Le calibrage de modèles à base d'agents pour la simulation de systèmes complexes.

Benoît Calvez

▶ To cite this version:

Benoît Calvez. Le calibrage de modèles à base d'agents pour la simulation de systèmes complexes.. Modélisation et simulation. Université d'Evry-Val d'Essonne, 2007. Français. NNT : . tel-00422236

HAL Id: tel-00422236
https://theses.hal.science/tel-00422236

Submitted on 6 Oct 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ D'ÉVRY-VAL D'ESSONNE

THÈSE

présentée en vu d'obtenir le grade de Docteur,
spécialité « Informatique »

par

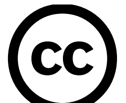
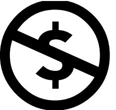
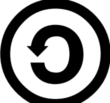
Benoît CALVEZ

LE CALIBRAGE DE MODÈLES À BASE D'AGENTS POUR LA SIMULATION DE SYSTÈMES COMPLEXES

Thèse soutenue le 18 décembre 2007 devant le jury composé de :

M.	ÉRIC ANGEL	Université d'Évry Val d'Essonne	Examinateur
M.	FRANÇOIS BOURDON	Université de Caen Basse-Normandie	Examinateur
M.	ALEXIS DROGOUL	IRD, UR 079	Rapporteur
M.	ÉRIC RAMAT	Université du Littoral Côte d'Opale	Rapporteur
M.	GUILLAUME HUTZLER	Université d'Évry Val d'Essonne	Encadrant
Mme	HANNA KLAUDEL	Université d'Évry Val d'Essonne	Directrice de thèse

Thèse préparée dans l'équipe LIS du laboratoire IBISC
FRE 2873 CNRS / Université d'Évry Val d'Essonne

Titre

Le calibrage de modèles à base d'agents pour la simulation de systèmes complexes

Mot-clefs

modèle à base d'agents, simulation multi-agent, calibrage, optimisation

Résumé

L'approche à base d'agents s'intéresse à la modélisation et la simulation de systèmes complexes.

Un des aspects importants dans le processus de conception est lié à la mise au point des paramètres du modèle. En effet, ces modèles sont généralement caractérisés par de nombreux paramètres qui déterminent la dynamique globale du système simulé. L'espace des paramètres peut être alors gigantesque. De plus, le comportement de ces systèmes complexes est souvent chaotique.

L'approche que nous suggérons est de considérer le problème de calibrage des modèles à base d'agents comme un problème d'optimisation. La validation peut alors être reformulée comme l'identification d'un jeu de paramètres qui optimise une fonction, par exemple une mesure de distance entre le modèle artificiel que nous simulons et le système réel. Nous avons proposé trois approches complémentaires dans le cadre de la thèse.

La première approche se fonde sur l'application directe d'un algorithme d'optimisation où le modèle est vu comme une boîte noire dont les entrées sont les valeurs de paramètres et la sortie la valeur d'une fonction objectif évaluée après la simulation du modèle. La deuxième approche consiste ensuite à explorer différentiellement l'espace des paramètres en le découpant de manière adaptative en sous-espaces d'autant plus finement découpés que les paramétrages correspondants sont a priori intéressants. Enfin, la troisième approche consiste à effectuer une seule simulation où les agents sont modifiés en ligne.

Title

Calibration of Agent-based Model For Complex Systems.

Keyword

agent-based model, agent-based simulation, calibration, optimization

Abstract

The agent based approach is interested in the modelling and the simulation of complex systems.

The tuning of the model constitutes a crucial step of the design process. Indeed, agent-based models are generally characterized by lots of parameters, which together determine the global dynamics of the system. The search space can thus be gigantic. Moreover the behavior of these complex systems is often chaotic.

The approach that we suggest is to consider the problem of the tuning of the agent-based model as an optimization problem. The validation can thus be reformulated as the identification of a parameter set that optimizes some function, for instance a measure of the distance between the artificial world that we simulate and the real system. We proposed three approaches.

The first approach is based on the direct use of an optimization algorithm where the model is seen as a black box whose inputs are the values of the parameters, and whose output is the value of fitness function computed after the simulation of the model. The second approach consists in exploring the parameter space differentially, dividing it adaptively into sub-spaces. The third approach consists in a single simulation where the agents are modified online.

Table des matières

Introduction **xv**

I État de l'art 1

I.1 Modélisation et simulation multi-agent **3**
I.1.1 Définition . . . 3
I.1.2 Simulation à base d'agents : exemple et intérêts . . . 4
I.1.3 Calibrage/Validation . . . 9
I.1.4 Conclusion et objectif de la thèse . . . 10

I.2 Espace de paramètres **13**
I.2.1 Calibrage pour les modèles à base d'agents . . . 14
I.2.2 Discussion . . . 16
I.2.3 Conclusion . . . 18

I.3 Optimisation **19**
I.3.1 Algorithme génétique . . . 19
I.3.2 Programmation génétique . . . 21
I.3.3 Recuit simulé . . . 21
I.3.4 Colonies de fourmis . . . 23
I.3.5 Apprentissage par renforcement . . . 26
I.3.6 Discussion . . . 27

II Exploration de l'espace des paramètres : les différentes méthodes 29

II.1 Algorithmes génétiques ou les méthodes classiques d'optimisation **31**
II.1.1 Méthode . . . 31
II.1.2 Résultats . . . 35
II.1.3 Choix de la *fitness* . . . 36
II.1.4 Par rapport aux autres méthodes d'optimisation . . . 39
II.1.5 Discussion . . . 41

II.2 Découpage de paramètres **43**
II.2.1 Méthode en détail . . . 43
II.2.2 Premiers résultats . . . 50
II.2.3 Variantes . . . 51
II.2.4 Résultats . . . 58
II.2.5 Discussion . . . 64

II.3 Évolution d'agents **67**
II.3.1 Description détaillée de la méthode . . . 67
II.3.2 Résultats . . . 73
II.3.3 Discussion . . . 76

II.4 Discussion **79**
II.4.1 Comparaison des méthodes . . . 79
II.4.2 Mélange des méthodes . . . 81

II.5 Applications 89
II.5.1 Hyperstructure . . . 89
II.5.2 Hétérogénéité dans les cultures *in vitro* de populations de cellules clonales . . . 95

Conclusion 101

Bibliographie 105

A Plates-formes multi-agents 119

B Exemples de simulations multi-agents 127
B.1 Modèle « *Cooperation* » de NetLogo . . . 127
B.2 Modèle « *Traffic 2 Lanes* » de NetLogo . . . 128
B.3 Modèle « *Flocking* » de NetLogo . . . 129
B.4 Modèle « *Divide The Cake* » de NetLogo . . . 130
B.5 Modèle « *Ants* » de NetLogo . . . 130
B.6 Modèle « *Wolf Sheep Predation* » de NetLogo . . . 132

C Implantation des différentes méthodes 133

D Exemple détaillé de l'utilisation des méthodes 137

E Fonctions de référence 143
E.1 *Rastrigin* . . . 143
E.2 Fonction « plate » . . . 144
E.3 Fonction « somme de cosinus » . . . 147
E.4 Fonction « somme de sinus » . . . 148
E.5 Fonction de *Ackley* . . . 149

Table des figures

I.1.1 Principe de la simulation 4
I.1.2 Évolution des populations de proies et de prédateurs par le modèle de Lotka-Volterra . . 6
I.1.3 Jeu de la vie 6
I.1.4 Évolution du comportement des fourmis en fonction du taux d'évaporation des phéromones 10

I.2.1 Principe de construction d'un méta-modèle 14
I.2.2 Outil *BehaviorSpace* 15
I.2.3 Outil *Parameters Variation* de `AnyLogic` 16
I.2.4 Vision entrées/sortie d'une simulation à base d'agents 17
I.2.5 Évolution d'une simulation multi-agent 18
I.2.6 Outil *Optimization* de `AnyLogic` 18

I.3.1 Schéma d'un algorithme génétique 19
I.3.2 Exemple de « *genetic programming* » 22
I.3.3 Recombinaison 22
I.3.4 Mutation 23
I.3.5 Schéma d'un recuit simulé 24
I.3.6 Schéma d'un recuit simulé parallèle 24
I.3.7 Expérience du double pont 25

II.1.1 Vision en boîte noire de la simulation multi-agent 31
II.1.2 Schéma de la méthode algorithme génétique 32
II.1.3 Schéma de la méthode algorithme génétique avec les simulations distribuées 33
II.1.4 Principe de la méthode tenant compte de la stochasticité 35
II.1.5 Résultat de la méthode algorithme génétique sur le modèle de fourragement par colonie de fourmis : l'évolution de la *fitness* des meilleurs modèles paramétrés en fonction du nombre de simulations 37
II.1.6 Évolution de la valeur du paramètre `speed` des meilleurs modèles paramétrés au cours de l'algorithme génétique 37
II.1.7 Évolution de la valeur d'un paramètre artificiel des meilleurs modèles paramétrés au cours de l'algorithme génétique 38
II.1.8 Schéma de la méthode avec recuit simulé 40
II.1.9 Résultat des méthodes algorithme génétique et recuit simulé sur le modèle de fourragement par colonie de fourmis 40
II.1.10 Évolution de la *fitness* à différentes exécutions de la méthode à base d'algorithme génétique et à base de recuit simulé 41

II.2.1 Méthode « Découpage de paramètres » 44
II.2.2 Initialisation des paramètres d'un agent 45
II.2.3 Sélection des intervalles 46
II.2.4 Distribution des valeurs de paramètres pour les 100 premières simulations 47
II.2.5 Exemple d'une convergence de *fitness* globale 49
II.2.6 Exemple d'une non-convergence de *fitness* globale 50
II.2.7 Exemple d'une convergence pour un découpage d'un paramètre 50
II.2.8 Exemple de non-convergence pour un découpage d'un paramètre 50
II.2.9 Évolution de la *fitness* en fonction du nombre de simulations avec la méthode découpage de paramètres 51
II.2.10 Évolution du découpage du paramètre `speed` avec la méthode découpage de paramètres 52

II.2.11 Évolution du découpage du paramètre `foo` avec la méthode découpage de paramètres . 52
II.2.12 Évolution du découpage d'un paramètre `speed` avec la méthode découpage de paramètre avec ajout d'aléatoire 54
II.2.13 Principe de l'initialisation d'un modèle pour le calcul de la valeur d'heuristique d'un intervalle 55
II.2.14 Schéma résumant la variante de la méthode découpage de paramètre s'inspirant des algorithmes de colonies de fourmis 56
II.2.15 Évolution de la *fitness* en fonction du nombre de simulations avec la méthode découpage de paramètres avec inspiration des colonies de fourmis 57
II.2.16 Évolution du découpage du paramètre `speed` avec la méthode découpage de paramètre avec inspiration des colonies de fourmis 57
II.2.17 Évolution du découpage d'un paramètre artificiel avec la méthode découpage de paramètre avec inspiration des colonies de fourmis 57
II.2.18 Évolution de α et de β 58
II.2.19 Évolution de la *fitness* en fonction du nombre de simulations avec la méthode découpage de paramètres en limitant à 25 intervalles par paramètres 59
II.2.20 Évolution du découpage du paramètre `speed` avec la méthode découpage de paramètres en limitant à 25 intervalles par paramètre 59
II.2.21 Évolution du découpage du paramètre artificiel avec la méthode découpage de paramètres en limitant à 25 intervalles par paramètre 60
II.2.22 Évolution de la *fitness* en fonction du nombre de simulations avec la méthode découpage de paramètres en limitant à 100 intervalles par paramètres 60
II.2.23 Évolution du découpage du paramètre `speed` avec la méthode découpage de paramètres en limitant à 100 intervalles par paramètre 61
II.2.24 Évolution du découpage du paramètre artificiel avec la méthode découpage de paramètres en limitant à 100 intervalles par paramètre 61
II.2.25 Principe de la méthode de découpage en 2-étapes 62
II.2.26 Évolution de la *fitness* en fonction du nombre de simulations avec la méthode découpage de paramètre en n-étapes dans le cas $n = 3$ 63
II.2.27 Évolution du découpage du paramètre `speed` avec la méthode découpage de paramètre en n-étapes dans le cas $n = 3$ 63
II.2.28 Évolution du découpage du paramètre `foo` avec la méthode découpage de paramètre en n-étapes dans le cas $n = 3$ 64
II.2.29 Comparaison des différentes méthodes de découpages de paramètres 65

II.3.1 Schéma du principe de la méthode d'évolution d'agents 68
II.3.2 Schéma du principe de la méthode d'évolution d'agents par générations 69
II.3.3 Évolution de la *fitness* en fonction du nombre de générations et du taux de remplacement 70
II.3.4 Schéma du principe de la méthode d'évolution d'agents en continu 71
II.3.5 Évolution de la *fitness* en fonction du nombre de pas de simulation pour le modèle « *Traffic 2 Lanes* » par la méthode évolution d'agents 74
II.3.6 Le modèle « *Traffic 2 Lanes* » optimisé par la méthode évolution d'agents 74
II.3.7 Évolution des populations d'agents en fonction du nombre de pas de simulation pour le modèle « *Divide the cake* » par la méthode évolution d'agents 75
II.3.8 Le modèle « *Divide the cake* » optimisé par la méthode évolution d'agents 75
II.3.9 Évolution de la *fitness* globale en fonction du nombre de simulations pour le modèle « *Ants* » par la méthode évolution d'agents 76
II.3.10 Évolution des valeurs du paramètre `speed` en fonction du nombre de simulations par la méthode évolution d'agents 77
II.3.11 Évolution des valeurs du paramètre artificiel en fonction du nombre de simulations par la méthode évolution d'agents 77

II.4.1 Comparaison des différentes méthodes sur le modèle « *Ants* » 80
II.4.2 Principe de la méthode découpage de paramètres puis algorithmes génétiques 83
II.4.3 Évolution de la *fitness* globale en fonction du nombre de simulations pour le modèle « *Ants* » par la méthode découpage de paramètres puis algorithmes génétiques 84

II.4.4 Évolution de la *fitness* globale en fonction du nombre de simulations et en fonction du nombre de simulations alloués à la partie découpage de paramètres par la méthode découpage de paramètres puis algorithmes génétiques 84
II.4.5 Principe de la méthode algorithmes génétiques et évolution d'agents 85
II.4.6 Évolution de la *fitness* globale en fonction du nombre de simulations pour le modèle « *Ants* » par la méthode algorithmes génétiques et évolution d'agents 86
II.4.7 Principe de la méthode découpage de paramètres puis algorithmes génétiques et évolution d'agents 87
II.4.8 Évolution de la *fitness* globale en fonction du nombre de simulations pour le modèle « *Ants* » par la méthode découpage de paramètres puis algorithmes génétiques et évolution d'agents 88

II.5.1 Complexe enzymatique 90
II.5.2 Schéma représentant la Glycolyse 92
II.5.3 Schéma de la glucose phosphotransférase 92
II.5.4 Schéma récapitulatif des réactions 93
II.5.5 Comparaison de résultats sur les hyperstructures. 94
II.5.6 Comparaison de résultats sur les hyperstructures dans le cadre où les métabolites intermédiaires sont labiles. 94
II.5.7 Image du modèle représentant une population de cellule *C2C12* 95
II.5.8 Principe des différentes hypothèses 96
II.5.9 Histogramme de la distribution du nombre de voisins de la population *SP* 96
II.5.10 Évolution de la *fitness* du modèle sur l'hétérogénéité dans les cultures *in vitro* de populations de cellules clonales par la méthode à base d'algorithmes génétiques 97
II.5.11 Évolution du découpage du paramètre `radius_av` du modèle sur l'hétérogénéité dans les cultures *in vitro* de populations de cellules clonales par la méthode de découpage de paramètres 98

A.1 Le modèle « *Boids* » sous `Mathematica` 119
A.2 Le modèle « *Flockers* » sous `Mason` 120
A.3 Le modèle « *HeatBugs* » sous `Repast` 122
A.4 Le modèle « *Swarm* » sous `Breve` 122
A.5 Le modèle « *Boids* » sous `StarLogo` 122
A.6 Le modèle « *Flocking.xml* » sous `SeSAm` 124
A.7 Le modèle « *Flocking* » sous `Netlogo` 125

B.1 Image du modèle « *Cooperation* » 127
B.2 Image du modèle « *Traffic 2 Lanes* » 128
B.3 Image du modèle « *Flocking* » 129
B.4 Image du modèle « *Divide The Cake* » 130
B.5 Image du modèle « *Ants* » 131
B.6 Image du modèle « *Wolf Sheep Predation* » 132

C.1 Exemple de scénario de fonctionnement 135

D.1 Image du modèle « *Shepherds* » 137
D.2 Évolution de la *fitness* en fonction du nombre de simulations sur le modèle « *Shepherds* » 140
D.3 Évolution du découpage du paramètre `speed-shepherds` avec la méthode découpage de paramètres sur le modèle « *Shepherds* » 140
D.4 Évolution du découpage du paramètre `vision-shepherds` avec la méthode découpage de paramètres sur le modèle « *Shepherds* » 141
D.5 Évolution du découpage du paramètre `speed-sheep` avec la méthode découpage de paramètres sur le modèle « *Shepherds* » 142

E.1 Visualisation de l'espace des paramètres d'un agent pour une fonction de *Rastrigin* en dimension 2. 144
E.2 Visualisation de l'évolution de la *fitness* moyenne à travers les trois méthodes pour la fonction de test *Rastrigin*. 144

E.3 Visualisation de l'espace des paramètres d'un agent pour la fonction « plate » pour deux paramètres. 145
E.4 Visualisation de l'évolution de la *fitness* à travers les trois méthodes. 146
E.5 Visualisation de la moyenne du paramètre 2 des 5 meilleurs modèles de chaque génération. 146
E.6 Visualisation du l'intervalle final du paramètre 2. 147
E.7 Visualisation de l'espace des paramètres d'un agent pour la fonction « somme de cosinus » pour deux paramètres 148
E.8 Visualisation de l'espace des paramètres d'un agent pour la fonction « somme de sinus » 149
E.9 Visualisation de l'espace des paramètres d'un agent pour la fonction d'*Ackley* 149

Liste des tableaux

II.1.1 Paramétrage solution obtenu avec la méthode algorithme génétique 36
II.1.2 Différents paramétrages solutions pour différentes *fitness* 39
II.1.3 Paramétrage solution obtenu avec la méthode recuit simulé 39

II.2.1 Paramétrage solution obtenu avec la méthode découpage de paramètres 51

II.3.1 Paramétrage solution obtenu avec la méthode évolution d'agents 77

II.4.1 Tableau récapitulatif des différentes méthodes 81

Introduction

Le domaine des systèmes complexes est un domaine actuellement en plein essor, ainsi qu'en témoignent le développement de multiples instituts de recherche, et la création régulière de nouvelles revues et conférences sur le sujet. Issues d'échanges transdisciplinaires entre de nombreux domaines scientifiques tels que les sciences humaines, les sciences du vivants, les sciences de la matière ou encore les mathématiques et l'informatique, les études sur les systèmes complexes visent à améliorer la compréhension du fonctionnement et de l'organisation dynamique de ces systèmes. Ceux-ci présentent notamment comme caractéristiques d'être constitués d'un grand nombre d'entités en interaction locale et simultanée, avec un haut degré d'imbrication, des boucles de rétroaction, et potentiellement différents niveaux d'organisation qui s'influencent mutuellement. Dans ce cadre, certaines études visent à comprendre le fonctionnement particulier de certains systèmes spécifiques, tandis que d'autres visent à proposer des méthodes et outils d'analyse applicables à tous types de systèmes, ou en tout cas à la majorité d'entre eux. Tout au long de ce manuscrit, nous nous plaçons dans le cadre de cette deuxième approche, en faisant abstraction du type de système complexe modélisé, et en ne retenant que leurs principales caractéristiques.

Ces caractéristiques ont notamment pour conséquence qu'il n'est pas possible d'appliquer, dans l'étude de ces systèmes, une approche purement réductionniste dans laquelle les différents composants sont étudiés de manière détaillée, en les isolant des autres composants. Une grande partie des propriétés que l'on peut caractériser au niveau global lorsque l'on observe le système sont en effet la résultante des interactions locales entre les composants et l'on ne peut donc étudier les entités du système indépendamment des interactions qu'elles ont entre elles et avec leur environnement. Par ailleurs, le fait que les interactions sont généralement locales (les entités interagissent avec un nombre limité d'autres entités présentes dans un voisinage) et différenciées (tout le monde n'interagit pas avec tout le monde) limite l'intérêt des approches de type « systèmes dynamiques ». Ces approches, modélisant les systèmes par un ensemble de variables agrégées et d'équations différentielles traduisant les interactions entre les variables, sont impuissantes à rendre compte des phénomènes d'émergence et d'auto-organisation caractérisant bon nombre de systèmes complexes.

L'approche de modélisation et de simulation à base d'agents offre une alternative intéressante aux deux approches précédentes : de l'approche réductionniste, elle reprend le principe de décomposition du système en entités autonomes ; de l'approche systèmes dynamiques, elle reprend la description des interactions entre ces entités. Celles-ci sont modélisées comme autant d'agents dont on spécifie les comportements et les interactions. Ces agents évoluent dans un environnement également modélisé pour permettre la prise en compte des interactions localisées entre les agents. Cette approche permet ainsi la prise en compte de l'espace et de l'hétérogénéité du système, mais apporte avec elle son lot de difficultés. Parmi ces difficultés se pose en particulier le problème du choix des valeurs paramétrant le comportement des agents. Ce choix non seulement n'est pas simple car ces paramètres, pour une grande part, ne correspondent pas à des valeurs mesurables dans le système réel (on pourra seulement les estimer), mais il est surtout crucial car des valeurs de ces paramètres dépend bien souvent la dynamique globale du système. Comme le rappelle l'encyclopédie en ligne Wikipedia à propos des systèmes complexes [Wikipedia, 2007], « contrairement au cas d'un système chaotique, ce n'est pas tant la précision de ses paramètres qui est primordiale, mais le nombre de paramètres, et le fait que chacun d'entre eux peut avoir une influence essentielle sur le comportement du système. Pour prévoir ce comportement, il est nécessaire de tous les prendre en compte, ce qui revient à effectuer une simulation du système étudié. »

Une des étapes fondamentales dans la conception d'une modélisation à base d'agents résidera donc dans le calibrage et la validation du modèle. La validation devra permettre de déterminer si le modèle à base d'agents ainsi conçu représente de manière satisfaisante le système modélisé. Une

première validation d'un tel modèle est obtenue en comparant la dynamique résultante du modèle simulé avec celle du système réel. Pour valider ce modèle, il est nécessaire dans un premier temps de le calibrer, ce qui revient à choisir des valeurs pour chacun des paramètres qui le constituent. Dans le cadre de l'approche à base d'agents, un modèle est généralement caractérisé par de nombreux paramètres qui déterminent la dynamique globale du système simulé, et l'espace des paramètres peut alors être gigantesque. Pour certains de ces paramètres, le modélisateur dispose de valeurs très précises, soit à partir de connaissances du domaine, soit à partir d'observations du système. Pour d'autres paramètres, il dispose juste d'un ordre de grandeur plus ou moins fin des valeurs, à cause par exemple de connaissances du domaine incomplètes. Pour d'autres enfin, il peut n'avoir simplement aucune idée des valeurs : par exemple, des paramètres qui sont hors contexte du système comme la discrétisation de l'espace (le comportement d'un modèle peut être très différent suivant que l'espace est discrétisé finement ou grossièrement).

D'une part, une simulation d'un modèle à base d'agents peut être sensible à certaines valeurs de certains paramètres : une petite modification peut mener à des comportements de la simulation totalement différents. Un mauvais choix de paramètres peut conduire à l'apparition en simulation de comportements non désirés par le modélisateur, au sens où ils ne correspondent pas aux comportements connus dans le système réel. Ce mauvais paramétrage peut ainsi amener à rejeter un modèle alors qu'il aurait été possible de trouver un jeu de paramètres menant aux comportements attendus. D'autre part, certains phénomènes « émergents » ne se produisent que dans des conditions très spécifiques, nécessitant alors de paramétrer le modèle plus finement. La calibrage du modèle apparaît donc comme une étape primordiale dans la phase de conception. La difficulté est que le développement et le réglage des paramètres peut devenir long et fastidieux en l'absence de stratégie précise, automatique et systématique d'exploration de l'espace de paramètres. Proposer des méthodes, des techniques pour calibrer de façon plus ou moins automatique un modèle constitue donc une démarche importante, et c'est ce que nous proposons à travers ce manuscrit.

Le but de cette thèse est ainsi de proposer des méthodes pour calibrer des modèles à base d'agents de façon automatique. Nous nous plaçons dans le cadre où nous avons un modèle déjà conçu, que nous cherchons à calibrer. Dans ce contexte, les objectifs principaux de notre travail sont les suivants :

- proposer des méthodes qui minimisent le nombre de simulations à exécuter, chacune d'entre elles étant coûteuse en temps de calcul ;
- proposer des méthodes simples et aussi génériques que possible, applicables à tous types de modèles.

Le manuscrit se compose de deux parties principales : la première partie, composée de trois chapitres, constitue un état de l'art des domaines de modélisation et simulation à base d'agents d'une part, d'optimisation d'autre part ; nous présentons dans la deuxième partie, composée de cinq chapitres, les différentes méthodes d'exploration de l'espace des paramètres que nous avons développées, en discutant leurs points forts respectifs, et en montrant leur application sur plusieurs exemples réels.

Nous rappelons d'abord brièvement ce qu'est un agent, puis ce qu'est une simulation d'un modèle à base d'agents, et les principaux avantages de cette approche. À partir de ces définitions, nous présentons ensuite le problème spécifique d'exploration de l'espace des paramètres et les différentes méthodes pour l'explorer. L'approche qui nous apparaît comme la plus satisfaisante consiste à explorer l'espace des paramètres de manière aussi systématique que possible, tout en utilisant des heuristiques pour accentuer la recherche sur les zones a priori les plus intéressantes. Nous nous appuyons sur les conclusions de ce chapitre pour annoncer notre approche qui consiste à voir le calibrage comme un problème d'optimisation associé à une fonction objectif ou *fitness*. Nous passons donc également en revue différentes techniques d'optimisation.

Dans la deuxième partie du manuscrit, nous exposons les trois familles de méthodes que nous avons développées.

Nous nous intéressons tout d'abord à l'application d'un algorithme d'optimisation à notre problème de calibrage. Les modèles sont vus comme des boîtes noires ayant pour entrées les valeurs des paramètres et en sortie la valeur de *fitness* après simulation du modèle. Nous montrons comment appliquer une telle méthode en tenant compte des aspects stochastiques de la simulation, et en proposant de réaliser les simulations de façon distribuée. A travers différents résultats, se basant tant sur l'utilisation d'algorithmes génétiques que sur le recuit simulé, et en appliquant l'approche aussi bien à des modèles à base d'agents qu'à des fonctions mathématiques de référence, nous montrons qu'elle permet effectivement le calibrage de modèles avec une convergence nette vers des solutions proches des solutions optimales connues. Cette méthode présente l'intérêt d'être générique (pouvant s'appliquer à tous types de modèles) et d'être facilement parallélisable ; elle nous sert également de méthode de référence pour l'évaluation des autres méthodes développées.

