Signe x	Signe y	Centaines x	Centaines y	Reste x	Reste y
			$\lfloor \frac{ x }{100} \rfloor$	$\lfloor \frac{ y }{100} \rfloor$	$\text{round}(x \bmod 100)$	$\text{round}(y \bmod 100)$
	coordonnées					

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.

Lorsque qu'un robot reçoit un message, il le decode 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 5.2.

L'implémentation en Lua de la communication est détaillée dans l'annexe 5.D.

5.5 Gestion de l'autonomie

Une condition sur l'autonomie des robots a été imposée : l'autonomie est fixée dans le temps, ce qui signifie que leur batterie se décharge de manière constante à chaque pas d'ARGoS. Une valeur chiffrée n'a pas été imposée, mais les robots doivent juste être capables d'effectuer un aller-retour vers un point le plus éloigné de leur nid de départ à leur vitesse de régime. Une fois la batterie épuisée, le robot est rendu incapable de se déplacer mais pourra peut-être être dépanné par un autre membre de l'essaim dans le futur.

Puisque pour le moment le seul but d'un robot est d'exploiter une ressource, celui-ci peut avorter un aller retour dès qu'il estime qu'il n'est plus capable d'atteindre son but et de revenir ensuite à son nid.

Algorithm 5.3 Gestion de l'autonomie

Require: $0 \leq \text{battery} \leq 100$:= the battery left of the robot

Ensure: footbot tries to get back to nest when current goal judged not safely reachable

```

1 while goal not reached do
2   update footbot position and battery
3    $\text{cost} \leftarrow \text{evaluate cost}(\text{position}, \text{goal}, [\text{battery}])$   $\triangleright$  The cost of what is left to do
4   if  $\text{cost} > \text{battery}$  then
5      $\text{goal} \leftarrow \text{nest}$   $\triangleright$  Get back to the nest. If the footbot is already on its way back, this doesn't do anything. Appropriate to try to get to the nest when you know you don't have enough to get there ?
6   end if
7 end while
```

Où $\text{evaluate cost}(\text{position}, \text{goal}, [\text{battery}])$ est la fonction qui évalue l'énergie nécessaire à l'accomplissement du reste du trajet. On peut par exemple utiliser la distance à vol d'oiseau restante à parcourir accompagnée d'un facteur de sécurité, ou alors utiliser une approche heuristique utilisant le rapport entre la batterie utilisée jusqu'au step courant et le déplacement net parcouru vers le but.

En vue de la prochaine étape (la non omniscience des robots), le choix d'implémentation de la batterie autorise des sacrifices, c'est à dire, si un robot, étant à une position, estime devoir aller à une autre position et si sa batterie le lui permet, il ira à cette nouvelle position, sans toutefois vérifier qu'en y allant il pourra rentrer à la base recharger sa batterie. Ce choix

favorise l'exploration de l'arène en mettant en avant la réussite de l'ensemble du groupe et non la réussite personnelle, pouvant être interprété comme la recharge de la batterie. Cependant des modifications doivent encore être apportées car nous ne pouvons nous permettre de sacrifier l'ensemble des robots, faute de quoi l'objectif ne sera pas atteint.

Annexe 5.A Implémentation des opérations communes en Lua

5.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 5.2 – 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<10

  emerProx, emerDir=readProxSensor()
  if battery==0 then
    BASE_SPEED=0
    logerr("batt_empty")
  end
  if currentStep%5000==0 then
    log(travels)
  end
  return obstacleProximity, obstacleDirection,
    onSource, foundSource, backHome,
    gotSource, emerProx, emerDir
end
```

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

Listing 5.3 – 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]={posX,posY,score=.5}
      --store source. Initial quality index = 1/2
      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 5.4 – 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 5.5 – 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 5.B Implémentation du modèle de diffusion en Lua

Listing 5.6 – 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 5.7 – 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 5.8 – 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 5.C Implémentation du choix de ressource à exploiter en Lua

Listing 5.9 – fonction doMine

```
function doMine(obstacleProximity, obstacleDirection,
               onSource, backHome)
  if onSource then
    if goalX~=0 and goalY~=0 then
      --if we don't check for this then
      --we reevaluate at every step we're on the source
      evalSource(sourceId, battery)
    end
    goalX=0
    goalY=0
  elseif backHome then
    sourceId,goalX,goalY=chooseNewSource(ressources)
    travels=travels+1
  end
  if backForBattery then
    move(obstaclesTable, obstacleProximity, obstacleDirection,0,0)
  else
    obstaclesTable = updateObstaclesTable("long_range",obstaclesTable)
    move(obstaclesTable, obstacleProximity, obstacleDirection,goalX,goalY)
  end
end
end
```

Listing 5.10 – Choix de la ressource à exploiter