Nous proposons ensuite une méthode pour explorer l'espace des paramètres de façon adaptative. Pour cela, les plages de valeurs des paramètres sont découpées en intervalles, et au fur et à mesure des simulations, la méthode divise les intervalles les plus intéressants en plus petits intervalles, et fusionne au contraire ceux qui sont moins intéressants. Nous montrons comment implanter une telle méthode, et montrons les principales variantes de cette approche. Sur un exemple, nous exposons des résultats et discutons des avantages de cette méthode par rapport à d'autres. Ces avantages sont notamment, en plus d'être générique et facilement parallélisable comme l'approche précédente, de converger plus rapidement en début d'optimisation, et de fournir en fin d'optimisation, non seulement un paramétrage solution mais également un « paysage » de solutions.

Nous proposons enfin une méthode qui consiste à ouvrir le modèle pour paramétrer individuellement chacun des agents, en modifiant leur paramétrage dynamiquement. Cela permet de n'avoir qu'une seule simulation au cours de laquelle le paramétrage des agents évolue jusqu'à obtenir une dynamique globale satisfaisante du système. Nous illustrons l'application de la méthode sur différents exemples et présentons les résultats correspondants. Nous concluons enfin sur les principaux intérêts de cette approche. Malgré une difficulté de conception accrue par rapport aux méthodes précédentes, qui la rend également moins facilement généralisable, cette méthode se révèle en effet très rapide puisqu'une seule simulation suffit à obtenir le calibrage du modèle.

Nous discutons ensuite des avantages et inconvénients des différentes approches, et nous proposons des méthodes les combinant les unes avec les autres de manière à essayer d'en combiner les avantages. Nous montrons finalement quelques applications à des cas réels.

Le manuscrit comporte également cinq annexes, présentant différentes plates-formes de simulation multi-agent, les modèles à base d'agents et les fonctions mathématiques de référence utilisés dans les évaluations, des détails d'implantation, ainsi qu'un exemple détaillé d'application des différentes méthodes.

Première partie

État de l'art

I.1 Modélisation et simulation multi-agent

Nous allons nous intéresser dans ce chapitre à la simulation multi-agent. Tout d'abord, nous définirons dans la première section la notion de simulation multi-agent, puis nous exposerons dans la deuxième section les principaux intérêts et avantages de la simulation à base d'agents, et enfin, nous discuterons de la question de la validation et du calibrage des modèles à base d'agents.

I.1.1 Définition

I.1.1.1 Qu'est-ce un agent ?

La notion d'agent provient de l'intelligence artificielle distribuée, mais elle varie en fonction du domaine ou des auteurs [Bird, 1993, Franklin et Graesser, 1997, Stone et Veloso, 2000, Hare et Deadman, 2004]. Par conséquent, avoir une définition précise d'agent est difficile.

Une des plus utilisées est celle de Wooldridge et Jennings [Wooldridge et Jennings, 1995a, Wooldridge et Jennings, 1995b]. Le terme agent [1] est utilisé pour dénoter un système informatique matériel ou plus usuellement logiciel qui a les propriétés suivantes :

- autonomie : les agents opèrent sans intervention directe des humains ou des autres agents, et ont une certaine forme de contrôle sur leurs actions et leur état interne.
- capacité sociale : les agents interagissent avec les autres agents (éventuellement avec des humains) via un langage de communication entre agents.
- réactivité : les agents perçoivent leur environnement (lequel peut être le monde physique, un utilisateur avec une interface graphique utilisateur, une collection d'autres agents, Internet, ou peut-être tous ces types combinés), et répondent en un temps limité aux changements qui s'y produisent.
- pro-activité : les agents n'agissent pas simplement en réponse de leur environnement, ils sont capables d'exhiber un comportement dirigé par un but en prenant des initiatives.

Une autre définition très courante est la définition de Ferber [Ferber, 1995] où un agent est une entité physique ou virtuelle qui :

- est capable d'agir dans un environnement ;
- peut communiquer directement avec d'autres agents ;
- est mue par un ensemble de tendances (sous la forme d'objectifs individuels ou d'une fonction de satisfaction, voire de survie, qu'elle cherche à optimiser) ;
- possède des ressources propres ;
- est capable de percevoir (mais de manière limitée) son environnement ;
- ne dispose que d'une représentation partielle de cet environnement (et éventuellement aucune) ;
- possède des compétences et offre des services ;

1. Dans ce manuscrit, nous proposons une traduction de la définition anglaise suivante : *Perhaps the most general way in which the term agent is used is to denote a hardware or (more usually) software-based computer system that enjoys the following properties:*
   - *autonomy: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;*
   - *social ability: agents interact with other agents (and possibly humans) via some kind of agent-communication language;*
   - *reactivity: agents perceive their environnement (which may be the physical world, a user via a graphical user interface, a collection of other agents, the Internet, or perhaps all of these combined), and respond in a timely fashion to changes that occur it;*
   - *pro-activeness: agents do not simply act in reponse to their environment, they ara able to exhibit goal-directed behaviour by taking the initiatives.*

– peut éventuellement se « reproduire » ;
– a un comportement qui tend à satisfaire ses objectifs, en tenant compte des ressources et des compétences dont elle dispose, et en fonction de sa perception, de ses représentations et des communications qu'elle reçoit.

I.1.1.2 Et la simulation?

Le terme de simulation informatique peut avoir différentes significations. Ainsi, comme le signale Varenne[Varenne, 2001], Pritsker [Pritsker, 1979] a répertorié 21 définitions différentes. Nous retiendrons dans ce manuscrit la définition de Drogoul [Drogoul, 1993] : « On nomme simulation la démarche scientifique qui consiste à réaliser une reproduction artificielle, appelée modèle, d'un phénomène réel que l'on désire étudier, à observer le comportement de cette reproduction lorsqu'on en fait varier certains paramètres, et à induire ce qui se passerait dans la réalité sous l'influence de variations analogues. » Cette démarche est résumée par la figure I.1.1.

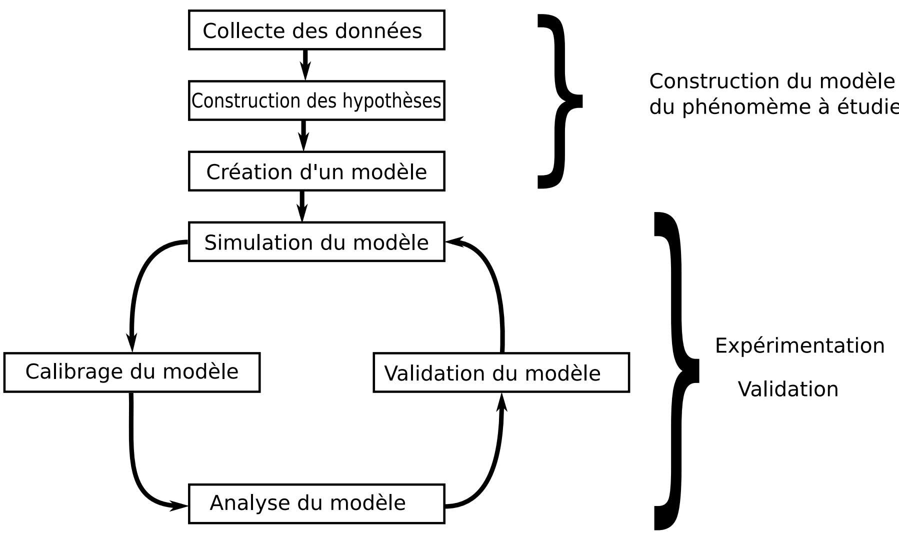


FIG. I.1.1 – *Principe de la simulation*

Pour définir la simulation, il faut aussi évoquer la notion de modélisation. D'après [Coquillard et Hill, 1997], « un modèle est une image simplifiée de la réalité, forgée à partir d'une certaine sélection des données d'observation et d'un certain nombre d'hypothèses. Ces dernières concernent l'importance présumée des faits à retenir et, si le modèle est explicatif, les mécanismes probablement en jeu » Minsky [Minsky, 1965] propose une définition de modèle[2] : « Pour un observateur B, un objet A^* est un modèle d'un objet A dans la mesure où B peut utiliser A pour répondre à des questions qui l'intéressent au sujet de A^* ». En résumé, nous retiendrons qu'un modèle est une représentation simplifiée de la réalité, élaborée en vue de répondre à certaines questions.

I.1.2 Simulation à base d'agents : exemple et intérêts

Dans cette section, nous allons caractériser la simulation à base d'agents et en montrer les principales qualités, notamment en comparaison avec d'autres approches de simulation.

2. Dans le manuscrit, nous proposons une traduction de la définition suivante : « *To an observer B, an object A^* is a model of an object A to the extent that B can use A^* to answer questions that interest him about A*» (définition citée dans de nombreuses thèses sur la simulation multi-agent [Servat, 2000, Duboz, 2004, Calderoni, 2002]...).

I.1.2.1 Exemple

À cette fin, nous allons utiliser un exemple concernant l'évolution de deux populations : le système proie-prédateur.

Historiquement, ce système a été représenté par un modèle à base d'équations différentielles. Avant de s'intéresser au modèle à base d'agents, nous nous intéressons d'abord à la modélisation continue dont les équations différentielles sont les plus grands représentants. Le modèle, constitué d'équations différentielles, représente l'ensemble des mêmes entités composant le système par une grandeur moyenne : c'est une modélisation par agrégation. Ces équations décrivent l'évolution de ces moyennes. Ce type de modélisation traduit une conception descendante ou *top down*. Il est possible de distinguer différentes sortes d'équations différentielles [Servat, 2000] : ordinaires, aux dérivées partielles, stochastiques... Un exemple de modélisation de l'évolution de deux populations par ce type d'approche est le modèle de Lotka-Volterra [Lotka, 1925, Volterra, 1926]. Celui-ci décrit les interactions entre une population de prédateurs et une population de proies à travers le système d'équations différentielles suivant :

$$\begin{cases} \dfrac{\mathrm{d}N_1}{\mathrm{d}t} = b_1 N_1 - k_1 N_1 N_2 \\ \dfrac{\mathrm{d}N_2}{\mathrm{d}t} = k_2 N_1 N_2 - d_2 N_2 \end{cases}$$

Nous avons dans ce modèle une agrégation du nombre d'individus de chaque population. La première équation représente l'évolution de la population des proies : le premier terme b_1N_1 correspond à la reproduction des proies avec un taux b_1. Le second terme $k_1N_1N_2$ reflète la fréquence d'interaction entre les proies et les prédateurs : à chaque fois qu'une proie rencontre un prédateur, elle disparaît. La deuxième équation représente l'évolution de la population de prédateurs : le premier terme $k_2N_1N_2$ correspond à la reproduction des proies avec un taux dépendant de la fréquence d'interaction entre les proies et les prédateurs. Le deuxième terme d_2N_2 correspond au taux de mortalité des prédateurs avec une taux d_2. La figure I.1.2 page suivante montre l'évolution des populations des proies et de prédateurs : en choisissant correctement les paramètres, il est possible d'observer des oscillations des populations.

Les équations différentielles font partie des modélisations continues. Une approche complémentaire est constituée par les modélisations discrètes, et plus particulièrement les automates cellulaires [Von Neumann, 1966, Burks, 1971]. Un automate cellulaire est composé d'un ensemble d'automates à états finis répartis sur les noeuds d'un réseau à d dimensions. Chaque automate, appelé cellule, calcule son état à l'instant $t + 1$ en fonction de son état et de ceux des automates de son voisinage à l'instant t. La spécification de l'automate est simple d'apparence, mais insuffisante pour permettre d'en déduire la dynamique globale : un automate peut avoir un comportement global complexe, nécessitant la simulation pour l'appréhender. Un exemple d'automate cellulaire est le jeux de la vie de John Conway [Gardner, 1970], sans doute le plus célèbre, dans un espace de dimension deux. Les règles d'évolution sont très simples : une cellule devient active si et si seulement trois de ses cellules voisines sont actives ; une cellule reste active si et si seulement deux ou trois de ses voisines sont actives, sinon elle devient inactive. Avec ces règles très simples, il est possible d'obtenir un comportement très complexe : « *glider* », canon à « *glider* »... La figure I.1.2.1 montre une visualisation d'un jeu de la vie. Les automates cellulaires sont très utilisés dans la modélisation de systèmes complexes [Ganguly et al., 2003], et il est notamment possible de modéliser le modèle de Lokta-Volterra par un automate cellulaire [Cattaneo et al., 2006]. Il présente cependant un certain nombre d'inconvénients : problème de découpage et discrétisation de l'espace, calcul du voisinage, conditions initiales (très importantes pour les automates cellulaires)...

Ce modèle de Lotka-Volterra peut être aussi représenté en utilisant d'autres modélisations discrètes, telle que l'approche à base d'agents. Le principe de cette modélisation est de décomposer un phénomène réel en un ensemble d'éléments qui agissent et inter-agissent, puis de modéliser chacun de ces éléments, leurs comportements et leurs interactions, de manière à essayer de produire une construction artificielle aussi proche que possible de la réalité étudiée. Le modélisateur construit, à partir de chaque élément du phénomène étudié, un agent, autrement dit, une entité logicielle dont il décrit les différents états possibles et les règles de comportement, mettant en relation état interne et perceptions externes pour en déduire les décisions prises, qui elles-mêmes peuvent se traduire par une modification de l'état interne de l'agent, et des actions sur son environnement et les autres

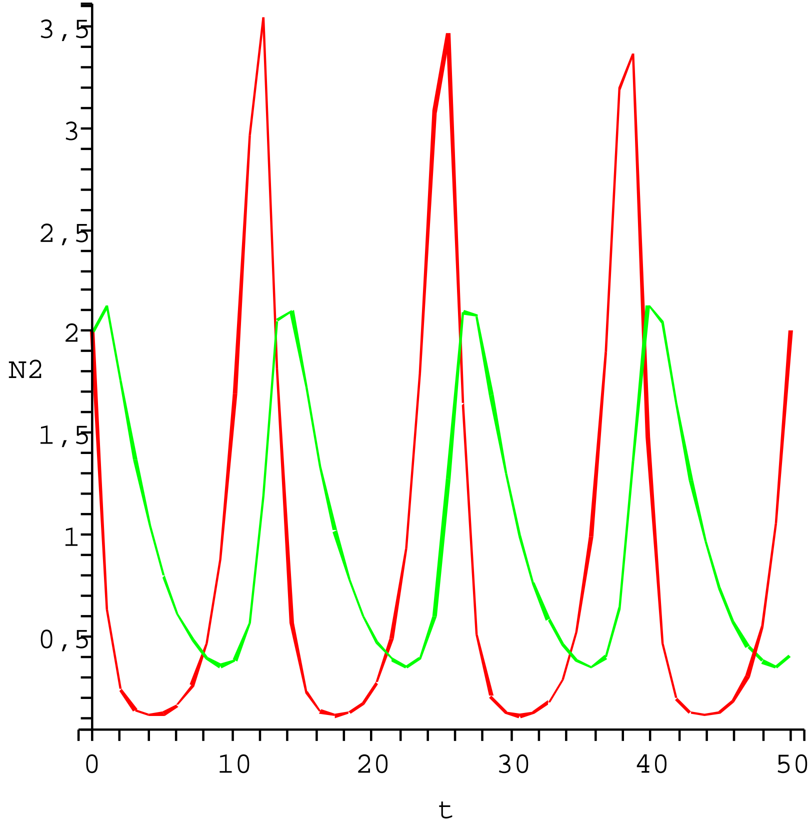


FIG. I.1.2 – *Évolution des populations de proies et de prédateurs par le modèle de Lotka-Volterra*

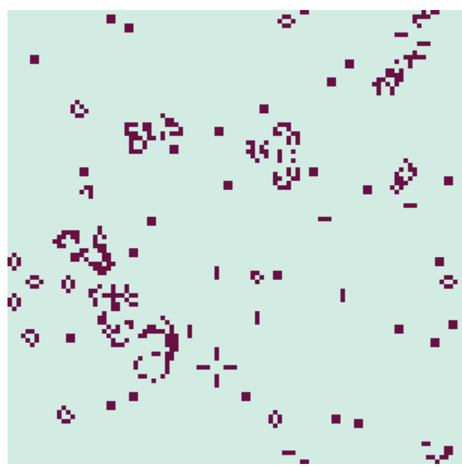


FIG. I.1.3 – *Jeu de la vie*

agents. Le modèle général est la résultante des actions et des interactions entre les agents. La dynamique du système est le résultat des actions locales des agents et des interactions entre agents. Nous pouvons appliquer cette démarche pour le modèle proie-prédateur. Chaque proie est représentée par un agent, de même que chaque prédateur. Les agents sont placés dans un environnement où ils se déplacent. Lors de leurs déplacements, si une proie rencontre un prédateur, elle est mangée par celui-ci. Ce comportement correspond au terme $k_1 N_1 N_2$ du système d'équations différentielles. Il faut ajouter un mécanisme pour prendre en compte la mortalité des prédateurs (le terme $d_2 N_2$ dans le modèle à base d'équations différentielles) : nous ajoutons des points de vie pour chaque agent prédateur. À chacune de leurs actions, ils perdent des points de vies. Quand ils attrapent une proie, leur nombre de points de vie augmente. Si ce nombre devient négatif, ils disparaissent. Il faut de la même manière prendre en compte la reproduction des proies (le terme $b_1 N_1$ du système d'équations différentielles). Chaque agent proie a une probabilité, à chaque pas de simulation, de se cloner. Tout ce travail a déjà été réalisé [Wilensky et Reisman, 2006] : Wilensky a ainsi utilisé la plate-forme multi-agent `NetLogo` [Wilensky, 1999] (voir l'annexe A page 119) pour développer le modèle « *Wolf Sheep Predation* » [Wilensky, 1998g] (voir l'annexe B.6 page 132). Le modèle proie-prédateur a été instancié dans ce cas-là sur la prédation de loups sur des moutons. En calibrant le modèle, il est possible d'obtenir les mêmes oscillations que dans le modèle de Lotka-Volterra à base d'équations différentielles ; ce comportement oscillatoire est cependant très difficile à obtenir avec le modèle à base d'agents.

I.1.2.2 Discussion

Nous avons rapidement montré, sur un même exemple, comment différentes approches peuvent être employées pour la modélisation de systèmes complexes. Nous pouvons maintenant nous intéresser aux avantages et inconvénients de chacune de ces méthodes, et plus particulièrement pour la simulation à base d'agents et les équations différentielles.

Le principal avantage des équations différentielles [Duboz, 2004] est que cette modélisation se fonde sur une théorie mathématique et des outils développés et utilisés depuis de nombreuses années. Ainsi, il existe de nombreux outils pour l'analyse et la résolution analytique et numérique des équations différentielles. De plus, les équations différentielles manipulent des valeurs continues, ce qui, pour certains systèmes, permet de confronter directement les valeurs obtenues avec des données réelles du système modélisé (concentration chimique, par exemple). Il est également possible d'augmenter le nombre d'entités dans le système sans ajouter de charge de calcul supplémentaire. Par exemple, pour le modèle précédent, multiplier par 100 le nombre d'individus de chaque population n'augmente pas le temps de calcul pour le système d'équations différentielles, alors que pour la simulation multi-agent, ce n'est pas le cas : le temps de calcul va augmenter de manière significative, de même que l'utilisation de la mémoire.

Mais les équations différentielles présentent aussi des inconvénients [Calderoni, 2002] : d'une part, elles manquent souvent de réalisme et d'autre part elles ne permettent une analyse qu'à un niveau global du fait de leur approche par agrégation.

Ce premier inconvénient, le manque de réalisme, est dû à l'approche des équations différentielles : elles obligent à une agrégation, et si le modélisateur veut augmenter le degré de réalisme de son modèle, il sera obligé de complexifier ses équations en distinguant différentes sous-populations dans le système étudié. Nous pouvons illustrer ces constats sur le modèle proie-predateur. Le modèle à base d'agents issu de `NetLogo` semble plus réaliste qu'un système de deux équations différentielles (même si le modèle à base d'agents est aussi très loin de la réalité : par exemple, avec la reproduction qui se résume à du clonage...). Comme le montre la figure I.1.2 page précédente, le système d'équations différentielles considère les différentes variables comme continues : cela peut être correct dans le cas général, mais c'est problématique pour représenter le nombre d'individus dans une population. La notion de « quart de mouton » n'est pas très réaliste : en-dessus de 1, il y a extinction de la population et donc il est impossible d'obtenir une explosion du nombre d'individus de cette population par la suite, alors que c'est possible avec le modèle à base d'équations différentielles. Ceci explique en grande partie la difficulté qu'il y a à calibrer le modèle à base d'agents pour observer le comportement oscillatoire du nombre d'individus des deux populations. Le fait de discrétiser permet d'avoir des comportements différents qu'une approche continue : extinction ou survie d'une population [Shnerb et al., 2000, Hutzler, 2007]. De même, Wilson [Wilson, 1998] a comparé la simulation multi-agent et les équations différentielles sur un même modèle. Les résultats obtenus

par la simulation multi-agent sont généralement plus réalistes : sur son modèle à base d'équations différentielles, Wilson a été obligé d'ajouter de la stochasticité.

Le second inconvénient à savoir que les équations différentielles ne permettent qu'une analyse à un niveau global, rend difficile la modélisation des actions au niveau individuel ainsi que des effets spatio-temporels. Au contraire, ces aspects individuels et spatio-temporels sont les deux principaux avantages de la simulation à base d'agents. En effet, il est possible de repérer individuellement chaque entité du système, et de la localiser. Les modèles à base d'agents permettent de modéliser les actions des entités du système et des interactions entre elles. Ainsi, les hétérogénéités locales du système [Hutzler, 2007] peuvent être modélisé, ce qui est difficilement réalisable avec les équations différentielles. L'utilisateur modélise le comportement des entités au niveau individuel et leurs interactions, et non, comme les équations différentielles, l'évolution globale du système. Parunak et ses coauteurs [Parunak et al., 1998] ont comparé la simulation multi-agent et les équations différentielles sur un même modèle. En plus de l'aspect spatial, la simulation multi-agent permet une validation supplémentaire par rapport aux équations différentielles : non seulement, il est possible de valider au niveau global, mais il est aussi possible de valider les résultats au niveau local. Avec ses aspects spatiaux et individuels, la simulation multi-agent permet de modéliser les systèmes complexes [Parker et al., 2003, Macal et North, 2005], qui sont caractérisés par des interdépendances entre les entités du système, à travers le temps et l'espace, et aussi de modéliser l'émergence (« le tout est plus grand que la somme des parties »).

Avec le modèle multi-agent, il est possible d'ajouter facilement des comportements, des hypothèses supplémentaires. Par exemple, dans le modèle « *Wolf Sheep Predation* » est ajouté un comportement supplémentaire pour les proies : les proies (dans le cadre de ce modèle, des moutons) peuvent manger l'herbe, qui elle-même pousse à une certaine vitesse et avec une taille maximale. En calibrant le modèle sont obtenues des oscillations pour les trois types de populations. Nous pourrions essayer de faire de même avec le modèle à base d'équations différentielles. Un tel système pourrait être :

$$\begin{cases} \frac{\mathrm{d}N_1}{\mathrm{d}t} = b_1 N_1 N_3 - k_1 N_1 N_2 \\ \frac{\mathrm{d}N_2}{\mathrm{d}t} = k_2 N_1 N_2 - d_2 N_2 \\ \frac{\mathrm{d}N_3}{\mathrm{d}t} = b_3 N_3 - k_3 N_1 N_3 \end{cases}$$

avec N_3 la quantité totale d'herbe disponible. Il faudrait alors à nouveau résoudre le système. Avec un modèle à base d'équations différentielles, ajouter des hypothèses est souvent synonyme de grands remaniements nécessaires dans la modélisation : il faut modifier globalement les équations (la première équation a dû être modifiée pour prendre en compte la présence d'herbe) et, dans le cas d'une résolution analytique, il faut également refaire l'étude. Par contre, avec un modèle à base d'agents, il est facile d'ajouter des hypothèses grâce à une approche incrémentale. Pour prendre en compte la présence d'herbe dans le système, il suffit ainsi d'ajouter un nouveau type d'agent, de déterminer ses règles d'évolution, et d'enrichir le répertoire comportemental des autres agents afin qu'ils prennent éventuellement en compte ce nouveau type. Enfin, du fait d'une part de la très grande similitude entre le modèle à base d'agents et le système réel, et d'autre part de la facilité de poser de nouvelles hypothèses, il sera relativement aisé de reproduire en simulation des expériences menées habituellement sur le système réel, voire d'en imaginer de nouvelles, impossibles à réaliser sur le système réel, d'où la notion de "laboratoire virtuel" que l'on peut attacher à l'approche à base d'agents.

Le principal inconvénient de la simulation à base d'agents est le manque de formalisme, surtout en comparaison aux équations différentielles.

En résumé, en quoi les modèles à base d'agents peuvent-ils être intéressants? [Bonabeau, 2002]

- Quand le comportement des entités du système est non-linéaire, peut être caractérisé par des seuils, des conditions, ou des couplages non-linéaires, ou peut inclure de la mémoire, des corrélations temporelles, de l'apprentissage et de l'adaptation. Décrire des discontinuités dans un comportement individuel est difficile avec les équations différentielles.
- Quand la notion d'espace est cruciale et que la position des entités n'est pas fixe.
- Quand la population est hétérogène, voire quand chaque entité est (potentiellement) différente.
- Quand les interactions entre les agents sont complexes, non linéaires, discontinues, ou discrètes.

– Quand la moyenne ne fonctionne pas : l'agrégation des équations différentielles tend à lisser les fluctuations, ce que les modèles à base d'agents ne font pas. Ceci peut être très important : un système peut être linéairement stable sans perturbation, mais instable en présence de perturbations.

I.1.3 Calibrage/Validation

Après, la phase de construction du modèle vient la phase fondamentale d'expérimentation et de validation. Cette dernière phase est composée de trois étapes : la simulation, le calibrage et la validation. La simulation consiste seulement à faire agir les agents ensemble. Cependant les deux autres étapes sont plus complexes.

I.1.3.1 Calibrage

Lors de la construction d'un modèle multi-agent, et du simulateur associé, il y a une phase de calibrage des paramètres. Il est possible de distinguer deux sortes de paramètres. D'abord, il y a ceux liés au simulateur ou à la plate-forme de simulation. Par exemple, ce sont des paramètres liés à la discrétisation du temps ou de l'espace. Et généralement, ces paramètres ne sont pas modifiables par l'utilisateur. Puis, il y a les paramètres liés directement au modèle. En général, il faut en choisir les valeurs. Pour une partie de ces paramètres, le modélisateur dispose des valeurs précises car il est possible d'extraire ces valeurs de la connaissance du domaine. Pour d'autres paramètres, il ne possède que des valeurs plus ou moins précises car la connaissance du domaine pour ces paramètres n'est pas suffisante. Pour les paramètres restants, le modélisateur peut n'avoir aucune idée pour le choix des valeurs de ces paramètres car la connaissance du domaine est inexistante pour ce paramètre, ou non applicable dans le cadre de ce modèle. Le problème du calibrage consiste à déterminer des valeurs pour ces paramètres, ce qui pose la question conjointe de la validation de ce modèle avec ce jeu de paramètres.

L'une des difficultés spécifiques avec l'approche à base d'agents, liée au problème du calibrage, est l'absence de formalisation. Dans le cas des équations différentielles, lorsque le système d'équations peut être résolu de manière analytique, il est alors possible d'étudier les classes de comportements du système en fonction des valeurs de paramètres, grâce à des outils mathématiques, et sans avoir à tester individuellement chaque jeu de paramètres. Une telle approche mathématique n'est pas applicable dans le contexte de la modélisation à base d'agents et il sera donc nécessaire de développer des stratégies spécifiques d'exploration de l'espace des paramètres.

I.1.3.2 Validation : qu'est-ce un système valide ?

L'une des étapes les plus importantes du processus de modélisation concerne la validation du modèle [Troitzsch, 2004] : il existe principalement trois grandes méthodes de validation de modèle. La première consiste à confronter les résultats de la simulation avec des données réelles. Cette méthode est souvent utilisée dans le domaine de l'économie [Sallans et al., 2003, Windrum et al., 2007, Bianchi et al., 2007a, Bianchi et al., 2007b] qui possède beaucoup de données sur les phénomènes à simuler. Le modélisateur crée un modèle, puis il compare les résultats de la simulation du modèle avec les données réelles. Et si les résultats de la simulation concordent avec ces données, il peut envisager de faire des prédictions à partir des résultats de simulation.

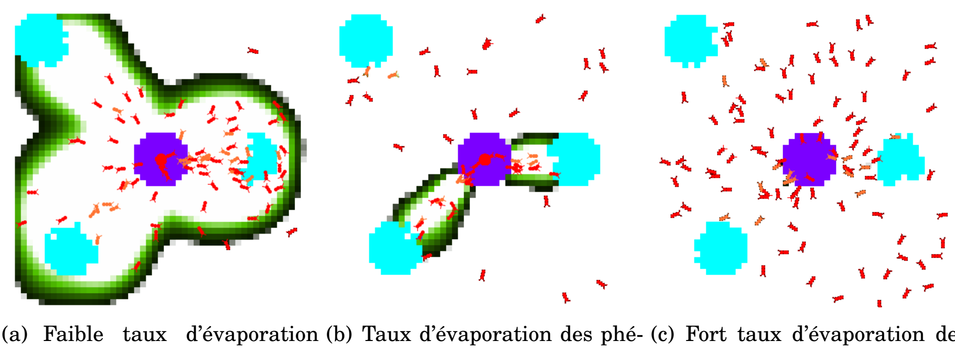
Une autre méthode est d'avoir un modèle mathématique décrivant le phénomène à simuler. Il est possible d'illustrer cette approche en reprenant l'exemple du modèle proie-prédateur de la sous-section I.1.2.1 page 5 : le comportement global du système est décrit par un système d'équations différentielles. Le but du modélisateur est de reproduire ce comportement par un modèle à base d'agents. Donc, il crée ce modèle, puis, pour le valider, il va comparer les résultats obtenus par simulation avec les résultats théoriques obtenus par l'approche mathématique.

Ces deux premières méthodes supposent d'avoir soit beaucoup de données réelles, soit un modèle mathématique. Généralement, les modélisateurs ne sont dans aucun de ces deux cas-là. Alors, une troisième méthode pour valider un modèle est l'alignement de modèles (*docking*) [Axtell et al., 1996] : dans cette méthode, le même modèle est créé plusieurs fois de façon indépendante (par exemple, par des équipes différentes), puis, les résultats de simulation des différents modèles sont comparés :

si les résultats obtenus concordent, il est possible de valider le modèle [Takadama et al., 2003]. Il est possible aussi qu'une même équipe implante le même modèle sur différentes plate-formes de simulation, puis compare les résultats de simulation des différents modèles [Xu et al., 2003].

I.1.3.3 Exemple sur le fourragement : pourquoi est-ce un problème intéressant ?

Il paraît opportun de souligner l'importance de la phase de calibration. L'exemple est la modélisation du comportement des fourmis lors du fourragement. Dans la nature, des files de fourmis sont observées lors de l'exploitation de sources de nourriture par une fourmilière. Le but de cette modélisation est d'observer ce comportement. Pour cela, afin de reproduire ce phénomène est créé un modèle à base d'agents en utilisant des règles très simples : quand une fourmi rapporte de la nourriture à la fourmilière, elle dépose des phéromones sur le sol du chemin de retour. Quand une fourmi part à la recherche de nourriture, si elle perçoit des phéromones, elle en suit les traces. Sinon, dans le cas où elle ne perçoit rien, elle fait une recherche aléatoire. En appliquant ces règles très simples (beaucoup plus simples que la réalité), le modèle obtenu est le modèle multi-agent « *Ants* » de `NetLogo` [Wilensky, 1998a] (pour plus de détails sur ce modèle, il faut se reporter à la section B.5 page 130). Ce modèle est caractérisé par un grand nombre de paramètres : vitesse des fourmis, angle de perception des phéromones, distance de perception des phéromones, taux d'évaporation des phéromones sur le sol, taux de diffusion des phéromones... Ainsi, il faut choisir des valeurs à ces paramètres : un des objectifs du calibrage est de trouver au moins un jeu de paramètres qui permette d'observer le comportement désiré c'est-à-dire les files de fourmis. Par exemple, si le taux d'évaporation des phéromones varie en maintenant constants les autres paramètres, il est possible d'obtenir deux comportements : une recherche aléatoire de nourriture, et une recherche via des files de fourmis. Si le taux d'évaporation est faible, alors les phéromones saturent tout l'environnement, et donc les fourmis effectuent une recherche aléatoire de nourriture. Si le taux d'évaporation est fort, alors il n'y a pas de phéromones sur l'environnement, et donc les fourmis effectuent encore une recherche aléatoire de nourriture. C'est seulement quand l'évaporation est moyenne que des files de fourmis sont observables. La figure I.1.3.3 montre les différences de comportements en fonction de la quantité de phéromones. Cet exemple montre l'importance de calibrer correctement son modèle.



(a) Faible taux d'évaporation des phéromones. (b) Taux d'évaporation des phéromones moyen. (c) Fort taux d'évaporation des phéromones.

FIG. I.1.4 – *Évolution du comportement des fourmis en fonction du taux d'évaporation des phéromones*

I.1.4 Conclusion et objectif de la thèse

Nous avons montré dans ce chapitre que les modèles à base d'agents constituent une approche de modélisation intéressante. Cependant une phase délicate, lors de la conception d'un modèle à base d'agents, est le calibrage de ce modèle : un même modèle peut avoir des comportements totalement différents en fonction des valeurs de paramètres.

Réaliser un modèle recouvre différents objectifs. Le but le plus classique est d'avoir un modèle aussi proche que possible des données réelles. L'intérêt de cet objectif est de préparer la phase de

validation du modèle. Par exemple, l'utilisateur a un phénomène à modéliser avec beaucoup de données : le but de la phase de calibrage sera de réduire la distance entre le phénomène réel et le résultat de la simulation. Le modèle multi-agent simulé peut réaliser une fonction. Dans ce cas-là, un autre type d'objectif pourrait être d'optimiser le comportement du système, c'est-à-dire de maximiser l'efficacité de la fonction accomplie par le système. Par exemple, pour le modèle de fourragement par colonie de fourmis, un des buts serait d'optimiser la vitesse pour ramener la nourriture à la fourmilière. Un autre objectif est de déterminer les différentes classes de comportement d'un modèle. Par exemple, pour le modèle de fourragement par colonie de fourmi, il est possible de distinguer deux grands types de comportements : un premier comportement serait la recherche de nourriture avec des files de fourmis, un second serait la recherche de nourriture de façon aléatoire. Un objectif peut aussi être de tester des hypothèses : des hypothèses sont faites sur la modélisation d'un phénomène, et le modélisateur veut vérifier ces hypothèses.

Le modélisateur propose au final un modèle et ses objectifs dans le calibrage. Le but de ce manuscrit est de proposer des méthodes d'exploration de l'espace des paramètres au modélisateur afin de calibrer son modèle suivant ses objectifs de façon plus ou moins automatique.

I.2 Espace de paramètres

Nous allons nous intéresser dans ce chapitre à l'espace des paramètres d'un modèle à base d'agents, plus particulièrement au calibrage.

Lors de la conception d'un modèle (quelqu'en soit le type : modèle multi-agent, système d'équations différentielles), il y a toujours une phase de paramétrage. Comme nous l'avons évoqué précédemment, dans le cadre de modèles mathématiques que l'on peut étudier analytiquement, le problème du paramétrage revient à une étude mathématique d'un ensemble d'équations. Cependant, lorsque le nombre de variables et d'équations augmente, une résolution analytique devient rapidement impossible et il faut alors se tourner soit vers des techniques spécifiques d'estimation des paramètres, soit vers une résolution numérique des équations. Cette dernière approche nécessite alors une exploration empirique et expérimentale de l'espace des paramètres. De même dans le cadre de la modélisation à base d'agents, il n'est pas possible d'étudier de manière formelle et théorique l'influence de tel ou tel paramètre et il faudra alors explorer l'espace des paramètres en choisissant des jeux de paramètres et en simulant les modèles correspondants.

Une des méthodes les plus intuitives pour ce faire est de parcourir tout l'espace des paramètres. Cette méthode présente cependant deux inconvénients majeurs. Le premier est que le nombre de jeux de paramètres à évaluer augmente de manière très rapide avec les nombres de paramètres et avec le nombre de valeurs possibles pour chaque paramètre : avec seulement trois paramètres pouvant prendre chacun 100 valeurs différentes, il existe déjà un million de combinaisons possibles ! Le deuxième défaut est que l'évaluation d'un jeu de paramètres nécessite la simulation du modèle correspondant, ce qui signifie un temps d'évaluation qui est loin d'être négligeable. Une manière de contourner ce dernier problème consisterait à construire un modèle mathématique du modèle à base d'agents, ce qui pose également un certain nombre de problèmes.

Certains auteurs [Edmonds et Bryson, 2004] affirment ainsi l'insuffisance des méthodologies formelles pour créer des modèles multi-agents : « nous affirmons que n'importe quelle méthodologie purement conceptuelle sera confrontée à d'insurmontables difficultés dans les systèmes multi-agents ouverts et complexes d'aujourd'hui. Nous recommandons, à la place, une méthodologie fondée sur la méthode expérimentale classique. »[1].

S'il est difficile de formaliser un modèle à base d'agents au stade de la conception, il serait possible de formaliser a posteriori. Ainsi à partir des simulations, une idée serait de créer un nouveau modèle (un méta-modèle), équivalent au modèle à base d'agents, plus formel, qui, en fonction des mêmes entrées (c'est-à-dire les valeurs de paramètres), redonne les mêmes sorties (c'est-à-dire les résultats de la simulation). Cette idée a été un peu explorée durant mon DEA [Calvez, 2004]. L'idée est de réaliser de nombreuses simulations avec des jeux de paramètres différents, puis à partir de ces jeux de paramètres (entrées du système), et de ces résultats de simulation (sorties du systèmes), de créer un nouveau modèle qui prend en entrée un jeu de paramètres et qui en sortie redonne le résultat de la simulation. Ce nouveau modèle [Jin, 2005] peut se faire de différentes manières, parmi lesquelles :

- par un modèle polynomial [Ingu et Takagi, 1999] ;
- par un réseau de neurones [Jin et al., 2002] ;
- par un processus gaussien [Büche et al., 2004].

La figure I.2.1 page suivante en montre le principe.

Lors de mon DEA, c'est un réseau de neurones que nous avons utilisé comme le préconise Jin [Jin, 2005] quand la dimension de l'espace d'entrée est grande, et que le nombre de données d'apprentissage est faible. Cependant les résultats ont été peu probants (pour plus de détails, se reporter à [Calvez, 2004]). Ces résultats peu intéressants sont sûrement dûs au fait que le méta-modèle essayant de modéliser le modèle n'était pas assez complexe. Mais si le nouveau modèle qui est construit à partir

1. Dans ce manuscrit, nous proposons une traduction de la définition anglaise suivante : « We then argue that any pure design methodology will face insurmountable difficulties in today's open and complex MAS. Rather we suggest a methodology based on the classic experimental method ».

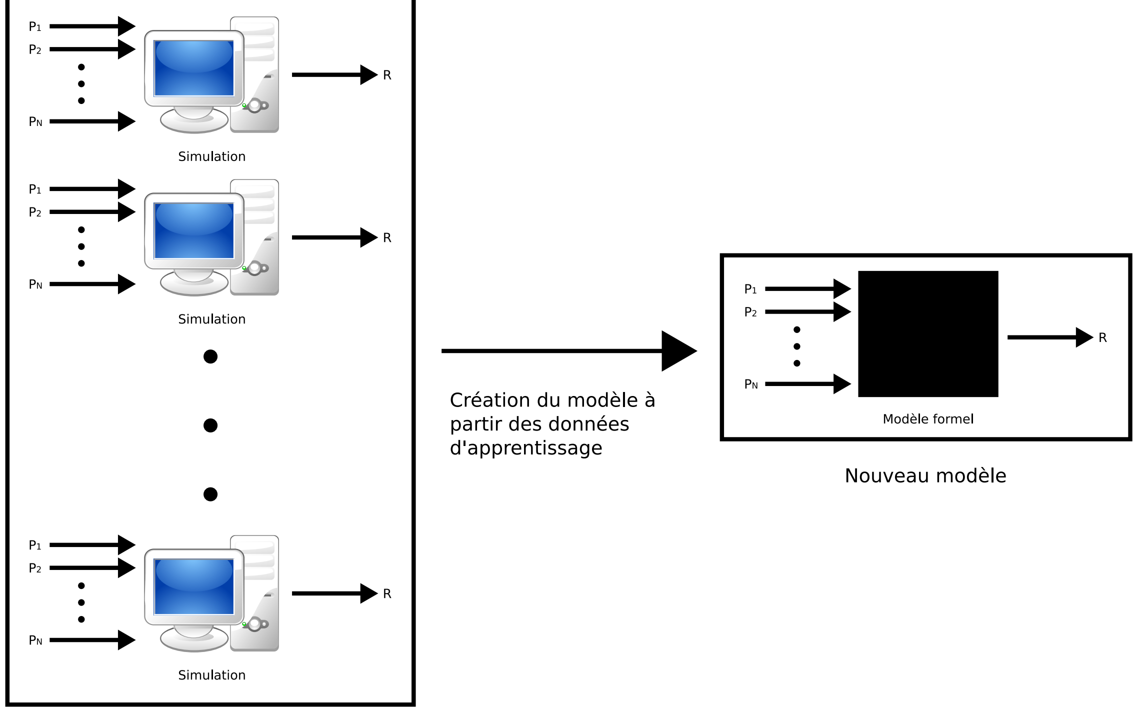


FIG. I.2.1 – *Principe de construction d'un méta-modèle*

du modèle multi-agent devient trop complexe (voire plus complexe que le modèle multi-agent), cette démarche devient inintéressante.

Il y a d'autres travaux sur cette idée de méta-modèle pour un modèle multi-agent [Klein et al., 2005, Klein et al., 2006] : l'inconvénient est qu'il faut imposer de grandes contraintes sur le système. Pour réduire le recours à des simulations, De Wolf et ses co-auteurs [De Wolf et al., 2005] proposent de coupler à une simulation à base d'agents une technique d'analyse macroscopique, comme par exemple, le paradigme « *equation free* » développé dans l'article de Kevrekidis et ses co-auteurs [Kevrekidis et al., 2004]. Dans cette approche, des mesures macroscopiques sont effectuées durant la simulation, puis sont analysées par l'approche « *equation free* ». À partir de cette analyse, la simulation peut être accélérée : une partie des pas de simulation est réalisée à partir du simulateur, puis à partir des données obtenues par les pas de simulation précédents, l'autre partie des pas de simulation est calculée grâce à l'analyse « *equation free* ». Cependant, cette approche se fonde sur l'observation des tendances globales du modèle qui sont supposées évoluer graduellement, ce qui n'est pas forcément le cas avec les systèmes complexes. Plus important, cette approche requiert une bonne compréhension des relations entre les comportements macroscopiques du système et les comportements microscopiques de ses constituants, ce qui est généralement l'un des objectifs de recherche lors de la création de modèles à bases d'agents.

En conclusion, l'idée d'essayer d'abstraire un modèle mathématique à partir du modèle à base d'agents semble difficilement applicable dans le cas général, ce qui rend les méthodes de calibrage à base de modèles mathématiques elles aussi difficilement applicables à l'approche agent.

I.2.1 Calibrage pour les modèles à base d'agents

Nous allons nous intéresser dans cette section au calibrage de simulations multi-agents en général.

Le plus souvent, la phase de calibrage d'un modèle à base d'agents est faite « à la main », comme par exemple, dans l'article de Antunes et ses co-auteurs[Antunes et al., 2006a]. Ce constat est encore plus fréquent dans le cas de l'utilisation de plates-formes de simulation telles que `NetLogo`. Par exemple, sur cette plate-forme, il est très simple de jouer avec la glissière (*slider*) d'un paramètre

pour en ajuster la valeur.

Il est aussi possible d'automatiser cette tâche par une fonctionnalité intéressante de `NetLogo` : *BehaviorSpace*. C'est un outil qui permet l'exploration automatique et systématique de l'espace des paramètres. À l'aide soit d'une interface graphique, soit d'une ligne de commande, soit d'un fichier au format `XML`, l'utilisateur choisit les paramètres qu'il désire faire varier, et précise leur valeur minimale, leur valeur maximale, et leur incrément pour chaque paramètre qui doit varier, et indique ce qu'il veut en sortie (par exemple, le nombre d'agents). L'outil *BehaviorSpace* exécute ensuite toutes les simulations avec les différents jeux de paramètres, et renvoie les résultats. La figure I.2.2 montre l'outil *BehaviorSpace* en mode avec interface graphique. Cependant, l'approche systématique n'est pas utilisable pour un modèle un peu complexe, car le nombre de simulations à réaliser devient énorme. À la différence d'un utilisateur qui calibre « à la main » un modèle, et qui ne va pas parcourir tout l'espace (par son intuition et son expérience, il sait que certains paramétrages ne sont pas intéressants), l'outil *BehaviorSpace* parcourt tout l'espace de façon systématique. Un autre exemple du même type d'outil est *Parameters Variation* dans `AnyLogic` [Technologies, 2007] : la figure I.2.3 page suivante montre l'interface graphique de l'outil *Parameters Variation*.

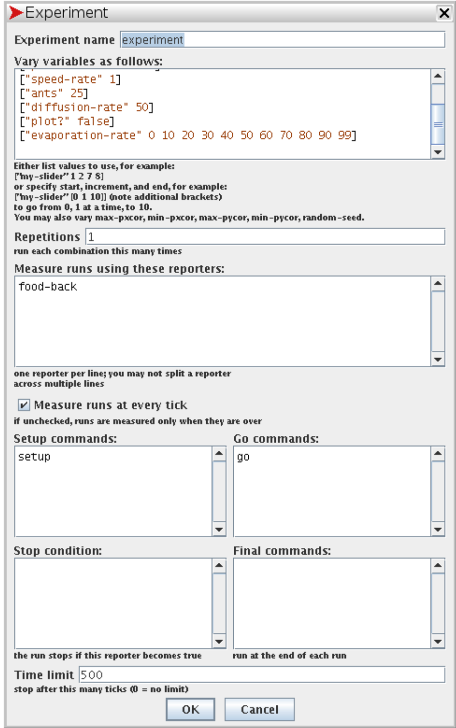


FIG. I.2.2 – *Outil BehaviorSpace*

Il est aussi possible de prévoir la partie de calibrage dès la conception du modèle. Par exemple, certains auteurs [Fehler et al., 2004, Fehler et al., 2006] proposent une calibrage en boîte blanche. Le principe est d'utiliser la connaissance du modèle à base d'agents pour améliorer le processus de calibrage. Le but est de réduire l'espace des paramètres en découpant le modèle en sous-modèles plus petits, ce qui peut être fait par différentes méthodes (« *General Model Decomposition* », « *Functional Decomposition* »...). Chaque sous-modèle est alors calibré, avant fusion de tous les sous-modèles afin de reformer le modèle entier. D'une part, l'opération de division requiert l'addition de connaissances sur le modèle, qui n'est pas forcément disponible. D'autre part, l'opération de fusion doit regrouper des sous-modèles calibrés en un modèle de plus haut niveau calibré, ce qui n'est pas forcément automatique. Une autre méthodologie est proposé par Antunes et ses co-auteurs [Antunes et al., 2006b] dans laquelle l'utilisateur doit calibrer les paramètres au moment de la conception du modèle.

Ces méthodes supposent de se trouver au tout début de la phase de conception d'un modèle. Lorsque le modèle est déjà conçu, il faut reprendre l'idée d'exploration de l'espace des paramètre, en évitant de tout explorer. C'est le cas de la méthode développée par Brueckner et Parunak[Brueckner et Parunak, 2003]. Celle-ci se fonde sur une infrastructure dite de balayage de paramètres, à la ma-

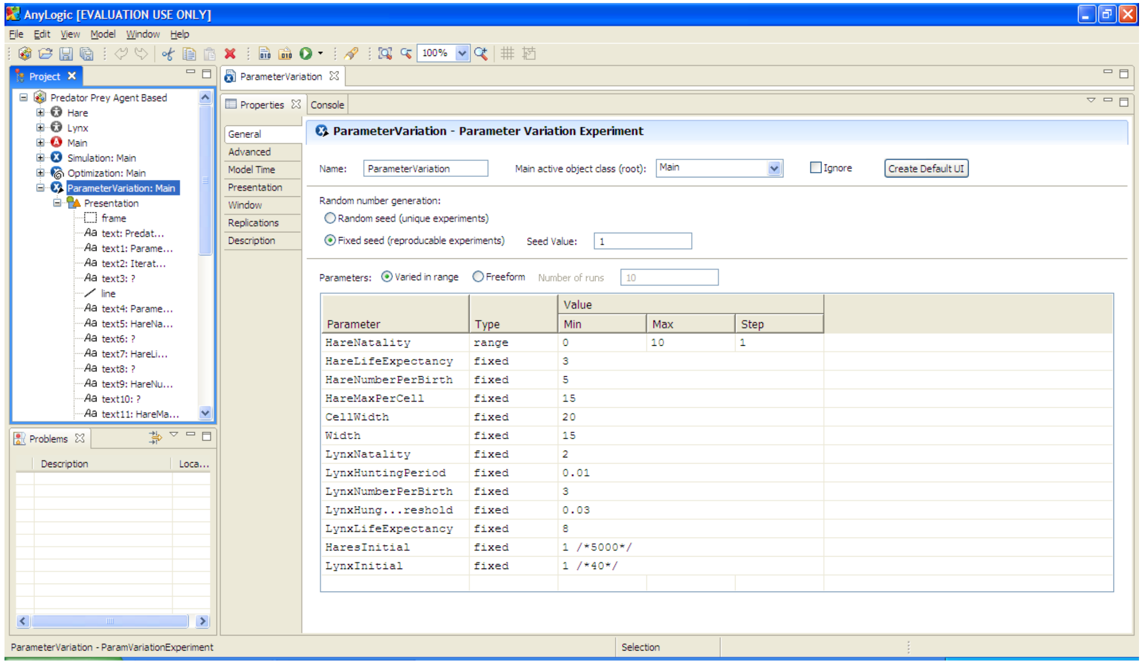


FIG. I.2.3 – *Outil Parameters Variation de* `AnyLogic`

nière du *BehaviorSpace* de `NetLogo`. Cependant, la notion de *fitness* est introduite pour caractériser la qualité d'un modèle et éviter ainsi une exploration systématique. Des agents chercheurs voyagent à travers l'espace des paramètres à la recherche de la *fitness* la plus haute. Partant d'un point précis de l'espace, les agents chercheurs ont deux choix : bouger ou simuler. Chaque agent choisit en fonction de la confiance qu'il a sur l'estimation de la *fitness* (proportionnelle au nombre de simulations en ce point) et de la valeur de la *fitness*. Si l'agent décide de bouger, il se dirige vers les zones voisines où la *fitness* est la plus haute. Un désavantage de cette méthode est qu'un agent chercheur peut tomber dans un maximum local de la *fitness*.

Un autre exemple est proposé par Sallans et ses co-auteurs [Sallans et al., 2003]. Pour créer un modèle avec de nombreux paramètres, une partie de ces paramètres est choisie à la main en faisant des simulations. Les autres paramètres sont choisis par une adaptation de la méthode de Monte-Carlo par chaînes de Markov pour faire une marche aléatoire dirigée à travers l'espace des paramètres, couplée avec un algorithme de Metropolis [Metropolis et al., 1953]. Cette méthode fonctionne bien sur un espace continu, mais sera difficilement utilisable quand l'espace des paramètres est chaotique et quand il y a de la stochasticité dans la simulation.

D'autres méthodes, au lieu de s'intéresser spécifiquement au calibrage, visent à étudier l'espace des paramètres, comme par exemple, l'analyse de sensibilité [Ginot et Monod, 2006]. Cette méthode permet de connaître la sensibilité de chaque paramètre. Mais, ce parcours de l'espace des paramètres est coûteux en simulations et ne résout pas le problème de calibrage.

I.2.2 Discussion

Ainsi que nous venons de le montrer dans les sections précédentes, en l'absence de méthodes formelles, calibrer un modèle à base d'agents est un problème difficile. En l'absence de modèle formel, il est nécessaire de réaliser des simulations du modèle avec différents jeux de paramètres. Or simuler un modèle multi-agent requiert du temps, et généralement une même simulation n'est pas « *multithreadable* », ni distribuable. Il faut donc absolument limiter le nombre de simulations, ce qui interdit de parcourir l'espace des paramètres de manière exhaustive. Terán et Edmonds [Terán et Edmonds, 2002, Terán et Edmonds, 2004] montrent que la complexité de l'exploration avec ajout de contraintes sur les trajectoires de simulation d'un modèle à base d'agents pour prouver des tendances dans la dynamique du modèle est coNP-complet. Le parcours de l'espace ne peut pas alors être systématique : il faudra une exploration de l'espace des paramètres de façon non-systématique.

Notre démarche va être de voir le problème d'exploration de l'espace des paramètres dans le but de calibrer un modèle à base d'agents comme un problème d'optimisation, comme le montre la figure I.2.4. Pour réaliser cette démarche, une fonction à optimiser est créée : elle prend en entrée un jeu de paramètres. Elle renvoie en sortie un résultat (c'est à la dire une *fitness*) qui dépendra des objectifs fixés par l'utilisateur pour calibrer son modèle (voir la section I.1.4 page 10) et du résultat de la simulation du modèle à partir du jeu de paramètres. Le but sera de minimiser (ou de maximiser) cette fonction.