```
function chooseNewSource(rscList)
  local sourceChosen=false
  local pickSource
  while not sourceChosen do
    pickSource=robot.random.uniform_int(1,#rscList+1)
    sourceChosen=robot.random.uniform()<rscList[pickSource].score
  end
  local x=rscList[pickSource][1]
  local y=rscList[pickSource][2]
  return pickSource,x,y
  --return ressource index as well to make
  --ressource evaluation easier later
end
```

Cette implémentation permettrait en plus très facilement au robot de recommencer à explorer l'arène si les sources disponibles sont estimées peu rentables. En effet, il suffit d'ajouter un compteur à la boucle while et de repasser dans l'état *explore* si ce compteur est atteint sans qu'une source ait été choisie. Cependant, il faudrait alors aussi permettre au footbot de changer d'état dans l'autre sens à d'autres occasions que seulement s'il trouve ou reçoit une nouvelle source, comme il a été évoqué dans la section 5.2.

Listing 5.11 – Evaluation de la qualité de la source

```
function evalSource(sourceId, battery)
  if battery>GOOD_MINE_BATT then
    ressources[sourceId].score=ressources[sourceId].score
      +(1-ressources[sourceId].score)
      *battery/100
    --Ensures quality score goes up while staying <1
  else
    ressources[sourceId].score=ressources[sourceId].score
      -ressources[sourceId].score
      *(100-battery)/100
    --Ensures quality score goes down while staying >0
  end
end
```

Annexe 5.D Implémentation en Lua de la communication

5.D.1 Envoi de message

Listing 5.12 – 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 5.13 – 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
```

5.D.2 Réception de message

Listing 5.14 – 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=.5  
                ressources[#ressources+1]=source  
                gotSource=true  
            end  
        end  
    end  
    return gotSource  
end
```

Listing 5.15 – 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
```

Chapitre 6

Résultats obtenus

La première partie du projet consiste à installer ARGoS, guider un robot à un point donné et remplir l'objectif lorsque les robots sont omniscients comme le présente le cahier des charges [cah13]. Celle-ci a, en effet, été achevée. L'installation a été réalisée avec succès par tous les membres du groupe. Les robots parviennent aux points désignés tout en évitant les obstacles. De plus, ils peuvent, étant omniscients, faire des allers-retours entre le source et le nid. Les robots ont aussi la faculté de détecter une source grâce à ses senseurs et de tenir compte de l'épuisement de leur batterie. Ceci est illustré sur la figure 6.1. Dans le cas de la figure 6.1b, on peut observer les robots qui se dirigent vers la source. Sur l'autre figure, les robots sont de retour au nid.

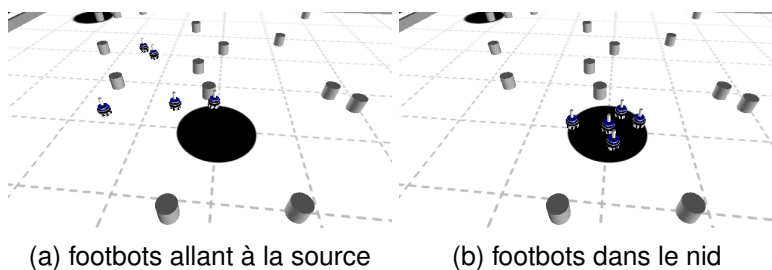


FIGURE 6.1 – Exécution d'ARGoS avec le comportement produit

La création d'une collaboration au sein de l'essaim, le choix et l'intégration d'une communication entre robots, font partie de la seconde partie du projet. A ce niveau, grâce à une études des déplacements de base, la non-omniscience des robots peut déjà être gérée et des recherches sur la communication entre robots ont déjà été faites comme présenté au chapitre 2.

Chapitre 7

Fonctionnement du groupe

7.1 Général

Le groupe est composé de 5 membres. Chaque membre a 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 mettre en avant ses points forts et d'améliorer ses points faibles durant le projet. Des tensions sont tout de même survenues. En effet, le manque de communication a souvent mené de groupe à mal se coordonner.