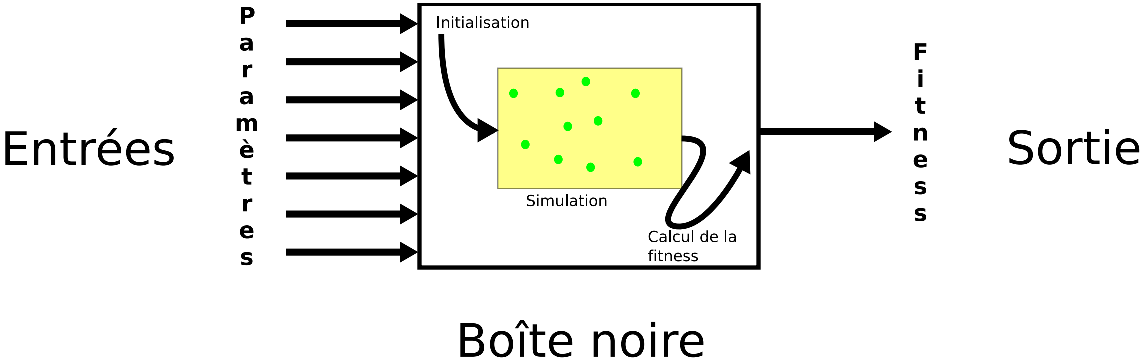


FIG. I.2.4 – *Vision entrées / sortie d'une simulation à base d'agents*

Le choix de la fonction objectif ou de la fonction de *fitness* dépend des buts de l'utilisateur. Par exemple, le modélisateur peut vouloir obtenir un modèle proche des données réelles : une fonction objectif pourrait être la distance entre les résultats du modèle simulé et de la réalité, du point de vue quantitatif. Du point de vue qualitatif, le modélisateur sait que le phénomène à modéliser fonctionne de telle manière, il veut obtenir ce même mode de fonctionnement dans le modèle à base d'agents. Par exemple, sur le modèle de fourragement par une colonie de fourmis, le phénomène intéressant à obtenir sur la simulation est l'apparition de files de fourmis. Il serait possible d'obtenir un modèle identique aux données réelles (par exemple, le temps moyen pour ramener la nourriture au nid. . .) sans file de fourmis. Ce modèle serait du point de vue quantitatif très bon, mais du point de vue qualitatif très mauvais. Cependant, il existe un problème pour optimiser un modèle du point de vue qualitatif : il faut d'abord traduire ce point de vue qualitatif en une mesure quantitative pour pouvoir avoir une fonction de *fitness* à optimiser. Caractériser un phénomène du point de vue qualitatif peut être difficile, d'autant plus que les phénomènes émergents et auto-organisés sont souvent le résultat d'interprétation subjective de la part de l'observateur [Moncion et al., 2006].

La fonction de *fitness* dépend aussi du résultat de la simulation du modèle à base d'agents. Or une simulation multi-agent est un processus dynamique, qui n'a pas forcément de fin, et qui évolue au cours du temps. Dans un problème d'optimisation classique, ce problème ne se pose pas : la fonction de *fitness* correspond au résultat d'un calcul et la question de déterminer quand arrêter le calcul n'a pas de sens : la mesure est faite quand le calcul est fini. Mais dans le cadre de la simulation multi-agent, la question de savoir à quel pas de simulation il faut calculer la *fitness* est importante. Si le modélisateur s'intéresse à un comportement transitoire, il faut peut-être répéter plusieurs fois une mesure de *fitness* à différents pas de temps. Ce problème peut être illustré sur l'exemple du fourragement par colonie de fourmis comme le montre la figure I.2.5 page suivante. Si, par exemple, la fonction de *fitness* est la quantité de nourriture ramenée avant n pas de simulation, et si n est choisi trop petit ($t \leq 30$) ou trop grand ($t \geq 600$), alors la valeur de la *fitness* sera la même quelque soit le jeu de paramètres choisi (sauf cas extrêmes) : si le calcul est effectué trop tôt, alors aucune nourriture n'est ramenée à la fourmilière, même pour le meilleur modèle. Si le calcul est effectué trop tard, alors toute la nourriture a été ramenée dans le nid, même pour le pire modèle. Il faut donc choisir le bon moment pour évaluer la fonction de *fitness* : il est donc nécessaire d'évaluer la fonction de *fitness* à différents pas de temps. De même, si la fonction de *fitness* est plus qualitative comme la présence de files de fourmis, alors si la fonction de *fitness* est évaluée trop tôt, ou trop tard, il n'y a pas de files de fourmis. De nouveau, c'est entre 60 et 420 pas de simulation qu'il faut calculer la fonction de *fitness*. Mais, le nombre de files évolue au cours de la simulation, ce qui suggère que la fonction de *fitness* devra être déterminée à différents pas de simulation sur cet intervalle.

En plus de l'aspect dynamique, il faut prendre en compte l'aspect stochastique des simulations

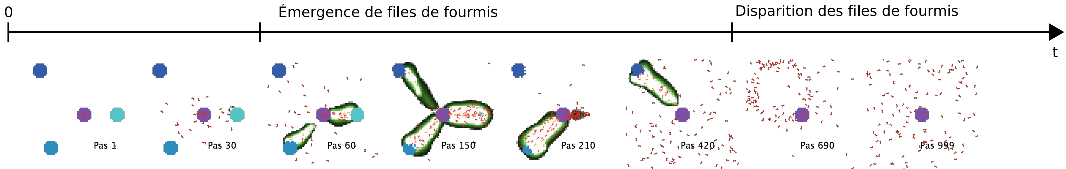


FIG. I.2.5 – *Évolution d'une simulation multi-agent*

multi-agents. Deux exécutions d'un même modèle multi-agent avec les mêmes paramètres ne fournit pas forcément les mêmes résultats à la fin de la simulation. Une seule exécution d'un modèle multi-agent ne donne qu'une estimation de la *fitness*. En fonction des modèles, il peut être nécessaire de simuler plusieurs fois le même modèle multi-agent pour avoir une *fitness* relativement précise. Il y a donc un choix à effectuer entre précision et temps de simulation.

I.2.3 Conclusion

En conclusion, l'objectif des méthodes proposées dans ce manuscrit est de calibrer un modèle à base d'agents en considérant cette exploration de l'espace des paramètres comme un problème d'optimisation, en se limitant au niveau du nombre de simulations et en tenant compte de la présence de stochasticité dans les modèles ou dans les simulateurs.

Dans cette démarche, il existe des outils qui permettent de calibrer un modèle comme un problème d'optimisation, comme par exemple l'outil *Optimization* (dérivant de *OptQuest*® [opttek, 1992] (basé sur une recherche « *scatter search* ») de la plate-forme commerciale de simulation `AnyLogic` [Technologies, 2007] ; mais ce sont des outils génériques, non spécifique à la simulation multi-agent, oubliant la stochasticité de la simulation à base d'agents. Sur les quelques tests que nous avons effectué sur cette plate-forme de simulation, nous avons rencontré à chaque fois des problèmes de consommation en excès de la mémoire. La figure I.2.6 montre l'interface graphique de l'outil.

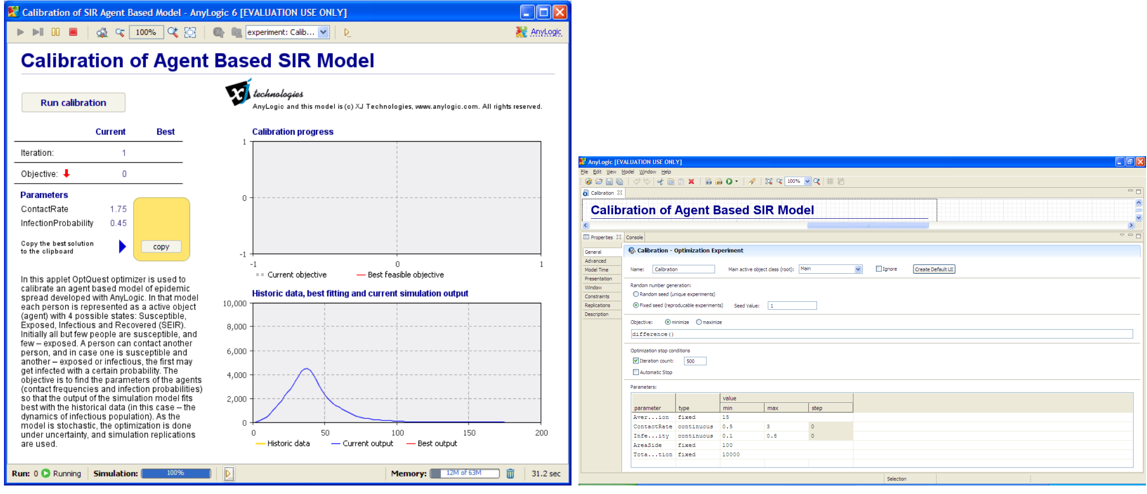


FIG. I.2.6 – *Outil Optimization de* `AnyLogic`

Les outils existants ne semblent donc pas répondre de manière satisfaisante aux problématiques qui nous intéressent et nous allons donc travailler à la création de nos propres outils. Puisque nous avons décidé de voir le problème de calibrage comme un problème d'optimisation, nous allons donc maintenant nous intéresser à différentes approches d'optimisation indépendamment des domaines d'applications, à savoir les systèmes multi-agents et la simulation.

I.3 Optimisation

Nous présentons dans ce chapitre différents algorithmes d'optimisation, qui seront par la suite utilisés dans la partie suivante sur l'exploration de l'espace des paramètres. Nous expliquons d'abord les algorithmes génétiques et le recuit simulé ; ces algorithmes sont utilisés dans le chapitre II.1 page 31. Nous présentons ensuite les algorithmes colonies de fourmis ; ils ont inspiré une méthode du chapitre II.2 page 43. Nous décrivons enfin la programmation génétique et l'apprentissage par renforcement employé dans le chapitre II.3 page 67.

I.3.1 Algorithme génétique

Les algorithmes génétiques [Whitley, 1994, De Jong, 1999, Bäck et al., 1997, Mitchell et Taylor, 1999] sont une famille de modèles de calcul inspirés par la théorie de l'évolution. Ils permettent de résoudre de nombreux problèmes, plus spécialement ceux liés l'optimisation. Ils encodent une solution potentielle à un problème spécifique sur une structure de données linéaire appelée chromosome (par analogie avec les chromosomes des êtres vivants) et y appliquent des opérateurs de recombinaison. L'algorithme commence avec une population de chromosomes (généralement initialisés de manière aléatoire). Puis les chromosomes sont évalués par une fonction de *fitness* caractérisant la qualité du chromosome par rapport au problème posé. Les chromosomes représentant les meilleurs solutions au problème posé ont alors plus de chances de se « reproduire » que les chromosomes représentant les plus mauvaises solutions.

Ce modèle a été proposé par John Holland [Holland, 1975, Holland, 1962] dans les années 60-70. Le schéma d'un algorithme génétique se déroule en quatre phases (figure I.3.1) :

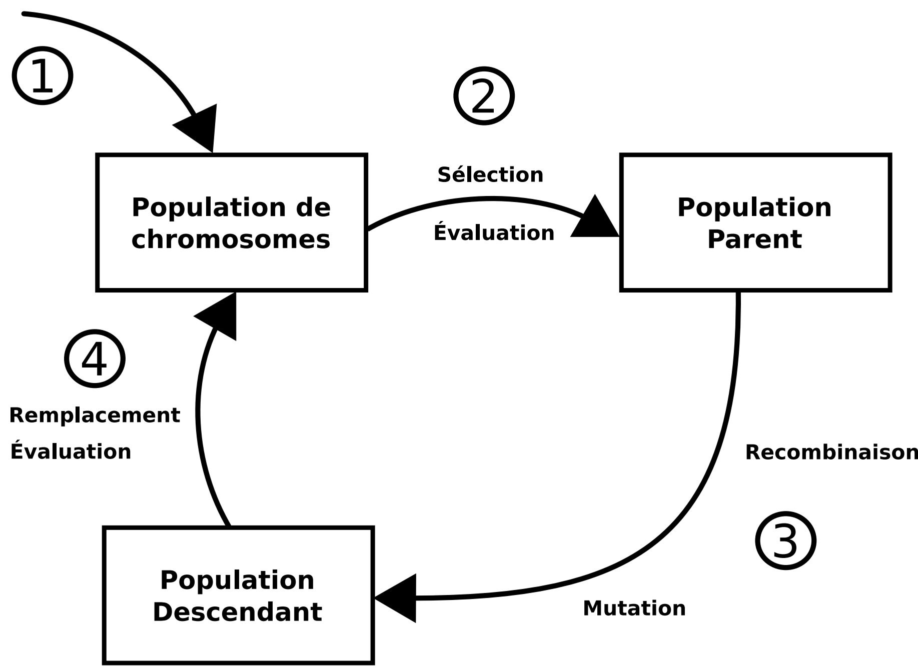


FIG. I.3.1 – *Schéma d'un algorithme génétique*

① l'algorithme génétique commence avec une population courante ;
② une sélection est appliquée sur la population courante pour créer une population intermédiaire ;
③ des opérateurs (recombinaison et mutation) sont appliqués sur la population intermédiaire ;

④ les chromosomes de cette population intermédiaire sont évalués et une nouvelle population est constituée à partir de la population intermédiaire et de la population courante.

Exemple d'application d'un algorithme génétique

Soit la fonction à minimiser

$$f : \begin{array}{ccc} [\![-127,127]\!]^2 & \longmapsto & \mathbb{R} \\ (x,y) & \longrightarrow & x^2 + y^2 \end{array}$$

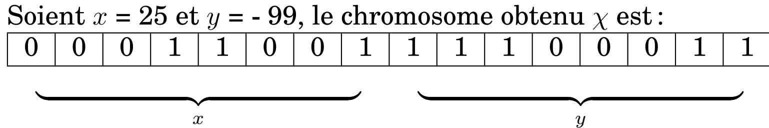
I.3.1.1 Chromosomes

Un chromosome est une structure de données linéaire où est encodée une solution potentielle.

Par exemple, si on encode les entiers sur 8 bits avec un bit de signe, alors une solution potentielle, c'est-à-dire un chromosome, est la séquence formée par l'écriture en binaire de x et y en 8 bits.

Exemple de chromosome

Soient x = 25 et y = - 99, le chromosome obtenu χ est :



Fonction de *fitness*

La fonction de *fitness* est primordiale dans l'algorithme génétique. Elle mesure si un chromosome est bon en fonction des hypothèses :

$$f : \{0,1\}^l \rightarrow \mathbb{R} \text{ avec } l \text{ longueur du chromosome}$$

S'il s'agit par exemple de minimiser une fonction, la fonction de *fitness* serait de façon intuitive la valeur de la fonction par rapport à un jeu de paramètres.

Pour cet exemple, une fonction de *fitness* possible pour ce problème de minimisation est la valeur de la fonction. Donc, ici, la valeur de la fonction de *fitness* pour le chromosome χ est 10426.

Opérateurs

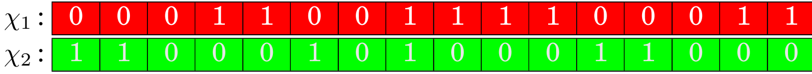
Lors de la reproduction, il est possible d'appliquer des opérateurs : recombinaisons et mutations.

Recombinaisons ou crossing-over

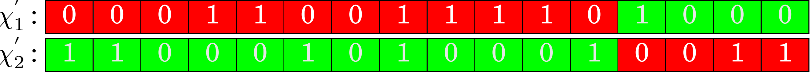
Lors de cette opération, deux chromosomes s'échangent des parties de leurs séquences, ce qui crée alors de nouveaux chromosomes. C'est l'opérateur le plus présent dans les algorithmes génétiques. Intuitivement, cela correspond à prendre dans plusieurs solutions la bonne partie du code génétique afin d'améliorer la solution globale. Il existe plusieurs sortes de recombinaisons, notamment ;

- croisement à un site : cet opérateur choisit un rang au hasard et échange les parties commençantes et finissantes des deux chromosomes ;
- croisement à deux sites : cet opérateur échange une partie choisie aléatoirement des deux chromosomes ;
- croisement uniforme : chaque gène est pris aléatoirement chez l'un des deux parents ;

Par exemple, soient deux chromosomes χ_1 et χ_2,



Par croisement à un site, les deux chromosomes deviennent :



Mutation

Un gène peut, au sein d'un chromosome, être substitué à un autre. Ce mécanisme permet de maintenir une certaine diversité dans la population et ainsi évite une convergence prématurée vers une solution locale.

Par exemple, soit un chromosome χ_1,

χ_1 :

0	0	0	1	1	0	0	1	1	1	1	0	0	0	1	1

Une mutation revient à modifier un gène c'est-à-dire ici un bit :

χ_1'' :

0	0	0	1	1	0	0	1	1	1	1	1	0	0	1	1

Ces opérateurs ne sont pas appliqués de façon aléatoire. Il existe des méthodes permettant d'augmenter la vitesse de convergence vers l'optimum global en évitant l'homogénéisation de la population : par exemple, le principe de la roulette biaisée (les opérateurs sont appliqués suivant une probabilité définie au départ. Les meilleurs chromosomes sont ensuite avantagés c'est-à-dire que ce sont eux qui vont avoir la plus grande possibilité de se voir appliquer un opérateur).

I.3.1.2 Variantes

Une variante des algorithmes génétiques est la stratégie d'évolution [Rechenberg, 1973, Schwefel, 1979, Schwefel et Rudolph, 1995, Schwefel, 1993]. À la différence des algorithmes génétiques classiques, elle utilise directement le codage des chromosomes en réel et non en binaire :

$$f : M \subseteq \mathbb{R}^n \rightarrow \mathbb{R}$$

Il est possible de distinguer plusieurs variantes. Une première est la stratégie (μ,λ) : elle utilise un schéma déterministe de sélection. Dans la phase de sélection, μ parents créent $\lambda > \mu$ descendants par recombinaisons et mutations. Dans la phase de remplacement, les μ meilleurs descendants remplacent leurs parents. Cet algorithme est non élitiste, c'est-à-dire que d'une génération à la suivante, il n'est pas assuré de conserver les meilleurs individus. Une deuxième variante est la stratégie $(\mu+\lambda)$. Comme précédemment, μ parents créent $\lambda > \mu$ descendants par recombinaisons et mutations. Mais dans la phase de remplacement, ce sont les μ meilleurs chromosomes parmi l'union des parents et des descendants qui sont choisis pour former la population de la génération suivante. Cette fois-ci, l'algorithme est élitiste. Une troisième variante est la généralisation des deux premières : la stratégie (μ,κ,λ) où κ indique le nombre de générations maximums que peut vivre un chromosome. Le cas où $\kappa = 1$ correspond à de la stratégie (μ,λ) ; si $\kappa = +\infty$, il s'agit de la stratégie $(\mu + \lambda)$.

I.3.2 Programmation génétique

La programmation génétique ou *Genetic Programming* [Cramer, 1985, Koza, 1992] est une extension des algorithmes génétiques. Au lieu de travailler sur des nombres, cet algorithme travaille sur des programmes. L'idée est de voir un programme comme une structure d'arbre et d'appliquer des opérateurs de mutation et de recombinaison de la même manière que sur les chromosomes d'un algorithme génétique. Les figures I.3.2, I.3.3 et I.3.4 montrent respectivement le codage sous forme d'arbre des opérations arithmétiques infixes, la recombinaison entre deux arbres et la mutation dans un arbre.

Le « genetic programming » peut s'appliquer à de nombreuses applications, comme par exemple pour étudier l'évolution des stratégies de recherche de nourriture d'un lézard des Caraïbes [Koza et al., 1992].

I.3.3 Recuit simulé

Le recuit simulé [Kirkpatrick et al., 1983] est une heuristique pour l'optimisation. Il s'appuie sur l'algorithme de Metropolis [Metropolis et al., 1953]. Ce dernier fait une analogie entre un processus

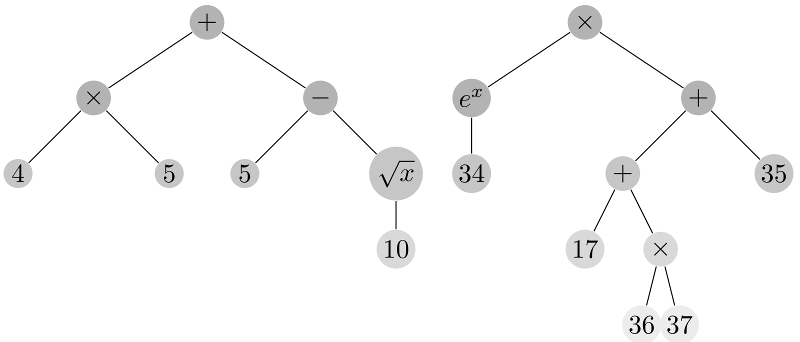
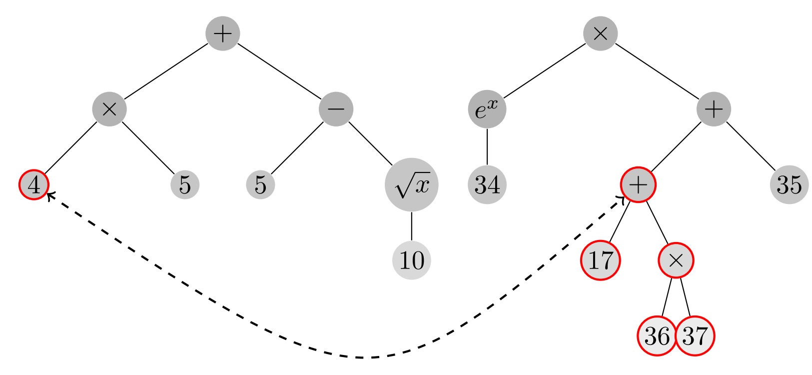


FIG. I.3.2 – *Exemple de « genetic programming »*



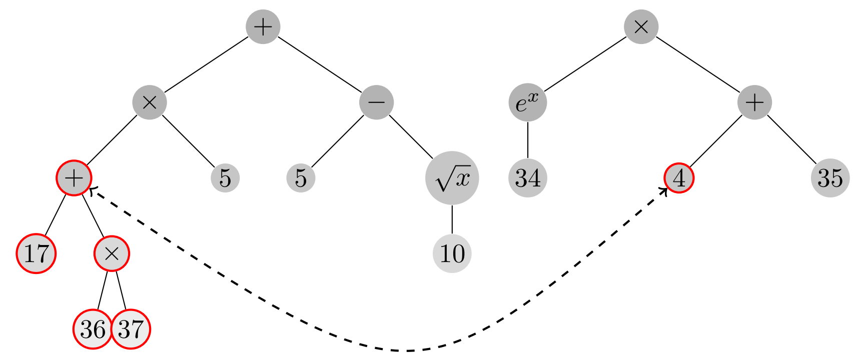
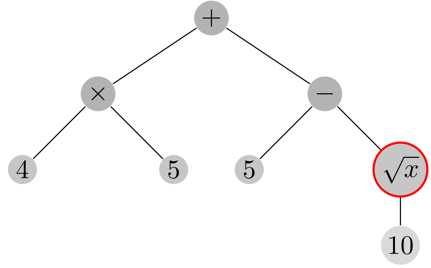


FIG. I.3.3 – *Recombinaison*



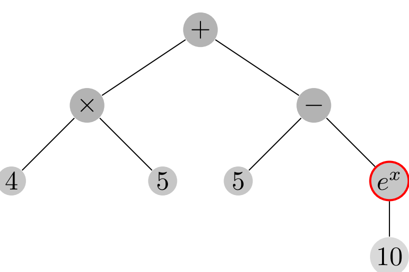


FIG. I.3.4 – *Mutation*

physique et le problème d'optimisation : la fonction à optimiser devient dans le recuit simulé l'énergie E du processus physique. Le processus est aussi caractérisé par une température T.

Initialement, l'algorithme du recuit simulé démarre d'une solution donnée : l'énergie du système est calculée en ce point. Puis, une seconde solution est choisie dans le voisinage de la solution initiale : l'énergie du système est calculée en ce nouveau point. Si la nouvelle énergie est plus basse que l'énergie initiale, c'est-à-dire que la nouvelle solution améliore la fonction à minimiser, alors cette solution est acceptée. Et l'algorithme recommence à partir de ce point. Il est aussi possible d'accepter une solution qui n'améliore pas la fonction à minimiser : cela permet d'éviter les optima locaux, et d'explorer une plus grande partie de l'espace des solutions. Cette acceptation va dépendre de la température du système. Au départ de l'algorithme, la température T est choisie haute ce qui permet à l'algorithme d'accepter des « mauvaises » solutions. Au fur et mesure du déroulement de l'algorithme, la température baisse. Plus elle sera basse, plus la probabilité que l'algorithme accepte une « mauvaise » solution, sera faible. Il existe deux grandes manières de faire varier la température :

- la première est de la baisser de façon continue ;
- la seconde est de la baisser par paliers c'est-à-dire que la température reste constante un certain nombre de générations de l'algorithme, puis elle diminue.

La figure I.3.5 résume le principe du recuit simulé.

Le recuit simulé peut être parallélisable à l'aide de diverses méthodes [Greening, 1990]. L'une d'elles [Ram et al., 1996] est d'exécuter un recuit simulé sur chaque machine et de synchroniser l'ensemble toutes les n générations. La figure I.3.6 en montre le principe.

I.3.4 Colonies de fourmis

Les algorithmes dits de « colonies de fourmis » (« *Ants Colony Algorithms* ») [Blum, 2005, Dorigo et Blum, 2005, Dorigo et al., 2006, Dorigo et al., 1999, Dorigo et of Socha, 2007] peuvent être considérés comme une partie de l'intelligence collective (« *Swarm Intelligence* ») [Bonabeau et al., 1999] c'est-à-dire la création de systèmes multi-agents intelligents en prenant inspiration de comportements

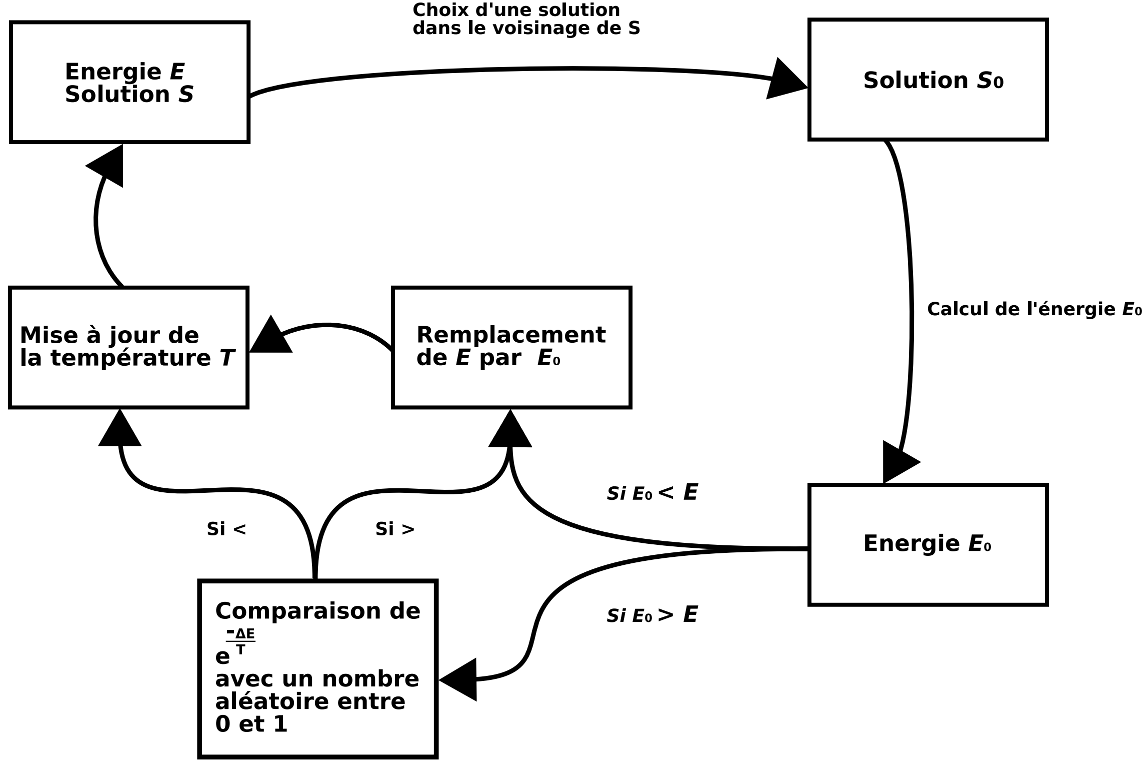


FIG. I.3.5 – *Schéma d'un recuit simulé*

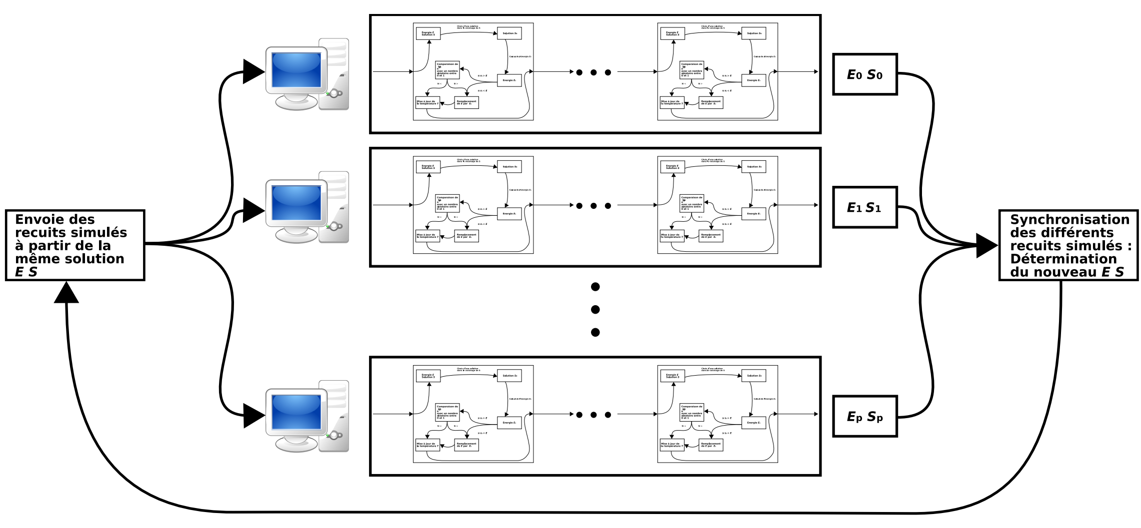
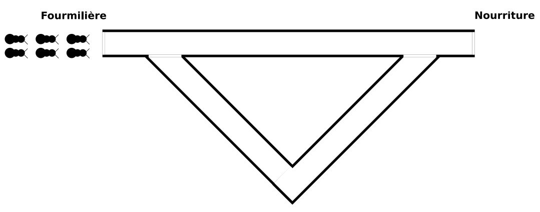


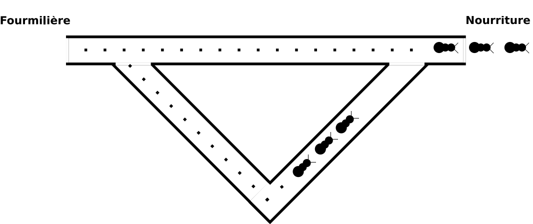
FIG. I.3.6 – *Schéma d'un recuit simulé parallèle*

naturels collectifs intelligents chez les animaux sociaux, plus particulièrement chez les insectes : c'est un exemple de stimergie [Grassé, 1959] où un jeu de mécanismes dynamiques permettant à la structure globale d'émerger comme résultat d'interactions parmi les entités locales.

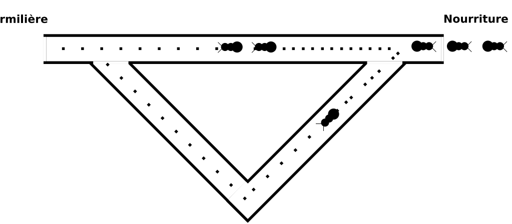
Le principe des algorithmes « colonies de fourmis » est inspiré du dépôt de phéromones par les fourmis (voir l'exemple du fourragement dans la section B.5 page 130 et la sous-section I.1.3.3 page 10). La figure I.3.4 en montre le principe à travers l'expérience du double pont [Deneubourg et al., 1990, Goss et al., 1989].



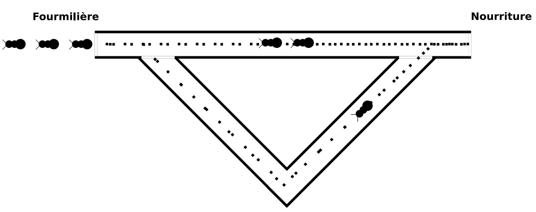
(a) Les fourmis partent de la fourmilière à la recherche de nourriture : deux chemins se présentent à elles. La probabilité de choix d'un chemin est proportionnelle à la quantité de phéromones présente sur ce chemin. La moitié d'entre elles choisit un chemin et le reste choisit l'autre chemin.



(b) Les fourmis ayant choisi le chemin le plus court arrivent et vont choisir le chemin de retour : comme le chemin le plus court possède le plus de phéromones, elles vont le choisir en priorité.



(c) Les trois dernières fourmis arrivent à leur tour et vont choisir le chemin de retour : comme le chemin le plus court possède le plus de phéromones, elles vont le choisir en priorité.



(d) Au final, c'est sur le chemin le court qu'il y a le plus de phéromones.

FIG. I.3.7 – *Expérience du double pont*

Pour illustrer les algorithmes « colonies de fourmis », nous pouvons nous intéresser à un algorithme tel que les « *Ant Systems* » [Dorigo et al., 1991, Dorigo et al., 1996]. Pour illustrer cet algorithme, nous allons utiliser l'exemple du voyageur de commerce : c'est un graphe complet non-orienté où les nœuds et les arcs représentent respectivement les villes et les distances entre les villes. Un nœud est choisi comme nœud de départ pour les fourmis. Puis les fourmis explorent le graphe. La probabilité pour une fourmi k d'aller de la ville i à la ville j est donnée par:

$$p_{ij}^k = \begin{cases} \frac{(\tau_{ij})^\alpha \times (\eta_{ij})^\beta}{\sum_{l \in J_i^k} (\tau_{ij})^\alpha \times (\eta_{ij})^\beta} & \text{if } j \in J_i^k \\ 0 & \text{if } j \notin J_i^k \end{cases}$$

où τ_{ij} est la quantité de phéromones, η_{ij} est la visibilité ($= \frac{1}{d_{ij}}$ où d_{ij} est la distance entre les villes i et j) (cela représente une heuristique), et J_i^k est une mémoire des villes déjà visitées.

Après la fin d'un tour, chaque fourmi k dépose une quantité $\Delta\tau_{ij}^k$ de phéromones sur l'arc (i,j) de la manière suivante :

$$\Delta\tau_{ij}^k = \begin{cases} \frac{Q}{L^k} & \text{if } (i,j) \in T^k \\ 0 & \text{if } (i,j) \notin T^k \end{cases}$$

où T^k est le tour fait par la fourmi k à l'itération t, L^k est sa longueur et Q est un paramètre.

La quantité de phéromones est alors mise à jour afin de simuler l'évaporation des phéromones, en utilisant la formule :

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij} + \Delta\tau_{ij}$$

où $\Delta\tau_{ij} = \sum_{k=1}^{m} \Delta\tau_{ij}^k$ et m est le nombre de fourmis.

La méthode peut se généraliser avec les méta-heuristiques d'optimisation par colonies de fourmis [Dorigo et Di Caro, 1999]. Le schéma général de la méta-heuristique d'optimisation par colonies de fourmis est :

---

tant que conditions de terminaison non atteintes **faire**
  OrdonnanceActivités
    ConstructionSolutionParFourmis()
    MiseAJourPheromones()
    ActionsDémons()
  fin OrdonnanceActivités
fin tant que

De plus l'espace des paramètres peut être un espace continu [Bilchev et Parmee, 1995]. Une approche intéressante dans ce cas est l'algorithme proposé par Socha et Dorigo [Socha et Dorigo, 2006].

Les algorithmes « colonies de fourmis » ont été utilisés dans de nombreux domaines [Blum, 2005, Dorigo et Stützle, 2002], allant du problème du voyageur de commerce jusqu'à à des problèmes de planification en passant par des problèmes en bio-informatique ou en musique.

I.3.5 Apprentissage par renforcement

L'apprentissage par renforcement est une technique d'apprentissage couramment utilisée. De nombreux papiers [Kaelbling et al., 1996, Gonçalo, 2005, Keerthi et Ravindran, 1995] proposent des états de l'art sur cette technique. Nous allons la présenter de manière succinte dans cette sous-section.

L'apprentissage par renforcement consiste à apprendre à partir d'expérimentations ce qu'il convient de faire dans différentes situations, de façon à optimiser une récompense au cours du temps. Un exemple très classique pour présenter cet apprentissage est de considérer un agent qui doit prendre des décisions en fonction de son état courant. Suivant son choix dans ses décisions, il reçoit une récompense algébrique de la part de l'environnement. L'agent cherche, au travers d'expériences itérées, un comportement décisionnel (appelé politique) optimal, c'est-à-dire il essaie de maximiser la somme des récompenses au cours du temps.

Plus formellement, l'apprentissage par renforcement peut se définir par l'intermédiaire d'un processus de décision de Markov. Un quadruplet (S,A,T,R) est définit où :

- A est un ensemble d'actions ;
- S est un ensemble d'états ;
- $T : S \times A \times S \rightarrow [0,1[$ est une fonction de transition définie comme une distribution de probabilité sur les états. Ici, on a $T(s,a,s') = Pr(s_{t+1} = s'|s_t = s,a_t = a)$ où s_{t+1} représente l'état du processus au temps $t+1$, s_t représente l'état du processus au temps t, a_t est l'action prise après l'observation de l'état s_t.
- R: $S \times A \times S \rightarrow \mathbb{R}$ est la fonction de récompense représentant la valeur attendue de la prochaine récompense étant donné l'état courant s et l'action a et de prochain état s' : $R(s,a,s') = E(r_{t+1}|s_t = s,a_t = a,s_{t+1} = s'$ Dans ce contexte, r_{t+1} représente le paiement immédiat de l'environnement à l'agent au temps $t + 1$.

À chaque pas d'interaction i, l'agent perçoit l'environnement à l'état courant s ($s \in S$). Puis, l'agent choisit une action $a \in A$. L'action change l'état de l'environnement : l'état courant devient $s' \in S$. La valeur de la transition d'état est communiquée à l'agent à travers un signal de renforcement r calculé par la fonction de récompense $R : r = R(s,a,s')$. L'agent choisit ses actions pour essayer de maximiser la somme des valeurs du signal de renforcement. Il a plusieurs façons d'optimiser cette somme. À chaque étape, l'agent choisit une action $a \in A$ suivant sa perception de l'environnement à l'état $s \in S$. On appelle politique de Markov déterministe π une fonction $S \mapsto A$ qui associe à chaque perception de l'environnement à l'état s, une action $\pi(s)$ à effectuer. Il existe différents algorithmes pour trouver une politique optimale : par exemple *TD(0)* [Sutton, 1988] ,ou ***Q**-Learning*[Watkins et Dayan, 1992].

Ces algorithmes d'apprentissage par renforcement ont notamment été utilisés dans le domaine des systèmes multi-agents [Panait et Luke, 2005, Shoham et al., 2003].

I.3.6 Discussion

Comme nous l'avons expliqué dans ce chapitre, il existe un grand nombre d'algorithmes d'optimisation par rapport à notre problème. Ce chapitre expose quelques algorithmes comme les algorithmes génétiques ou les colonies de fourmis mais il existe un grand nombre d'autres méta-heuristiques pour l'optimisation, par exemple la recherche tabou [Glover, 1986]. Et, pour augmenter la variété d'algorithmes, une tendance actuelle est d'hybrider les différentes méta-heuristiques : L'article de Sinha et de Goldberg [Sinha et Goldberg, 2003] donnent ainsi quelques possibilités d'hybridation avec les algorithmes génétiques.

Parmi toutes ces méthodes d'optimisation, l'utilisateur se pose très rapidement une question : quelle est la meilleure méta-heuristique ? Répondre à cette interrogation est difficile : dans la littérature scientifique, les réponses sont souvent contradictoires. Les algorithmes génétiques sont présentés comme étant plus efficaces dans un certain nombre de références [Chiaberge et al., 1994, Manikas et Cain, 1996, Vanier et Bower, 1999, Ghosh et al., 2005, Krishnan et Purdy, 2006]. D'autres références mettent en avant plutôt le recuit simulé [Lahtinen et al., 1996, Franconi et Jennison, 1997, Arostegui et al., 2006]. Ceci rejoint le théorème « *No Free Lunch* » [Wolpert et Macready, 1997, Wolpert et Macready, 1995, Ho et Pepyne, 2002] : aucune méta-heuristique n'est la meilleure quel que soit le problème.

Nous allons maintenant utiliser une partie de ces algorithmes pour explorer l'espace des paramètres.

Seconde partie

Exploration de l'espace des paramètres : les différentes méthodes

II.1 Algorithmes génétiques ou les méthodes classiques d'optimisation

Nous présentons dans ce chapitre une méthode de calibrage basé sur l'utilisation des algorithmes génétiques. Le but est d'exhiber au moins un jeu de paramètres pour un modèle multi-agent selon un objectif défini (voir la section I.1.4 pour une liste de différents objectifs possibles). Comme nous l'avons mentionné dans la section I.2.2, une possibilité pour calibrer un modèle à base d'agents est de parcourir l'espace des paramètres, mais ce parcours ne peut pas être exhaustif. Une solution est alors de considérer ce problème comme un problème d'optimisation, et d'appliquer des heuristiques. Ce problème peut être considéré comme un problème d'optimisation et être résolu en appliquant des heuristiques. Notre choix de l'heuristique s'est porté sur les algorithmes génétiques (voir la sous-section I.3.1 pour une description des algorithmes génétiques) [Rogers et von Tessin, 2004, Narzisi et al., 2006]. Pour résumer la méthode, le modèle sera vu comme un modèle en boîte noire : les entrées correspondront aux valeurs des paramètres ; la sortie correspondra au résultat de la fonction de *fitness*. Le calcul de la sortie à partir des entrées nécessite une simulation du modèle paramétré par l'entrée. La figure II.1.1 montre le principe.

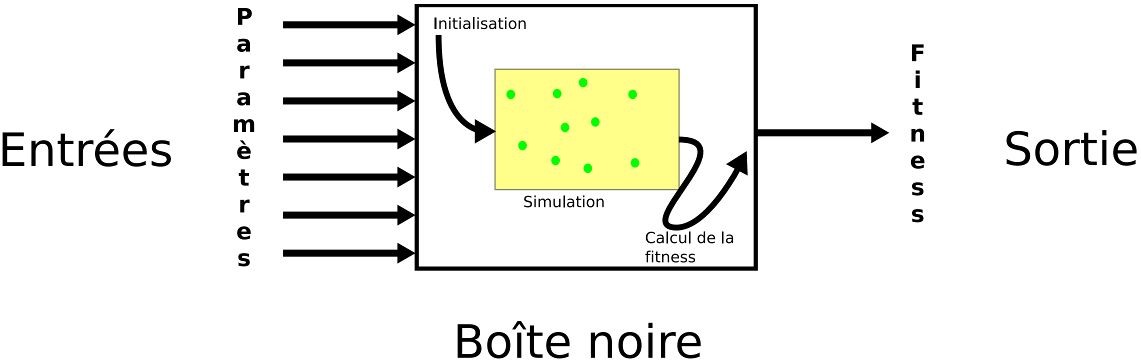


FIG. II.1.1 – *Vision en boîte noire de la simulation multi-agent*

II.1.1 Méthode

L'idée majeure de cette méthode est d'appliquer un algorithme génétique pour optimiser le paramétrage de la simulation multi-agent vue comme une boîte noire. Nous avons développé différentes versions. Toutefois seules les deux plus représentatives seront présentées ici : l'application directe d'un algorithme génétique et une variante plus complexe prenant en compte stochasticité.

II.1.1.1 Version naïve

Dans cette première version, nous appliquons directement un algorithme génétique (*AG*). Un modèle avec un jeu de paramètres (c'est-à-dire un modèle paramétré) correspond à un chromosome de l'algorithme génétique. Un paramètre de ce modèle coïncide avec un gène (ou plusieurs gènes suivant le codage des nombres et de l'algorithme génétique)[1].

La figure II.1.2 page suivante résume le principe de cette approche naïve :

① D'abord, une population de modèles paramétrés est créée de façon aléatoire c'est-à-dire que la méthode crée un ensemble de jeux de paramètres, dont les valeurs sont choisies de façon

1. Dans l'implantation informatique de cette méthode, un paramètre correspond à un gène.

aléatoire, pour un même modèle. Puis la *fitness* de chaque modèle paramétré est calculée ce qui nécessite une simulation (voire plusieurs simulations pour tenir compte de la stochasticité).

② À partir de cette population est créée une population « parents ».

③ Sur cette population « parents » sont appliqués des opérateurs : recombinaison et mutation. Une nouvelle population est créée : population « descendants ». Puis est calculée la *fitness* de chaque modèle de la population « descendants » : une simulation (voire plusieurs simulations pour tenir compte de la stochasticité) est exécutée.

④ À partir des populations « parents » et « descendants » est créée une nouvelle population. L'algorithme recommence avec cette nouvelle population.

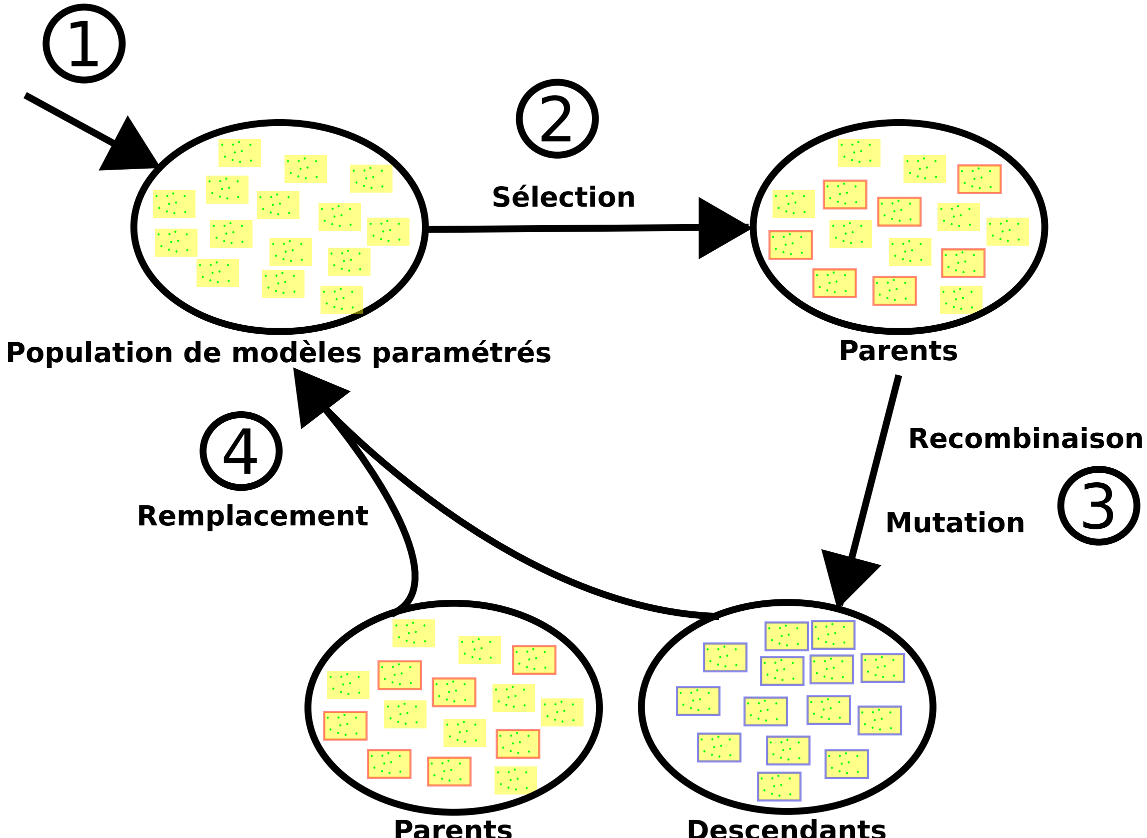


FIG. II.1.2 – *Schéma de la méthode algorithme génétique*

La partie la plus coûteuse en temps est le calcul de la *fitness* de chaque modèle paramétré. En effet, cela nécessite au moins une simulation, ce qui requiert du temps. Mais, toutes ces simulations peuvent se faire en parallèle pour chaque modèle paramétré. À partir de cette version initiale de la méthode, il dérive une version où, lors de l'évaluation de *fitness*, les simulations sont distribuées sur plusieurs ordinateurs pour évaluer en parallèle plusieurs modèles [Cantu-Paz et Goldberg, 2000, Back et al., 1995]. La figure II.1.3 page ci-contre résume le principe.

Cependant, comme nous l'avons montré dans la section I.2.2 page 16, les simulations multi-agent sont généralement stochastiques : deux simulations d'un même modèle avec le même jeu de paramètres ne donnent pas nécessairement les mêmes résultats. Il faut donc multiplier le nombre de simulations, ce qui augmente encore le temps de calcul. Nous avons donc développé une version plus complexe de la méthode visant à limiter le nombre de simulations en tenant compte de la stochasticité.

II.1.1.2 Quelques précisions sur la méthode

Cette sous-section détaille quelques points particuliers liées aux algorithmes génétiques.

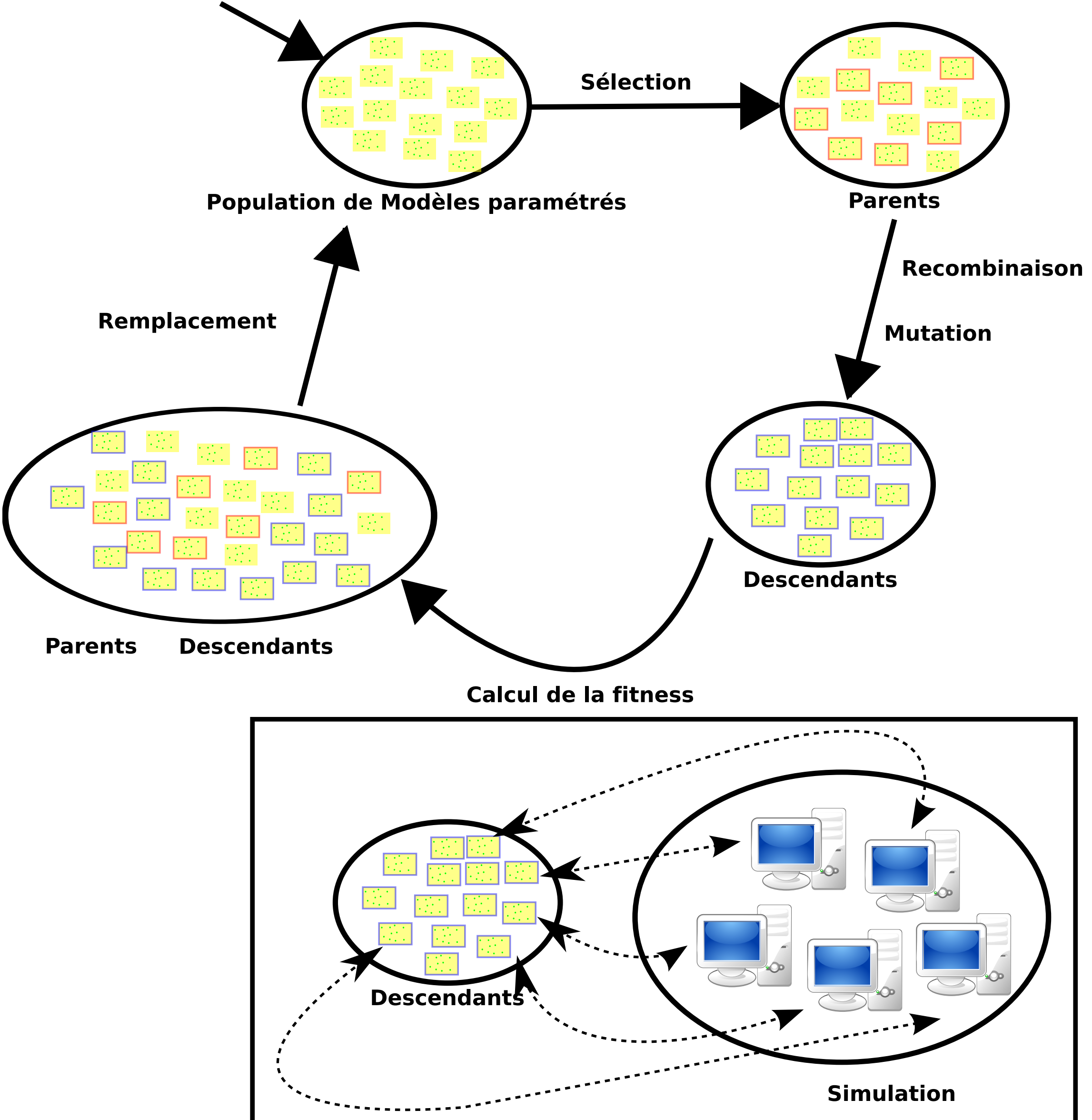


FIG. II.1.3 – *Schéma de la méthode algorithme génétique avec les simulations distribuées*

Choix de l'algorithme génétique

Un point important de la méthode est le choix de l'algorithme génétique. Il existe un très grand nombre d'algorithmes génétiques avec de très nombreuses variantes (voir la sous-section I.3.1 page 19 pour une description des algorithmes génétiques). Le but de ce manuscrit n'est pas de détailler les caractéristiques de toutes ces variantes mais de préciser les choix effectués. Nous avons choisi plus particulièrement d'implanter un modèle élitiste d'algorithme génétique afin d'avoir une amélioration continue de l'optimisation (les meilleurs chromosomes sont toujours conservés). Ce choix autorise ainsi l'arrêt de l'optimisation à n'importe quel moment, en étant assuré de conserver le meilleur résultat obtenu jusque là. Plus précisément, l'algorithme génétique, utilisé dans ce manuscrit, est un algorithme s'inspirant de l'algorithme proposé par Whitley [Whitley et Kauth, 1988, Whitley, 1989]. Il dérive de la stratégie $(\mu + \lambda)$: à chaque génération, les n chromosomes parents produisent n chromosomes enfants. Les $t\%$ des pires parents sont ensuite remplacés par les $t\%$ des meilleurs enfants. Ce taux de remplacement t est généralement fixé à 25%, valeur que nous avons également retenue. Par conséquent, le schéma de l'algorithme est :

1. Initialisation : génération aléatoire d'une population ;
2. Évaluation de cette population ;
3. Tri de cette population ;
4. Reproduction des chromosomes : les meilleurs chromosomes ont une grande probabilité de se reproduire ;
5. Évaluation des chromosomes fils ;
6. Tri des fils ;
7. Remplacement des 25 % pires parents par les 25 % meilleurs enfants ;
8. Retour à l'étape 4.