7.2 Organisation

Le groupe se fixe une réunion par semaine. Les tâches à effectuer pour la prochaine réunion sont distribuées à la fin de celle-ci et notées dans les PV qui sont généralement envoyés dans les 48h. Des rôles d'animateur et secrétaire sont réattribués toutes les 4 semaines. Pour diriger au mieux le projet, un diagramme de Gantt (cf. : annexe) a été conçu pour avoir une vision globale de l'avancé de celui-ci et également se fixer des dates butoirs.

7.3 Communication

Dans un premier temps, un groupe sur facebook a été créé. C'est un moyen simple et rapide de se communiquer les informations mais une communication par mail reste préférable. Pour le partage, des codes, des fichiers et des sources des recherches, les dispositifs github, dropbox et zotero ont été mis en place. L'emploi de ces 3 dispositifs n'a pas encore

été exploité au mieux mais son apprentissage est bénéfique non seulement dans le cadre de ce projet mais également pour plus tard.

Chapitre 8

Conclusion

En conclusion, une compréhension de l'intelligence artificielle, du fonctionnement des robots et l'apprentissage de Lua ont permis d'atteindre les objectifs du 1er quadrimestre, ceux-ci étant de guider les robots du nid à la source et de les programmer pour qu'ils puissent y faire des allers-retours. Les prochains objectifs à atteindre sont la mise en place d'un moyen de communication et d'exploration optimale l'environnement. Quelques options vont probablement être ajoutées dont l'introduction d'un système de priorité, dans le cas où 2 robots cherchent à s'éviter, et une batterie se déchargeant de manière différente suivant l'état du robot (au repos ou en mouvement). De plus, une étude du comportement en essaim généré par notre code sera menée afin d'évaluer l'«intelligence de l'essaim».

Algorithmes

4.1	Convergence simple sans évitement d'obstacle	11
4.2	Convergence avec évitement <i>greedy</i> d'obstacles lointains . .	12
4.3	Convergence avec évitement d'obstacles proches	13
4.4	Convergence avec chemin précalculé	15
5.1	Mise en place d'un score de qualité des sources	25
5.2	Partage total de la connaissance des ressources	26
5.3	Gestion de l'autonomie	27

Listings

3.1	Structure de base d'un comportement en Lua	9
4.1	Initialisation	18
4.2	Structure générale	18
4.3	Rafraîchir la table des mesures à chaque pas	18
4.4	Fonction move	19
4.5	Trouver la direction du goal vu du footbot	19
4.6	Trouver la direction optimale et la suivre	19
4.7	Fonction getToGoal	19
4.8	Capteur supplémentaire	20
4.9	lecture du <i>proximity sensor</i>	20
4.10	Fonction emergencyAvoidance	20
5.1	Fonction step	23
5.2	fonction doCommon	29
5.3	fonction checkGoalReached	30
5.4	Détection de la couleur du sol	30
5.5	Vérifier l'originalité de la source	30
5.6	fonction doExplore	31
5.7	Déplacement <i>gaslike</i>	31
5.8	Calcul de l'angle "réfléchi"	32
5.9	fonction doMine	33
5.10	Choix de la ressource à exploiter	33
5.11	Evaluation de la qualité de la source	34
5.12	Envoi d'un message différent à chaque pas	35
5.13	Encodage du message	35
5.14	Ecoute du capteur à chaque pas	36
5.15	Décodage du message	36

Table des figures

2.1	Cycle d'interaction entre l'environnement et les agents	5
3.1	Les différents senseurs et actuateurs d'un footbot [Pin]	7
4.1	Signification physique de v_r , v_l , v_g et ω_g [Pin]	10
4.2	Représentation d'une exécution de A* [wik13a]	15
4.3	Représentation d'une exécution de Dijkstra [wik13b]	17
6.1	Exécution d'ARGoS avec le comportement produit	37

Bibliographie

- [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 rationnelle 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. AR-GoS : 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>.
- [wik13a] A* search algorithm from wikipedia, the free encyclopedia, October 2013. Page Version ID : 579529277. Available from : http://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=579529277.
- [wik13b] Algorithme de dijkstra from wikipedia, the free encyclopedia, December 2013. Page Version ID : 98428897. Available from : http://fr.wikipedia.org/w/index.php?title=Algorithme_de_Dijkstra&oldid=98428897.