Paramètres de l'algorithme génétique

Il faut ensuite choisir les paramètres de l'algorithme génétique. Notre choix s'est porté sur un paramétrage standard de l'algorithme. Le principal paramètre de l'algorithme génétique où le choix est important est le nombre de générations de l'algorithme à effectuer. Ainsi, une question à se poser est de savoir quand arrêter la méthode ? Une première idée est de décider un nombre fixe de générations. Tout au long de ce manuscrit, c'est ce choix qui a été fait : étant donné un modèle multi-agent, l'objectif est de rechercher un paramétrage adéquat en un nombre fixé de simulations. Nous avons choisi $10\ 000$ simulations : ce choix peut paraître énorme dans le cadre de simulations nécessitant un temps d'exécution long. Toutefois cela permet d'étudier sur un grand nombre de simulations le comportement présenté dans ce manuscrit. Un des objectifs des méthodes de calibrage proposées dans ce manuscrit est d'arriver à trouver le meilleur calibrage possible en un minimum de simulations. Le modélisateur choisit donc le nombre de générations ou de simulations (par exemple, pour que le temps de calcul corresponde à une journée de calcul). Un deuxième critère pour arrêter l'algorithme est lié à la convergence de la *fitness* individuelle du meilleur modèle paramétré (ou des meilleurs) au fil des générations. Quand cette *fitness* converge, alors il est possible de décider d'arrêter la méthode algorithme génétique.

II.1.1.3 Prise en compte de la stochasticité

Diverses approches ont été proposées pour prendre en charge la stochasticité dans les algorithmes génétiques. Une première solution est d'augmenter la taille de la population [Goldberg et al., 1992] Une deuxième solution est d'évaluer plusieurs fois la *fitness* de chaque chromosome [Beyer, 2000, Cantú-Paz, 2004]. Chacune de ces deux approches ont pour conséquence une hausse du nombre d'évaluations de la *fitness*. Dans le cadre des modèles à base d'agents, cela signifie augmenter le nombre de simulations, et donc prolonger le temps d'exécution de l'algorithme. Une autre solution est d'approximer le résultat du calcul de la *fitness* à partir d'un modèle mathématiques [Jin, 2005] ou à partir du voisinage [Branke et Schmidt, 2005]. En partant de cette idée, une approche intéressante [Jin et al., 2000] est que l'évaluation de la *fitness* soit effectuée à partir du modèle mathématique dans tous les cas, et que toutes les λ générations, l'évaluation de la *fitness* soit effectuée à partir de la fonction de *fitness* réelle (c'est-à-dire, pour les modèles à base d'agents, avec des simulations) durant

η générations. Mais comme nous l'avons noté dans la section I.2 page 13, la création d'un méta-modèle pour un modèle multi-agent est délicate et n'est généralement pas une solution satisfaisante.

Cependant la *fitness* obtenue avec une unique simulation peut être considérer comme une approximation de la *fitness* réelle, qui nécessitera quant-à elle d'effectuer plusieurs simulations pour tenir compte de la stochasticité. Une solution est alors d'estimer la *fitness* avec une seule simulation durant n générations. Après ces n générations, selon la stochasticité de la simulation et du souhait de qualité de la *fitness*, cette dernière sera estimée avec x simulations. La figure II.1.4 résume ce principe. En pratique, l'algorithme génétique implanté est un algorithme génétique élitiste (voir la sous-section I.3.1 page 19). À chaque génération de l'algorithme, seulement 25 % de la population est remplacée. Le choix de n et de x dépend de la stochasticité du simulateur. Idéalement il faudrait étudier cette stochasticité, sachant qu'elle peut varier en fonction des valeurs de paramètres. Le problème est que cette analyse nécessite de réaliser de nombreuses simulations, donc est elle-même très consommatrice de temps. Dans l'exemple suivant, nous avons choisi que la *fitness* de chaque modèle soit estimée toutes les dix générations avec cinq simulations.

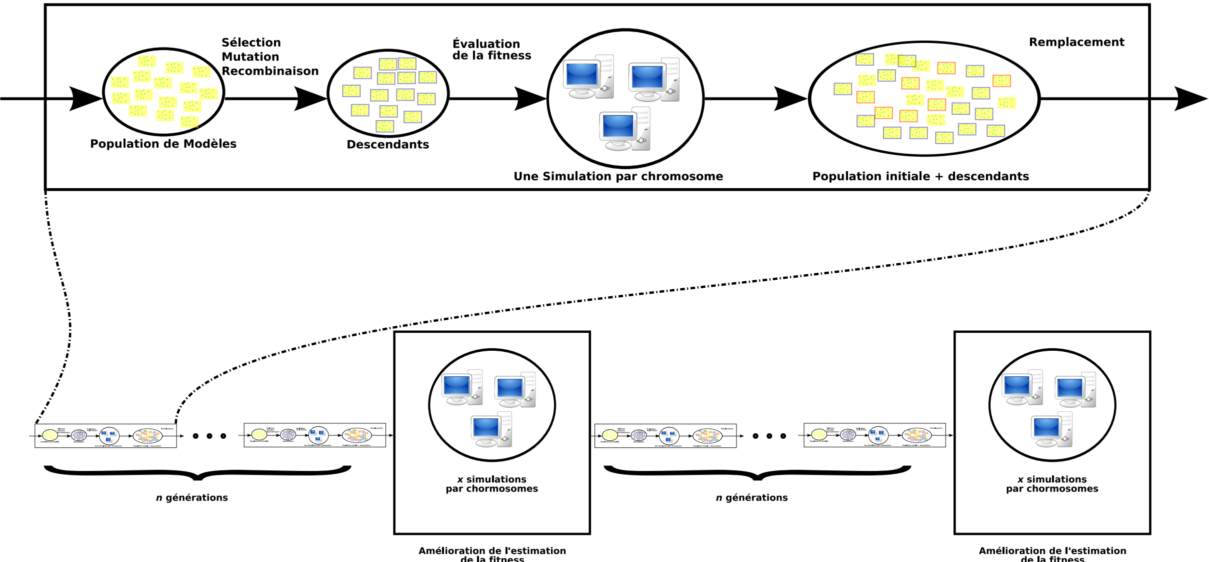


FIG. II.1.4 – *Principe de la méthode tenant compte de la stochasticité*

II.1.2 Résultats

Cette section va détailler quelques résultats sur un exemple simple.

II.1.2.1 Description du modèle

L'exemple est fondé sur le fourragement par une colonie de fourmis. Il utilise une version modifiée du modèle « *Ants* » de la plate-forme de simulation `NetLogo` : pour plus de détails sur le modèle original et le modèle modifié, il faut se reporter à la section B.5 page 130. Les paramètres de cet exemple à optimiser sont les suivants :

- `speed`
- `patch_ahead`
- `angle_vision`
- `drop_size`

Tous ces paramètres sont des paramètres propres aux agents. Tous les autres sont fixés avec des valeurs standards :

- le taux de diffusion des phéromones dans l'environnement `diffusion-rate` est fixé à 50 ;
- le taux d'évaporation des phéromones de l'environnement `evaporation-rate` est égal à 10 ;

– le nombre de fourmis dans l'environnement est de 25.

Le modèle est simulé durant 500 pas de simulation. L'objectif de ce calibrage est d'optimiser la fonction de récupération de nourriture par les fourmis : la fonction de *fitness* est la quantité de nourriture ramenée à la fourmilière au bout des 500 pas de simulation, que l'on va chercher à maximiser.

II.1.2.2 Résultats

La figure II.1.5 page suivante représente le résultat de l'évolution de la *fitness* des meilleurs modèles paramétrés en fonction du nombre de simulations. Cette courbe est construite en calculant, à chaque génération de l'algorithme, une *fitness* moyenne pour les « meilleurs » modèles (définis par rapport à la *fitness* moyenne et à l'écart-type de cette *fitness*). Nous aurions pu choisir plus simplement de ne retenir que la *fitness* du meilleur modèle, mais du fait de la stochasticité, cette *fitness* peut être surévaluée par rapport aux autres modèles. Plus préoccupant, du point de vue du paramétrage, si l'algorithme n'a pas convergé, les paramétrages des différents modèles peuvent être très différents. Au contraire, lorsque l'algorithme a convergé, il n'y a pas de différence notable entre le meilleur modèle et les n meilleurs modèles, du point de vue du paramétrage. Le paramétrage solution est ainsi calculé comme la moyenne des paramétrages des n meilleurs modèles. La courbe de la figure est très instable. Cette instabilité est due à la stochasticité de la simulation : durant les n générations de l'algorithme, la *fitness* est évaluée avec une seule simulation, et donc, la valeur de la *fitness* des meilleurs modèles paramétrés est surévaluée. Après cette phase, nous calculons de façon plus précise la *fitness* des modèles : la valeur de la *fitness* des meilleurs modèles paramétrés n'est plus surévaluée. Cette phase correspond au rond sur la courbe. La *fitness* des meilleurs modèles paramétrés a une convergence assez rapide. La table II.1.1 montre le paramétrage solution. Avec ce jeu de paramètres, l'agent fourmi se déplace avec une grande vitesse : cependant cette vitesse n'est pas la vitesse maximale. En effet, si l'agent allait à la vitesse maximale (c'est-à-dire 20), la distance parcourue en un pas de simulation serait plus grande que le diamètre des sources de nourriture, et, donc il pourrait, en se dirigeant vers une source de nourriture, la dépasser. Il a un angle de vision d'environ 200°: l'agent ne regarde que devant lui. Il a une perception longue distance maximale. Pour résumer, l'agent se déplace relativement rapidement, perçoit seulement devant lui et le plus loin possible, ce qui apparaît comme un comportement logique. La figure II.1.6 page suivante montre l'évolution de la valeur du paramètre `speed` des meilleurs modèles paramétrés au cours de l'algorithme génétique. Après un début un peu oscillant , il y a une convergence très franche vers une valeur. La figure II.1.7 page 38 montre la même évolution, mais sur un paramètre artificiel qui n'a aucune influence sur la simulation. Ce paramètre varie entre 0 et 5. Il a été ajouté pour tester la méthode. À la différence du précédent, les valeurs de ce paramètre ne convergent pas ; de plus, l'évolution des valeurs est très instable d'une génération de l'algorithme génétique à une autre.

nom du paramètre	valeur
`speed`	8,667
`patch_ahead`	9,525
`angle_vision`	203,687
`drop_size`	120,317

TAB. II.1.1 – *Paramétrage solution obtenu avec la méthode algorithme génétique*

II.1.3 Choix de la *fitness*

Un aspect très important de cette méthode est le choix de la *fitness*, qui comme nous l'avons vu dans la section I.1.4 page 10, dépend des objectifs du modélisateur. Ce choix doit en plus tenir compte de la dynamique du modèle : par exemple, si le modélisateur s'intéresse aux files de fourmis lors du fourragement, s'il évalue trop tôt ou trop tard, il n'y a aucune file, comme nous l'avons déjà indiqué dans la section I.2.2 page 16.

Les résultats dépendent aussi très fortement du choix de la *fitness*. Nous pouvons percevoir ce lien à l'aide d'un exemple avec trois choix de *fitness* différents. Pour cet exemple, nous utilisons le modèle

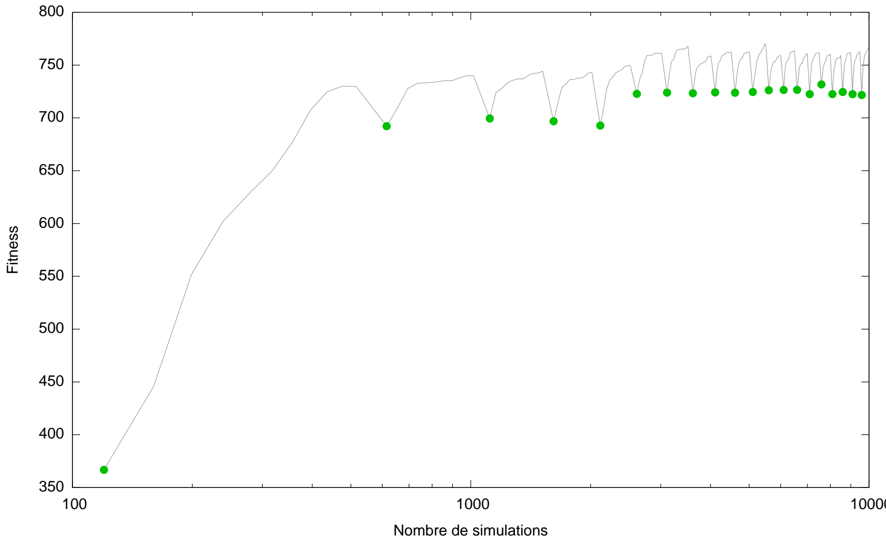


FIG. II.1.5 – *Résultat de la méthode algorithme génétique sur le modèle de fourragement par colonie de fourmis : l'évolution de la fitness des meilleurs modèles paramétrés en fonction du nombre de simulations*

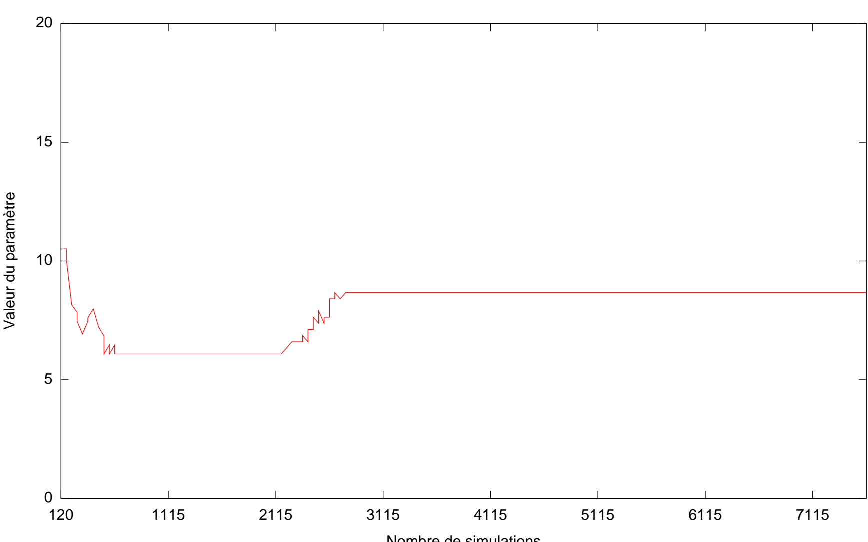


FIG. II.1.6 – *Évolution de la valeur du paramètre* `speed` *des meilleurs modèles paramétrés au cours de l'algorithme génétique*

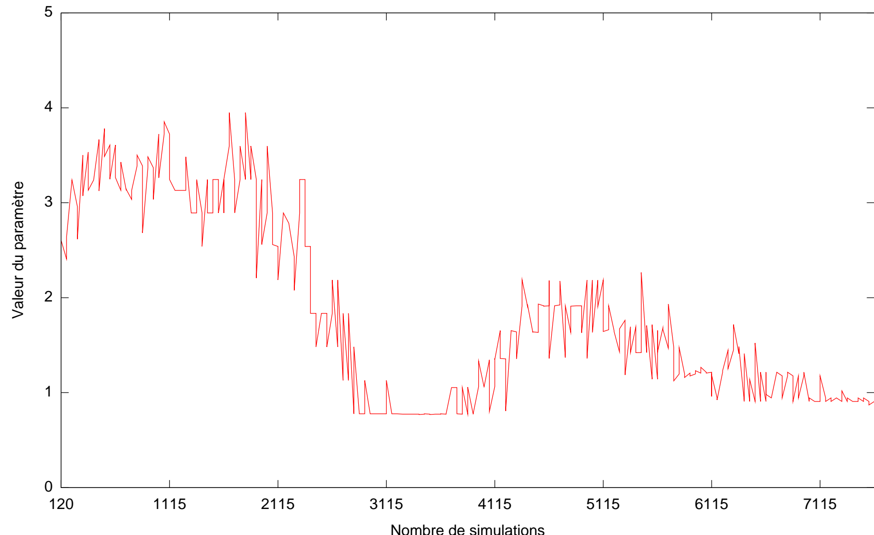


FIG. II.1.7 – *Évolution de la valeur d'un paramètre artificiel des meilleurs modèles paramétrés au cours de l'algorithme génétique*

« *Ant* ». Cependant, pour cet exemple, à la différence du précédent résultat et des résultats suivants, nous allons calibrer les deux paramètres globaux :

– `diffusion-rate`,
– `evaporation-rate`.

fitness 1

Dans ce premier cas, la *fitness* est la quantité de nourriture ramenée à la fourmilière entre le $100^{\text{ième}}$ et le $200^{\text{ième}}$ pas de simulation.

La table II.1.2 page suivante indique le paramétrage solution. À la fin des 100 premiers pas de simulation, les files de fourmis sont formées. Durant les 100 pas de simulation suivant, les fourmis exploitent les sources de nourriture en utilisant le chemin de phéromones ainsi formé. Sur les meilleurs modèles, les trois sources de nourriture sont exploitées en même temps. Cependant, sur ces modèles, le taux d'évaporation de phéromones est plutôt bas. Cela a pour conséquence qu'à l'épuisement d'une source de nourriture, les chemins de phéromones persistent dans l'environnement, et donc les fourmis continuent à vouloir exploiter cette source de nourriture.

fitness 2

Dans ce deuxième cas, la *fitness* est le temps pour ramener toute le nourriture à la fourmilière.

La table II.1.2 page ci-contre indique le paramétrage solution. Comme précédemment, les meilleurs modèles sont les modèles où les trois sources de nourriture sont exploitées en même temps. Cependant, le taux d'évaporation des phéromones dans cet exemple est plus grand que dans le précédent exemple. Cela permet d'obtenir un comportement beaucoup plus dynamique des fourmis : quand une source de nourriture s'épuise, rapidement les fourmis arrêtent de l'exploiter et cherchent d'autres sources de nourriture.

fitness 3

Dans ce dernier cas, nous nous intéressons à une *fitness* plus qualitative. L'objectif du modélisateur est d'obtenir des files de fourmis. Un choix de *fitness* est le nombre de lignes de fourmis. Plus quantitativement, nous déterminons les chemins continus de phéromones entre la fourmilière et les sources de nourriture. Ce nombre de chemins est mesuré toutes les 10 pas de simulation, et la *fitness* est la somme de ces valeurs durant 400 pas de simulations.

La table II.1.2 page suivante indique le paramétrage solution. Les meilleurs modèles exploitent encore les trois sources de nourriture en même temps. À la différence des deux précédents exemples, le taux d'évaporation et le taux de diffusion sont très bas. Cette faiblesse permet de concentrer les

phéromones sur un chemin étroit sans recouvrir tout l'environnement, mais réduit aussi la flexibilité du comportement des fourmis.

Paramètre	*fitness* 1	*fitness* 2	*fitness* 3
Taux d'évaporation	8,1	21,68	9,32
Taux de diffusion	88,6	2,83	1,42

TAB. II.1.2 – *Différents paramétrages solutions pour différentes fitness*

Sur ce petit exemple, nous avons montré que le choix de la *fitness* peut induire des résultats très différents, et donc des comportements des agents très différents. Ainsi le choix de la fonction de la *fitness* doit être effectué de manière attentive. Cette fonction de *fitness* doit ainsi être choisie avec beaucoup de soin.

II.1.4 Par rapport aux autres méthodes d'optimisation

Comme nous l'avons évoqué dans le chapitre I.3 page 19, il existe un grand nombre d'heuristiques d'optimisation, autres que les algorithmes génétiques, qui seraient utilisables pour explorer l'espace des paramètres. Par exemple, il est possible d'utiliser la méthode du recuit simulé décrite à la sous-section I.3.3 page 21. Comme pour l'algorithme génétique, c'est un recuit simulé parallèle que nous avons implanté, similaire à celui de la figure I.3.6 page 24. Pour gérer la stochasticité, nous avons utilisé la même technique que celle développée avec les algorithmes génétiques, c'est-à-dire que, dans la phase d'exploration en parallèle, une seule simulation est réalisée pour le calcul de la *fitness*, et dans la phase de synchronisation, plusieurs simulations sont réalisées pour avoir un calcul précis de la *fitness*. Il est important de calculer une *fitness* relativement précise lors de la phase de synchronisation, afin de choisir le meilleur point de l'espace (c'est-à-dire le meilleur jeu de paramètres) pour poursuivre l'exploration. La figure II.1.8 page suivante résume la méthode. La figure II.1.9 page suivante montre l'évolution de la *fitness* en fonction du nombre de simulations pour les méthodes utilisant le recuit simulé et les algorithmes génétiques. La table II.1.3 montre le paramétrage solution obtenu. Comme pour l'algorithme génétique, il y a convergence de l'évolution de la *fitness*. Le paramétrage solution présente les mêmes caractéristiques : des agents se déplaçant vite et regardant loin devant eux pour la recherche de phéromones. Au final, le résultat de l'évolution est moins bon qu'avec un algorithme génétique. Cependant il serait possible d'obtenir des résultats comparables aux algorithmes génétiques avec le recuit simulé en essayant de chercher un meilleur paramétrage de l'algorithme lui-même (température initiale, méthode de mise à jour de la température, nombre de recuits simulés en parallèle...) et/ou en changeant de recuit simulé.

nom du paramètre	valeur
`speed`	7.439
`patch_ahead`	10.0
`angle_vision`	196.370
`drop_size`	153.208

TAB. II.1.3 – *Paramétrage solution obtenu avec la méthode recuit simulé*

Nous pouvons aussi nous intéresser à la stabilité des résultats à différentes exécutions des algorithmes d'optimisation. La figure II.1.10 page 41 représente les évolutions de la *fitness* à différentes exécutions de la méthode : pour chaque méthode, il y a trois courbes : les valeurs moyennes, maximales et minimales des *fitness* lors des 26 exécutions. Ce test a été effectué sur des fonctions mathématiques prédéfinies : l'annexe E page 143 explique le principe de fonctionnement de ces tests et, plus particulièrement, la section E.3 page 147 détaille la fonction utilisée. Nous créons une « pseudo-simulation » où le comportement d'un agent est le résultat d'une fonction mathématique. L'intérêt est de pouvoir réaliser un grand nombre de « pseudo-simulations » en peu de temps. Sur les tests réalisés, les algorithmes génétiques semblent beaucoup plus stables que le recuit simulé. Cependant, il serait possible de proposer un recuit simulé qui aurait une stabilité comparable à celle de l'algorithme génétique.

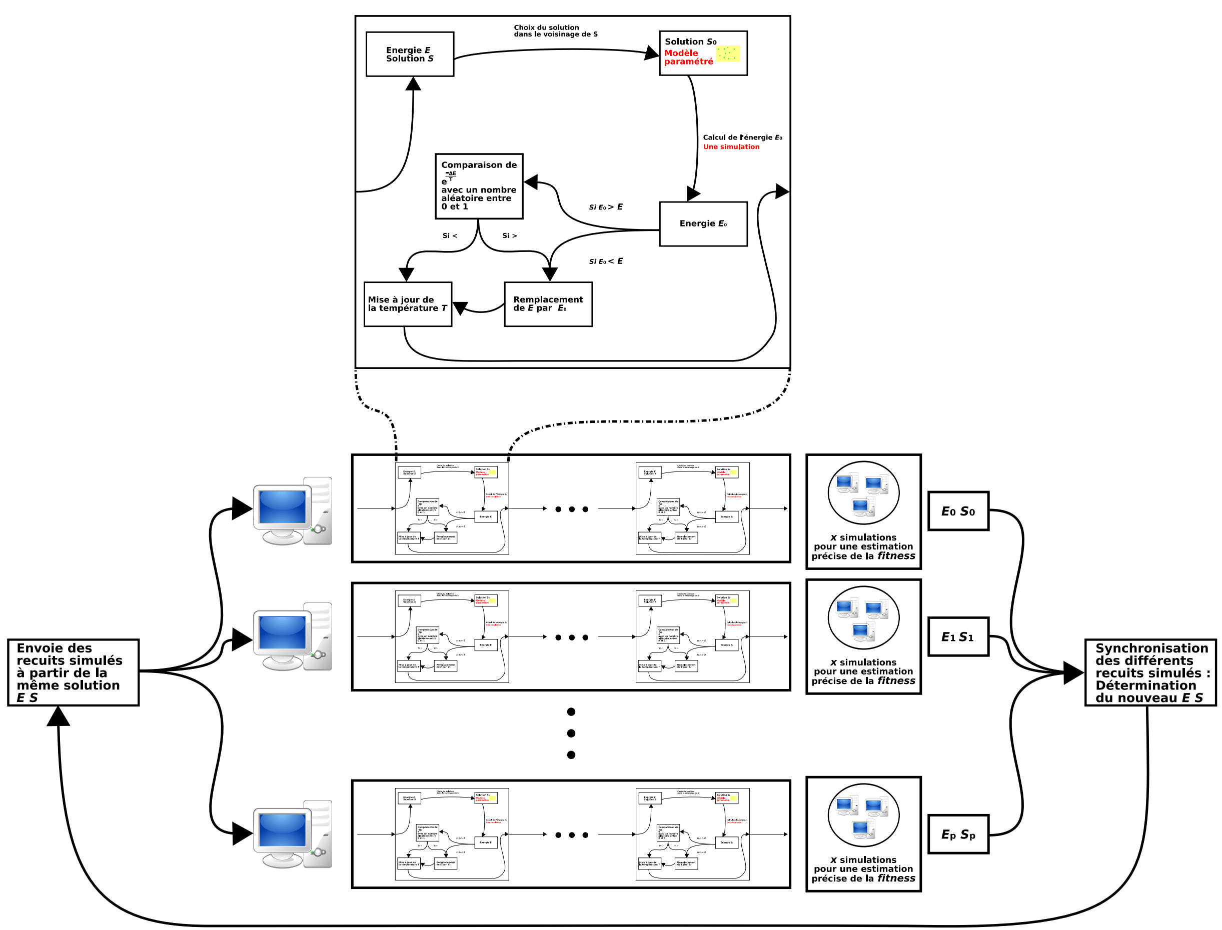


FIG. II.1.8 – *Schéma de la méthode avec recuit simulé*

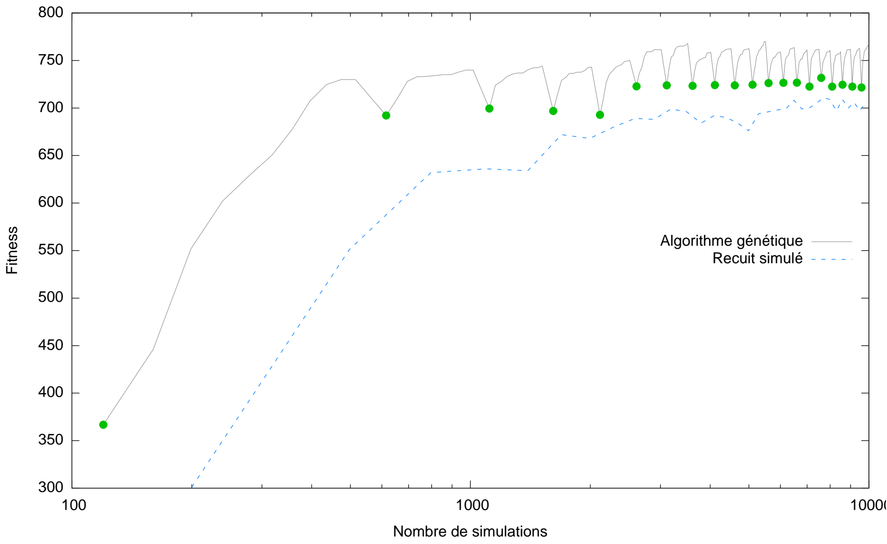


FIG. II.1.9 – *Résultat des méthodes algorithme génétique et recuit simulé sur le modèle de fourragement par colonie de fourmis*

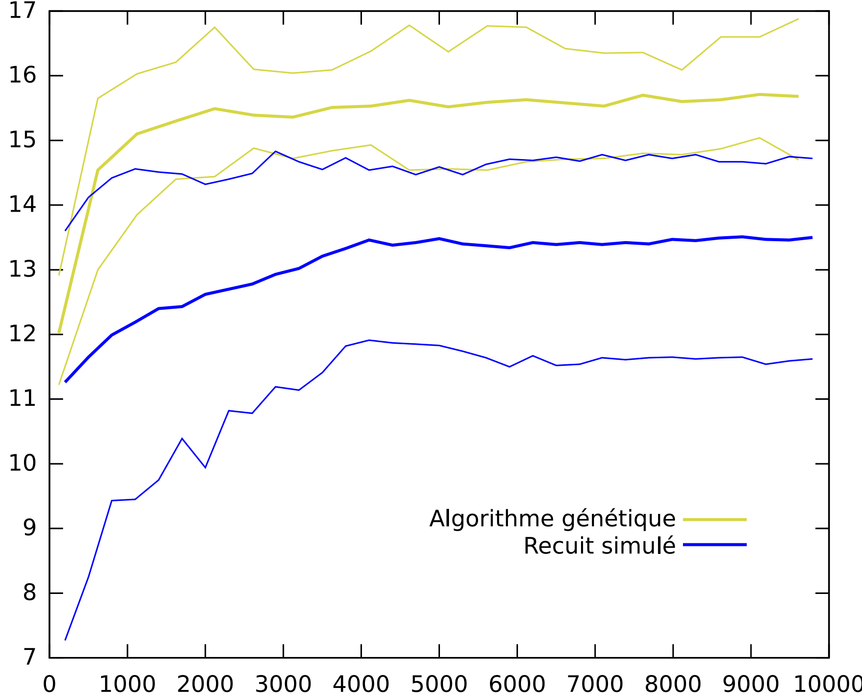


FIG. II.1.10 – *Évolution de la fitness à différentes exécutions de la méthode à base d'algorithme génétique et à base de recuit simulé*

Finalement la conclusion est donc identique à la conclusion de la sous-section I.3.6 page 27 sur le théorème « *No Free Lunch* » : aucune méta-heuristique n'est la meilleure quel que soit le problème. Étant donné que, dans cette approche, nous considérons la simulation multi-agent comme une boîte noire, nous n'avons aucune idée du paysage de l'espace des paramètres, et donc aucune possibilité de se rapprocher d'un type de fonction à optimiser connu : il n'est pas possible de déterminer a priori l'algorithme d'optimisation qui sera le plus efficace pour un modèle donné.

II.1.5 Discussion

Nous avons expliqué dans ce chapitre l'utilisation d'algorithmes d'optimisation, plus particulièrement les algorithmes génétiques et le recuit simulé dans le paramétrage de simulations multi-agents. Cette approche ne nécessite pas a priori des connaissances sur le modèle à base d'agents. Elle permet d'obtenir des résultats qui optimisent notre *fitness* et permettent ainsi de proposer un jeu de paramètres satisfaisant notre objectif de calibrage. Ces résultats s'expliquent par le fait que nous avons une vision en boîte noire du modèle : si cette boîte noire ne renvoie pas des résultats trop fortement stochastiques, et si nous tenons compte de ces aspects stochastiques, il apparaît logique que l'optimisation réussisse. Cette première méthode servira de méthode de référence par rapport aux autres méthodes, développées par la suite dans cette thèse.

Au contraire, du fait de cette approche en boîte noire, cette méthode n'exploite pas le fait de travailler sur un modèle à base d'agents. À chaque simulation, nous ne testons qu'un seul jeu de paramètres.

Il y a aussi tout un aspect de paramétrage de l'algorithme d'optimisation. D'abord, il faut choisir l'algorithme d'optimisation : algorithme génétique, recuit simulé ou autres... Après avoir choisi le type d'algorithme, il faut aussi choisir la version particulière de l'algorithme à utiliser : dans le cas des algorithmes génétiques, algorithme élitiste ou non, représentation binaire ou réelle, etc. Il faut enfin fixer les paramètres de l'algorithme : cette étape n'est généralement pas triviale. Comme nous l'avons vu, il est difficile a priori de choisir l'algorithme d'optimisation.

Sur la base de ces différents constats, nous nous sommes attachés à proposer une nouvelle mé-

thode d'exploration de l'espace des paramètres, plus simple que l'approche basée sur les heuristiques d'optimisation (en termes de nombres de paramètres de la méthode en elle-même), et en nous appuyant sur les spécificités de la modélisation à base d'agents.

II.2 Découpage de paramètres ou optimisation dichotomique adaptative

Dans notre objectif de calibrage de modèles, nous avons décrit des approches par exploration d'un espace des paramètres donné. Cependant, il n'est pas forcément utile d'explorer tout l'espace des paramètres. De plus, une exploration exhaustive n'est pas réalisable (voir la section I.2.2 page 16). Ainsi, il faut se focaliser sur les zones intéressantes de cet espace. Pour mettre en œuvre cette idée, une source d'inspiration vient du principe de la recherche dichotomique : chaque paramètre est discrétisé en un ensemble d'intervalles et l'on ne développe la recherche que dans les intervalles qui ont produit les meilleurs résultats, en les subdivisant.

Par ailleurs, il ne faut pas oublier une caractéristique importante de la simulation multi-agent : une simulation multi-agent comporte plusieurs agents. En général, une grande partie de ces agents est identique (voir la liste des exemples de modèles multi-agents dans l'annexe B page 127). Cette particularité va être utilisée pour explorer l'espace des paramètres : pour une simulation donnée, les différents agents de même type vont être initialisés individuellement avec des jeux de paramètres différents. Ainsi, en une seule simulation, une partie plus importante de l'espace des paramètres peut être explorée. L'idée sous-jacente est que le résultat d'une simulation sera d'autant meilleur que la simulation compte de « bons » agents, c'est-à-dire des agents ayant un bon paramétrage.

C'est en partant de ces deux idées qu'a été créée la méthode « Découpage de paramètres » pour l'exploration de l'espace des paramètres. En résumé, plusieurs simulations sont réalisées : pour chaque simulation, les agents sont initialisés avec des valeurs de paramètres choisis à partir du découpage des paramètres en intervalles. À la fin de la simulation, une *fitness* est calculée : chaque intervalle de chaque paramètre reçoit autant de récompenses que la simulation compte d'agents qui ont des valeurs de ce paramètre dans cet intervalle. Et l'opération est recommencée plusieurs fois. Enfin, l'intervalle de chaque paramètre ayant reçu le plus de récompense est sélectionné et est coupé en deux.

Nous détaillons cette méthode dans ce chapitre en nous focalisant sur quelques points particuliers, puis nous nous intéressons à quelques variantes de la méthode avant de conclure.

II.2.1 Méthode en détail

II.2.1.1 Introduction informelle

La figure II.2.1 page suivante résume le principe général. Au début (schéma ①), les paramètres sont découpés en un ensemble d'intervalles. À partir de cet ensemble d'intervalles va être créé un modèle paramétré (phase ②) de la manière suivante : pour chaque paramètre de chaque agent, un intervalle est choisi avec une probabilité uniforme parmi les intervalles de ce paramètre, puis une valeur est choisie avec une probabilité uniforme dans cet intervalle. En répétant la même procédure pour chaque paramètre de chaque agent, nous obtenons un ensemble d'agents paramétrés différemment. À partir de là, le modèle paramétré est simulé. À la fin de la simulation est calculée la *fitness*. Puis les intervalles sont récompensés (phase ③) : chaque intervalle de chaque paramètre reçoit autant de récompenses que la simulation compte d'agents qui ont des valeurs de ce paramètre dans cet intervalle. Puis le cycle *Création de modèle / Récompense* recommence. Au bout d'un certain nombre de cycles, une phase de découpage des paramètres s'effectue (phase ④) : durant cette phase, pour chaque paramètre, l'intervalle ayant reçu le plus de récompenses est sélectionné et est divisé en deux.

II.2.1.2 Formalisation

Il est possible de décrire de façon plus formelle la méthode.

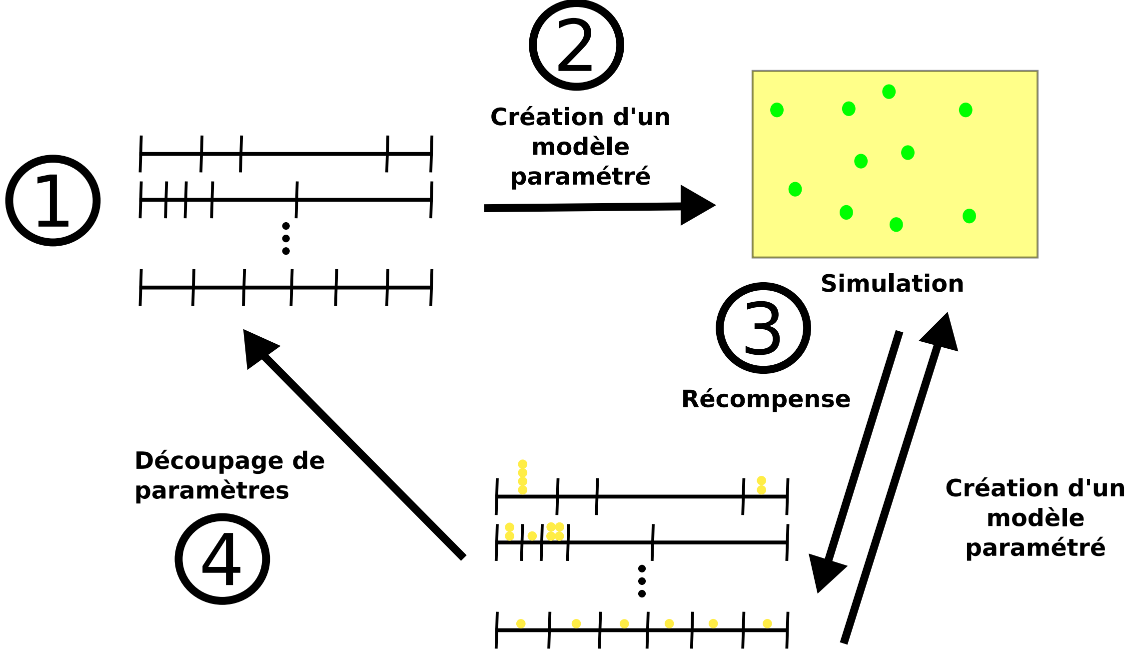


FIG. II.2.1 – *Méthode « Découpage de paramètres »*

Initialisation : phase ①

Soit une simulation multi-agent $\mathcal{S}$ dont les l ($l \in \mathbb{N}^*$) agents ont n paramètres P ($n \in \mathbb{N}^*$) tels que :

$$\forall i \in [\![0,n]\!] \;\; \exists(\alpha_i,\beta_i) \in \mathbb{R}^2 \;\; \text{tels que } \min P_i = \alpha_i \text{ et } \max P_i = \beta_i \text{ et } \alpha_i < \beta_i$$

Chaque paramètre P_i est divisé en un ensemble de m_i intervalles (où $m_i \in \mathbb{N}^+$) tel que :

$$\forall i \in [\![0,n]\!] \;\; P_i = \left\{ \bigcup_{j\in 0,\cdots,m_i-1} [t_j,t_{j+1}] \mid t_0 = \alpha_i \text{ et } t_{m_i} = \beta_i \text{ et } \forall k \in [\![0,m_i-1]\!] \; t_k < t_{k+1} \right\}$$

On définit pour chacun des m_i intervalles une variable appelé récompense : r_i^j où $j \in [\![0,m_i]\!]$

Création d'un modèle paramétré : phase ②

La réalisation d'un modèle nécessite d'abord la création des agents. Pour chaque paramètre de chaque agent, un intervalle est choisi avec une probabilité uniforme parmi les intervalles de ce paramètre, puis une valeur est choisie avec une probabilité uniforme dans cet intervalle. Plus formellement, les paramètres de chaque agent sont initialisés aléatoirement de la façon suivante :

- pour chaque agent $k \in [\![0,l]\!]$,
  - pour chacun de ses paramètres P_i^k où $i \in [\![0,n]\!]$,
    - la valeur de ce paramètre p_i^k est choisie de la façon suivante :
      1. un tirage aléatoire est effectué avec une probabilité uniforme sur l'ensemble des m_i intervalles du paramètre P_i.
         Supposons que l'intervalle tiré soit l'intervalle $[t_j,t_{j+1}]$ où $j \in [\![0,m_i]\!]$.
      2. Un tirage aléatoire est ensuite effectué avec une probabilité uniforme sur cet intervalle $[t_j,t_{j+1}]$.
    - Le résultat de ce tirage définit la valeur de p_i^k.

La figure II.2.2 résume le principe.

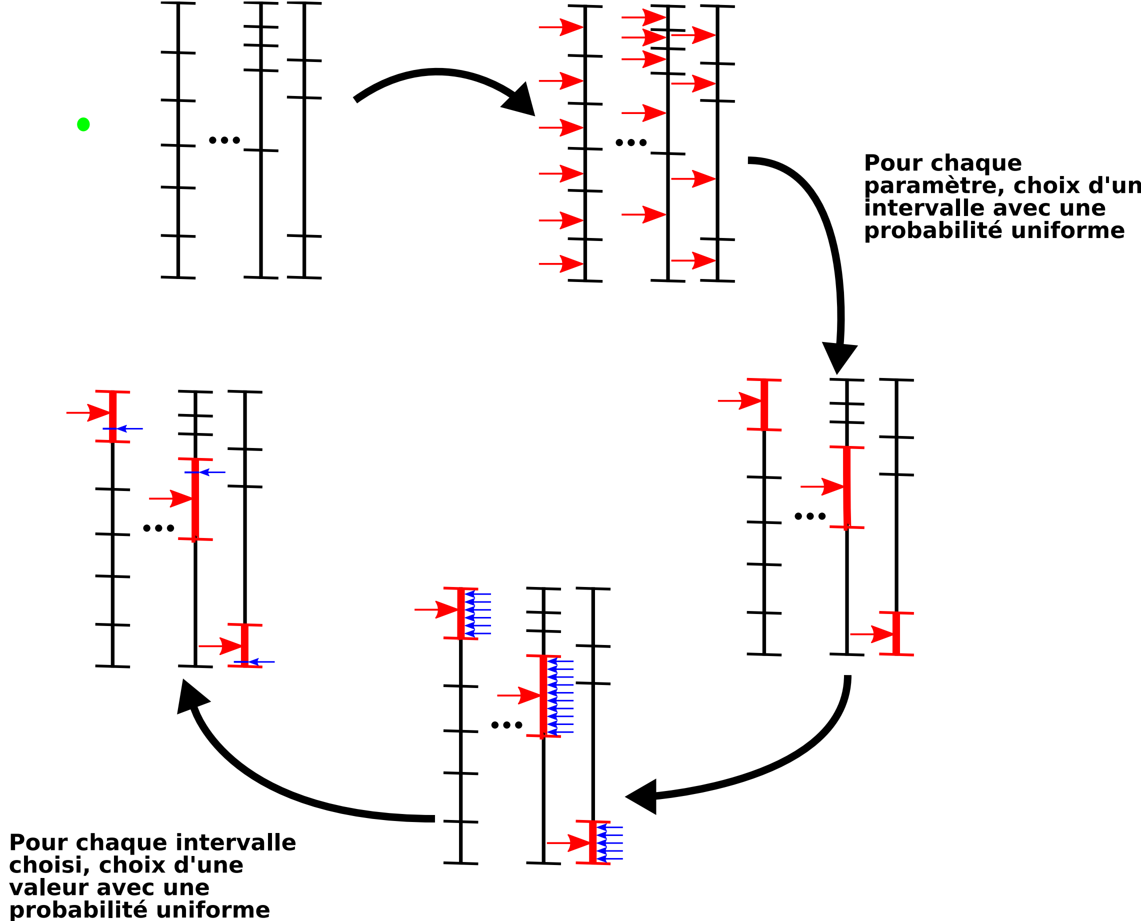


FIG. II.2.2 – *Initialisation des paramètres d'un agent*

Récompense : phase ③

À la fin de la simulation est calculée la *fitness* f, où $f \in \mathbb{R}$, est calculée. Puis les intervalles sont récompensés : chaque intervalle de chaque paramètre reçoit autant de récompenses que la simulation compte d'agents qui ont des valeurs de ce paramètre dans cet intervalle. Plus formellement, la phase de récompense des paramètres est exécutée de la façon suivante :

Pour chaque agent $k \in [\![0,l]\!]$, pour chacun de ses paramètres P_i^k où $i \in [\![0,n]\!]$,

$$\exists j \in [\![0,m_i]\!] \text{ tel que } p_i^k \in [t_j,t_{j+1}] \text{ alors } r_i^j \leftarrow r_i^j + f$$

Sélection et découpage : phase ④

Durant cette phase, pour chaque paramètre, l'intervalle ayant reçu le plus de récompenses est sélectionné et est divisé en deux.

Plus formellement, pour chaque paramètre P_i où $i \in [\![0,n]\!]$,

$$\exists a \in [\![0,m_i]\!] \text{ tel que } r_i^a = \max_{k \in [\![0,m_i]\!]} r_i^k$$

Alors, l'ensemble des intervalles du paramètre P_i est modifié de la façon suivante :

$$P_i = \left\{ \bigcup_{j \in 1,\cdots,m_i+1} [t'_j,t'_{j+1}] \mid t'_0 = \alpha_i \text{ et } t'_{m_i+1} = \beta_i \text{ et } \forall k \in [\![0,m_i+1]\!]\ t'_k < t'_{k+1} \right\}$$

où

$$\forall l \in [\![0,a]\!]\ t'_l = t_l$$

et

$$\forall l \in [\![a+2,m_i+1]\!]\ t'_l = t_{l-1}$$

et

$$t'_{a+1} = \frac{t_{a+1} - t_a}{2}$$

Les sélections se font de manière synchrone. La figure II.2.3 résume la phase de sélection.

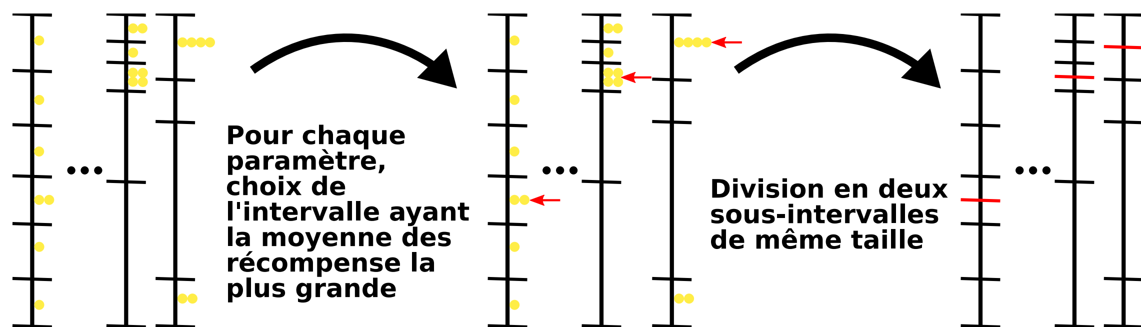


FIG. II.2.3 – *Sélection des intervalles*

II.2.1.3 Remarque sur l'espace des paramètres

Supposons que nous ayons une simulation avec n paramètres. L'espace des paramètres est un espace de dimension n : dans le cas de paramètres continues, l'espace de recherche est alors de l'ordre $\mathbb{R}^n$. Durant la phase de sélection, le nombre d'intervalles augmenterait de $2^n - 1$, ce qui causerait de grandes difficultés. Avec un tel espace de recherche, la méthode proposée dans ce chapitre ne serait pas utilisable.

Il faut donc faire une hypothèse pour simplifier l'espace des paramètres. Une hypothèse a été implicitement faite : il est possible de calibrer chaque paramètre indépendamment des autres c'est-

à-dire que le découpage est indépendant des autres découpages. Dans ce cas-là, l'espace des paramètres est divisé en n paramètres eux-même divisés en m_i intervalles : l'espace de recherche est alors de $n \times \mathbb{R}$. Avec cette hypothèse, le nombre d'intervalles augmente seulement de n lors de la phase de découpage.

Cette hypothèse peut se justifier par le fait que de nombreuses simulations avec de nombreux agents sont exécutées, ce qui implique que la totalité de l'espace des paramètres est globalement couvert. Par ailleurs, cette hypothèse a été confirmée du point de vue empirique : la méthode a été testée sur différents modèles à base d'agents, et l'hypothèse s'est avérée correcte. La figure II.2.4 montre sur un exemple la distribution des valeurs de deux paramètres pour les 100 premières simulations de l'exemple décrit à la section II.2.2 page 50. Malgré le peu d'agents dans ce modèle (seulement 25 agents), la distribution couvre globalement tout l'espace. Cependant, si des problèmes surviennent (par exemple, deux paramètres fortement liés), alors il est possible de poser une hypothèse moins forte : par exemple, il serait possible de considérer des groupes de paramètres indépendants à la place des paramètres individuellement indépendants.

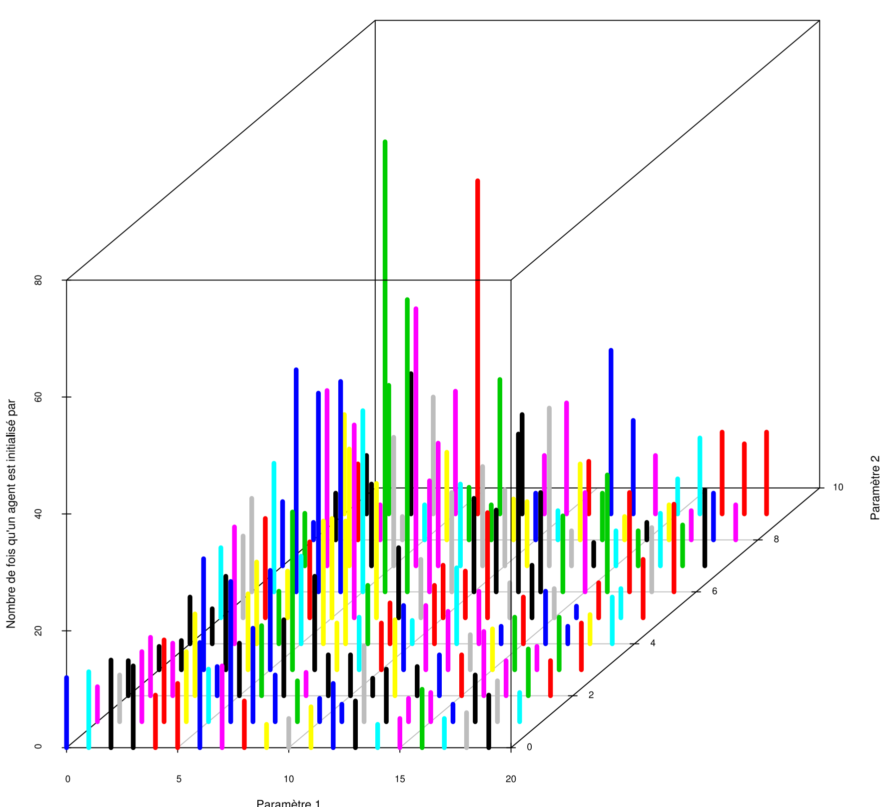


FIG. II.2.4 – *Distribution des valeurs de paramètres pour les* 100 *premières simulations*

Un autre point à noter concernant l'espace des paramètres est l'initialisation des agents. Le choix que nous avons effectué est que pour chaque agent, nous allons fixer une valeur pour chaque intervalle de la manière suivante : d'abord, un intervalle est choisi parmi tous les intervalles du paramètre avec une probabilité uniforme ; puis, une valeur est choisie dans l'intervalle avec une probabilité uniforme. Ce mode opératoire dans la création d'un agent a été choisi afin que n'importe quelle valeur puisse être choisie. Même jusqu'à la fin de l'algorithme, quoiqu'il arrive, des valeurs dans des zones de l'espace potentiellement inintéressantes auront toujours une probabilité non nulle d'être choisie. L'objectif est d'éviter de converger vers un extremum local, et d'y rester. Une autre raison de ce choix est l'hypothèse de paramètres indépendants : si les paramètres ne sont pas totalement indépendants les uns des autres (c'est généralement le cas), restreindre au fur et à mesure l'espace des

valeurs de certains paramètres risque d'entraîner de nombreux problèmes dont l'oubli d'une grande partie de l'espace de recherche. Cependant il est possible d'imaginer d'autres techniques pour initialiser les paramètres des agents :

- une probabilité proportionnelle à la largeur de l'intervalle : ce choix favorise les grandes plages d'intervalles, dont celles a priori les moins intéressantes. Au cours de l'algorithme, si le découpage en intervalles converge, de nombreuses simulations seraient créées et exécutées comportant un grand nombre d'agents dont le jeu de paramètres aurait des valeurs potentiellement peu intéressantes. Cela entraînerait une augmentation du nombre de simulations. De plus, ce choix risque d'empêcher l'algorithme de converger en favorisant toujours les plus grands intervalles, les zones de l'espace des paramètres les moins intéressantes.
- une probabilité inversement proportionnelle à la largeur de l'intervalle : ce choix favorise les petites plages d'intervalles au détriment des grandes plages d'intervalle. Le risque avec un tel choix est une convergence rapide vers un extremum local.
- une probabilité uniforme directement sur la valeur des paramètres : avec ce choix, la notion de découpage est perdue.

Le choix qui a été fait semble le plus raisonnable.

II.2.1.4 Paramétrage de l'algorithme

Un aspect important est le paramétrage de la méthode elle-même, plus particulièrement du nombre de simulations entre chaque phase de sélection.

Ce paramètre est nécessairement variable : en effet, au départ, il y a peu d'intervalles, et donc il suffit de peu de simulations pour avoir une évaluation satisfaisante des intervalles des paramètres. Mais, à mesure que l'algorithme progresse, le nombre d'intervalles de chaque paramètre augmente, et donc pour avoir une évaluation correcte des intervalles, il faut réaliser plus de simulations. Alors, comment augmenter ce nombre de simulations entre chaque sélection ? La réponse dépend du nombre d'agents dans la simulation, et du nombre d'intervalles.

Soit une simulation multi-agent $\mathcal{S}$ avec l agents possédant n paramètres. Chaque paramètre P_i est divisé en un ensemble de m_i intervalles.

Soit A le nombre minimal de récompenses que doit recevoir un intervalle pour que son évaluation soit significative. Ce nombre comporte deux grandes composantes : une composante liée à la stochasticité, et une autre composante liée à la méthode elle-même.

Soit s le nombre de simulations entre chaque sélection. En moyenne, le nombre de récompenses reçues par intervalle du paramètre P_i sera : $\frac{s \times l}{m_i}$

Ce nombre doit être alors strictement supérieur au nombre minimal de récompenses : $\frac{s \times l}{m_i} > A$

D'où,

$$s > \frac{A \times m_i}{l}$$

Donc,

$$s > max_{i \in [\![0,n]\!]} \frac{A \times m_i}{l}$$

En pratique, après des expérimentation , une valeur intéressante de A apparaît être 250.

$$\sum_{k \in [\![2,x]\!]} \frac{A \times k}{l} = \frac{A}{l} \times \left(\frac{x^2 + x}{2} - 1 \right)$$

II.2.1.5 Calcul de la solution

Une des étapes les plus importantes de la méthode est le calcul du paramétrage solution. Après la convergence de l'algorithme, il est nécessaire de calculer un paramétrage solution et d'évaluer la *fitness* de cette solution. En effet, le résultat de la méthode est seulement une discrétisation de l'espace des paramètres. Mais cette discrétisation, achevée indépendamment sur chaque axe, ne fournit pas directement un jeu de paramètres de solution unique. À partir des découpages en intervalles de chaque paramètre, il va falloir construire une solution globale.

Une première possibilité est de recommencer une nouvelle itération de l'algorithme (c'est-à-dire le cycle de création de modèles paramétrés, puis de simulation et de récompense des intervalles). Le paramétrage solution sera l'intervalle de chaque paramètre ayant reçu la moyenne des récompenses la plus haute. Et à partir de ce paramétrage solution, la *fitness* globale est calculée en exécutant une simulation (plutôt plusieurs simulations à cause de la stochasticité) où les agents sont paramétrés avec le paramétrage solution.

Une autre possibilité est de considérer que l'intervalle solution de chaque paramètre est l'intervalle de longueur minimale. En effet, ces intervalles de longueur minimale ont reçu le plus de récompenses durant les itérations de l'algorithme et ont donc été les plus divisés. Ces intervalles sont a priori les plus intéressants pour le paramétrage solution. En fait, il y a toujours au moins deux intervalles de longueurs minimales, parce que durant la phase de sélection, l'intervalle, qui a reçu le plus de récompenses, est divisé en deux intervalles de longueurs identiques. Dans le cas de deux intervalles adjacents, il faut prendre l'union des deux intervalles. Et s'il y a des intervalles non adjacents, il faut tous les considérer comme intervalles solutions. À partir de la détermination des intervalles solutions, une *fitness* globale est calculée de la façon suivante :

- Une simulation est créée. Pour cela, des agents sont initialisés de la façon suivante : pour chaque paramètre, un intervalle est choisi parmi les intervalles de largeur minimale avec une probabilité uniforme, puis une valeur dans cet intervalle avec une probabilité uniforme.
- Le modèle est simulé.
- Puis, les intervalles sont récompensés.

Enfin, nous répétons plusieurs fois ces opérations. Puis, une *fitness* globale finale est déterminée : somme des *fitness* divisées par le nombre de simulation.

Après des tests variés, c'est cette dernière solution qui a été choisie pour calculer le paramétrage solution et la *fitness* globale finale.

II.2.1.6 Critère d'arrêt

Il faut aussi s'intéresser au critère d'arrêt de l'algorithme.

Une première idée est de suivre l'évolution de la *fitness* globale au fur et mesure de l'exécution de l'algorithme. Quand la *fitness* converge, alors il est possible d'arrêter l'algorithme. La figure II.2.5 montre un exemple de convergence de *fitness* globale. Dans ce cas, l'évolution de la *fitness* converge, et donc l'utilisateur peut décider de stopper le déroulement de l'algorithme. Mais, ce cas idéal ne se produit pas à chaque fois. Par exemple, la figure II.2.6 page suivante montre un exemple où une *fitness* globale ne converge pas. Ainsi ce premier critère d'arrêt ne permet pas de savoir quand s'arrêter dans le cas où la *fitness* globale ne converge pas de façon franche.

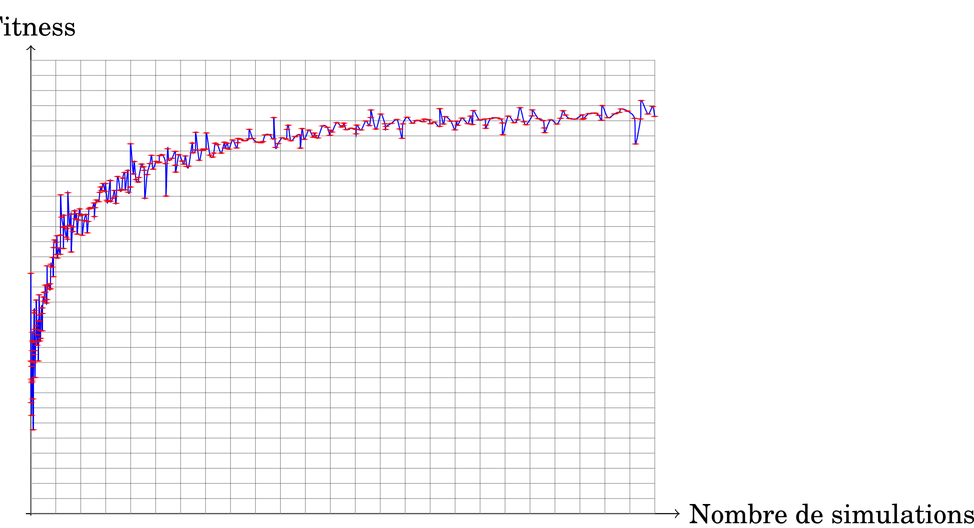


FIG. II.2.5 – *Exemple d'une convergence de fitness globale*

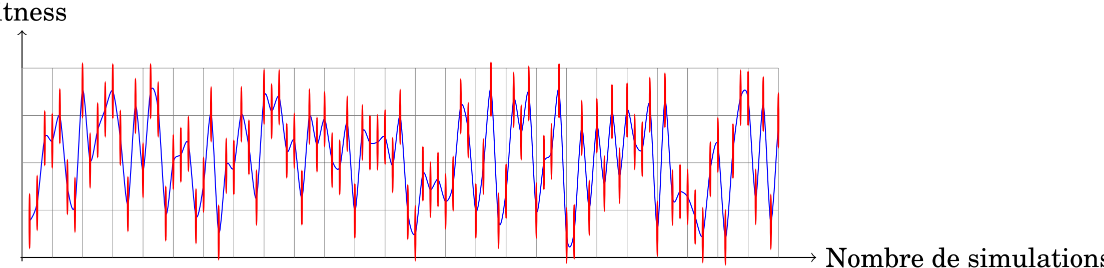


FIG. II.2.6 – *Exemple d'une non-convergence de fitness globale*

Au lieu de s'intéresser à la convergence de la *fitness* globale, il est possible de regarder le découpage des paramètres en intervalles. Un critère pourrait être la convergence du découpage en intervalles des paramètres, au lieu de la *fitness* globale. La figure II.2.7 montre un exemple de convergence, tandis que la figure II.2.8 montre un exemple de non-convergence. L'inconvénient de ce critère est alors qu'il n'est possible d'arrêter la méthode que quand il y a convergence nette. Un autre critère d'arrêt pourrait être une longueur minimale que devra atteindre un des intervalles de chaque paramètre. Cependant, ce critère peut être très coûteux en imaginant un découpage très régulier des paramètres ou pour des paramètres qui n'influencent pas réellement la qualité du modèle, donc qui seront découpés en intervalles de manière aléatoires sans convergence nette. Ce critère peut servir de critère maximal d'arrêt.

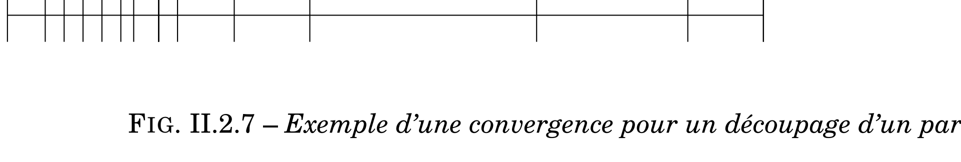


FIG. II.2.7 – *Exemple d'une convergence pour un découpage d'un paramètre*

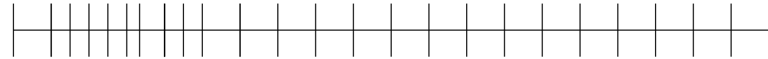


FIG. II.2.8 – *Exemple de non-convergence pour un découpage d'un paramètre*

Comme pour la méthode avec des algorithmes génétiques (sous-section II.1.1.2 page 32), arrêter la méthode à partir de l'observation d'une convergence n'est pas évident. Une solution est alors de se donner un nombre maximal de simulations à réaliser, et d'arrêter la méthode dès que le nombre de simulations dépasse ce nombre. C'est le choix arbitraire qui a été fait dans ce manuscrit, dans un soucis pragmatique de développement de la méthode, renvoyant à une étude ultérieure la délicate question du critère d'arrêt.

II.2.2 Premiers résultats

Cette section décrit l'application de la méthode « découpage de paramètres » avec un modèle à base d'agents. Le modèle utilisé et les conditions de tests sont identiques au test de la méthode avec algorithme génétique, décrit à la sous-section II.1.2.1 page 35 : même modèle, même fonction de *fitness*, même paramètres, même nombre de pas de simulation, même nombre de simulations.

La figure II.2.9 page suivante montre l'évolution de la *fitness* en fonction du nombre de simulations. Cette courbe montre la convergence assez rapide de la *fitness* en peu de simulations, ce qui est un bon résultat. La table II.2.1 page ci-contre montre le paramétrage solution obtenu. Le résultat est similaire à celui de la méthode algorithme génétique, décrit à la sous-section II.1.2.2 page 36. Un des aspects les plus intéressants de la méthode est l'étude du découpage des paramètres. La figure II.2.10 page 52 montre l'évolution du découpage du paramètre `speed`. Au commencement de l'algorithme, le paramètre est divisé en dix parts égales, puis, au fur et à mesure des itérations de

l'algorithme, le découpage évolue : dans cet exemple, il y a convergence des découpages vers une région précise du paramètre. Cela illustre un des intérêts de la méthode : la méthode démarre avec l'espace des paramètres entier, avant de se focaliser vers les parties les plus intéressantes. En plus de fournir un « meilleur » jeu de paramètres solution, cette méthode fournit en plus une sorte de cartographie de l'espace des paramètres. La figure II.2.11 page suivante montre l'évolution du découpage d'un paramètre artificiel, dont la valeur n'a aucune influence sur la simulation. À la différence du paramètre précédent, il n'y a pas de convergence dans le découpage : la méthode continue toujours d'explorer tout l'espace des paramètres.

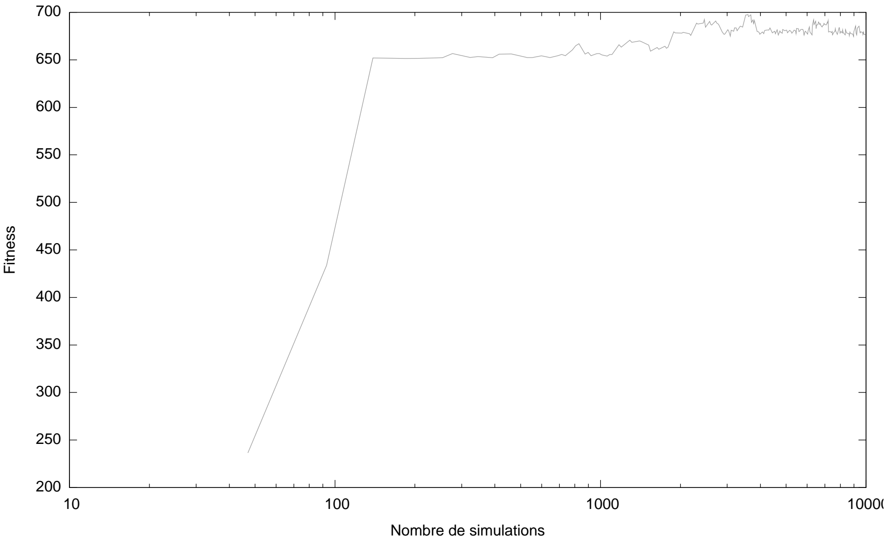


FIG. II.2.9 – *Évolution de la fitness en fonction du nombre de simulations avec la méthode découpage de paramètres*

nom du paramètre	valeur
speed	8,247
patch_ahead	8,989
angle_vision	208,492
drop_size	87,421

TAB. II.2.1 – *Paramétrage solution obtenu avec la méthode découpage de paramètres*

Cette méthode permet d'optimiser les paramètres internes d'un agent. Cependant, cette méthode, dans cette première approche, ne gère pas le calibrage de paramètres globaux à la simulation. Cette difficulté peut être résolue de la manière suivante : le modèle avec ses paramètres globaux est considéré comme un agent avec des paramètres locaux. La méthode s'appliquera donc sur cet agent « global » comme elle s'applique sur un agent « local ». Ainsi cette méthode aurait deux niveaux de granularité : un niveau global avec les paramètres globaux au modèle et un niveau local avec les paramètres propres aux agents. Mais, ce choix ajoute de nombreuses simulations entre chaque sélection : à la différence des paramètres locaux aux agents, ce paramètre est propre à un seul pseudo-agent, et donc, chaque simulation apportera seulement une évaluation pour ce paramètre alors que pour un paramètre local, elle apportera autant d'évaluations qu'il y a d'agents. Généralement, un modèle à base d'agents possède des paramètres globaux : une modification de la méthode précédente est requise afin de tenir compte de ces deux points.

II.2.3 Variantes

Nous avons décrit la méthode découpage de paramètre précédemment dans le cadre général. Dans cette section, nous allons nous focaliser sur quelques variantes de cette méthode.

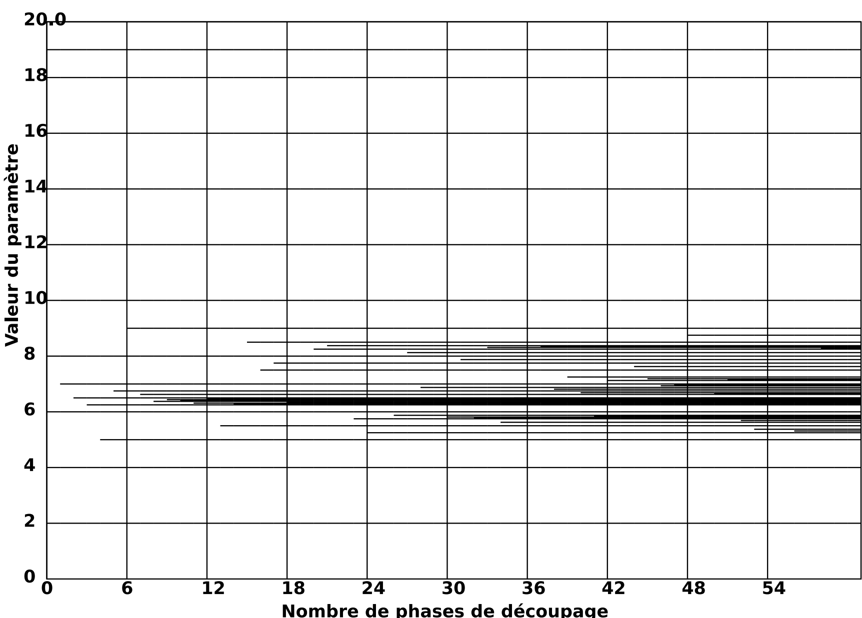


FIG. II.2.10 – *Évolution du découpage du paramètre* `speed` *avec la méthode découpage de paramètres*

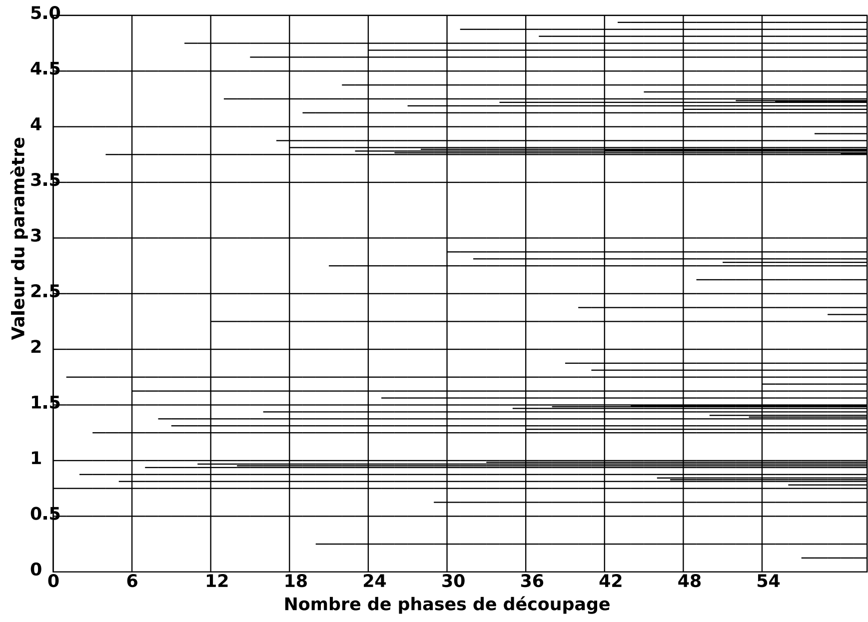


FIG. II.2.11 – *Évolution du découpage du paramètre* `foo` *avec la méthode découpage de paramètres*

II.2.3.1 Méthode asynchrone

Comme le suggère la section précédente, si nous cherchons à optimiser simultanément des paramètres internes aux agents et des paramètres globaux, ces derniers recevront une unique récompense chacun à chaque simulation, tandis que les premiers pourront avoir reçu de multiples récompenses. Lors des phases de sélection, les paramètres globaux auront donc reçu beaucoup moins de récompenses que les paramètres locaux, et la pertinence des découpages en intervalles de ces paramètres globaux sera donc plus faible. Ainsi, il faudra augmenter le nombre de simulations pour pouvoir obtenir une pertinence raisonnable pour les paramètres globaux. Cette augmentation du nombre de simulations va être coûteuse, surtout pour les paramètres locaux.

Pour résoudre ce problème, une idée est d'arrêter de réaliser la phase de sélection et de découpage de manière synchrone sur tous les paramètres. Dans cette variante, chaque paramètre réalise la phase de sélection et de découpage de manière indépendante : quand le nombre de récompenses et la répartition de ces récompenses lui semblent satisfaisantes, la phase de sélection et de découpage est exécutée sur ce paramètre.

II.2.3.2 Ajout d'aléatoire

Comme le montre la figure II.2.9 page 51, la *fitness* converge, mais à un niveau plus bas qu'avec la méthode par algorithme génétique : elle a probablement convergé vers un optimum local. Pour éviter ce problème, une idée est d'ajouter de la stochasticité. Cet ajout d'aléatoire peut se faire de deux façons. Une première manière est de choisir de façon complètement aléatoire des intervalles à diviser et à fusionner. Cependant il est possible de faire mieux en essayant de diriger l'aléatoire. Une première idée de cet aléatoire dirigé est d'explorer tout l'espace des paramètres en n'oubliant pas des zones qui semblent a priori peu intéressantes (c'est-à-dire celles qui sont peu découpées) : pour cela, il suffit de découper ces zones en de plus petits intervalles. Une deuxième idée est d'éviter de continuer d'explorer des zones qui semblaient auparavant intéressantes (c'est-à-dire fortement divisées), mais qui ne le sont plus : pour cela, pour éviter d'explorer ces zones devenues peu intéressantes, il suffit de fusionner ces zones en plus grands intervalles. En résumé, toutes les n itérations de l'algorithme, une phase d'ajout d'aléatoire dirigé est exécutée : pour chaque paramètre, les intervalles qui ont été les moins sélectionnés (c'est-à-dire les zones de l'espace a priori les moins intéressantes) sont choisis. Parmi ceux-là, les plus grands Les plus grands (c'est-à-dire ceux qui représente plus qu'un vingtième de la largeur totale du paramètre, par exemple) sont divisés : c'est l'exploration de zones de l'espace peu intéressantes. Les plus petits sont fusionnés : c'est la fin de l'exploration de zones de l'espace qui étaient auparavant intéressantes, mais qui ne le sont plus.

Cette variante a été appliquée sur le même modèle que précédemment avec les mêmes conditions de tests. Cette méthode converge à une *fitness* supérieure par rapport à la méthode sans aléatoire, mais démarre plus lentement. La figure II.2.12 page suivante montre l'évolution du découpage d'un paramètre. Grâce à cette variante, des zones peu intéressantes sont explorées, et des zones, anciennement intéressantes ne sont plus explorées. Les barres grises foncées sur la figure correspondent à la fusion d'intervalles, et les barres grises claires à la division d'intervalle. Cependant, il y a autre intérêt : cette variante de la méthode est beaucoup plus stable que la méthode sans aléatoire dans la répétition des mêmes expérimentations.

II.2.3.3 Colonie de fourmis

Une autre approche pour rendre le découpage plus dynamique consiste à s'inspirer des algorithmes par colonies de fourmis, décrits à la sous-section I.3.4 page 23, et plus particulièrement aux *Ant Systems*. Il est possible de faire un parallèle entre les phéromones des algorithmes par colonies de fourmis et les récompenses données aux intervalles. Plus précisément, la moyenne des récompenses peut être vue comme les phéromones pour les algorithmes par colonies de fourmis. À la fin de chaque simulation, nous calculons la *fitness* de la simulation, et nous récompensons les intervalles : chaque intervalle reçoit autant de récompenses (c'est-à-dire de valeur de la *fitness*) qu'il a « participé » à la simulation. Ce sont ces récompenses qui symbolisent les phéromones. À chaque phase de sélection, une partie des récompenses (50%) s'évapore : cet action correspond à l'évaporation des phéromones.

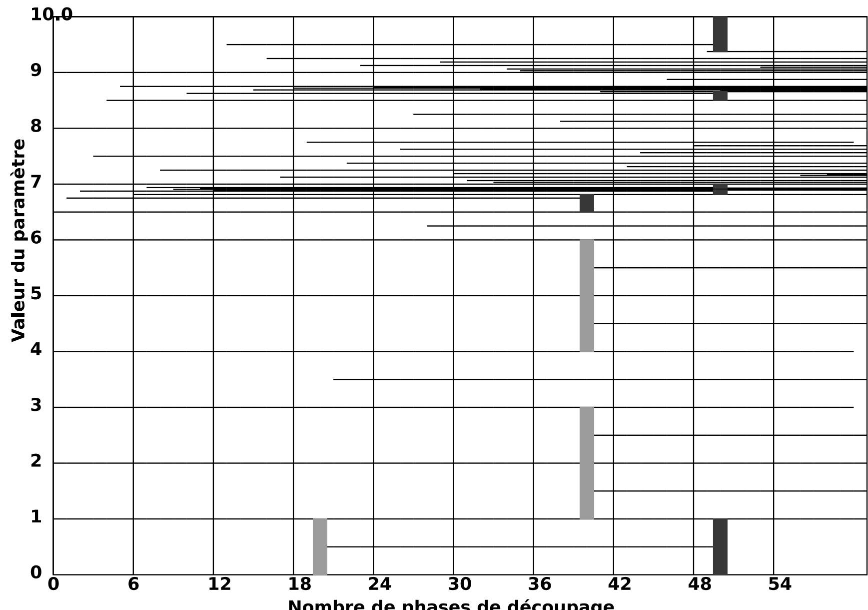


FIG. II.2.12 – *Évolution du découpage d'un paramètre* speed *avec la méthode découpage de paramètre avec ajout d'aléatoire*

Comme pour l'algorithme *Ant Systems*, il est possible d'ajouter une heuristique : à chaque intervalle est ajoutée une valeur d'heuristique, et une confiance sur cette valeur. Cette valeur d'heuristique est calculée en trois étapes. Dans la première étape, un modèle est créé où tous les agents sont initialisés avec le même jeu de paramètres qui est choisi de la manière suivante :

– pour le paramètre dont est issu l'intervalle spécifique dont la valeur d'heuristique est en train d'être calculée, la valeur médiane de l'intervalle est choisie ;
– pour les autres paramètres, la valeur médiane du paramètre est choisie.

La figure II.2.13 page suivante résume cette première étape. La seconde étape consiste à simuler le modèle ainsi initialisé (voire plusieurs simulations à cause de la stochasticité). Dans la dernière étape est calculée la valeur d'heuristique : la valeur d'heuristique est la moyenne des valeurs des *fitness* pour les différentes éxécutions de la simulation. La confiance sur cette valeur d'heuristique pour cet intervalle est alors égale à 1 après ce calcul. Lors d'une fusion de deux intervalles, la confiance est la moyenne des deux confiances, et la valeur d'heuristique est la moyenne des deux valeurs d'heuristique. Lors d'une division d'un intervalle en deux intervalles, la confiance des deux nouveaux intervalles est divisée par 2, et la valeur d'heuristique est la valeur d'heuristique de l'intervalle à diviser.

La phase de création des agents a été modifiée : d'abord pour chaque paramètre de chaque agent est choisi un intervalle avec une probabilité telle que la probabilité de choisir un intervalle j du paramètre i est

$$[\tau_{ij}]^{\alpha} \times [\eta_{ij}]^{\beta}$$

où τ est la quantité de phéromones de cet intervalle (c'est-à-dire la moyenne des récompenses), η est la valeur de l'heuristique de cet intervalle, et α et β sont les paramètres qui contrôlent l'importance entre la valeur d'heuristique et les phéromones. Puis, une valeur est choisie dans cet intervalle avec une probabilité uniforme.

La phase de sélection et de découpage d'intervalle a été modifiée. En effet, si nous avions gardé l'ancienne manière de diviser un intervalle, l'évaporation aurait tendance à supprimer les phéromones, et donc le choix de l'intervalle à découper ne serait pas pertinent.

Un intervalle j d'un paramètre i est divisé si sa quantité de phéromones est anormalement supérieure à la quantité moyenne de phéromones de tous les intervalles du paramètre i c'est-à-dire $\tau_{ij} > \overline{\tau_i} + 2 \times \sigma_i$ où σ_i est l'écart type et $\overline{\tau_i}$ est la moyenne de la quantité de phéromones du paramètre i. Un intervalle j d'un paramètre i est fusionné si sa quantité de phéromones est anormalement

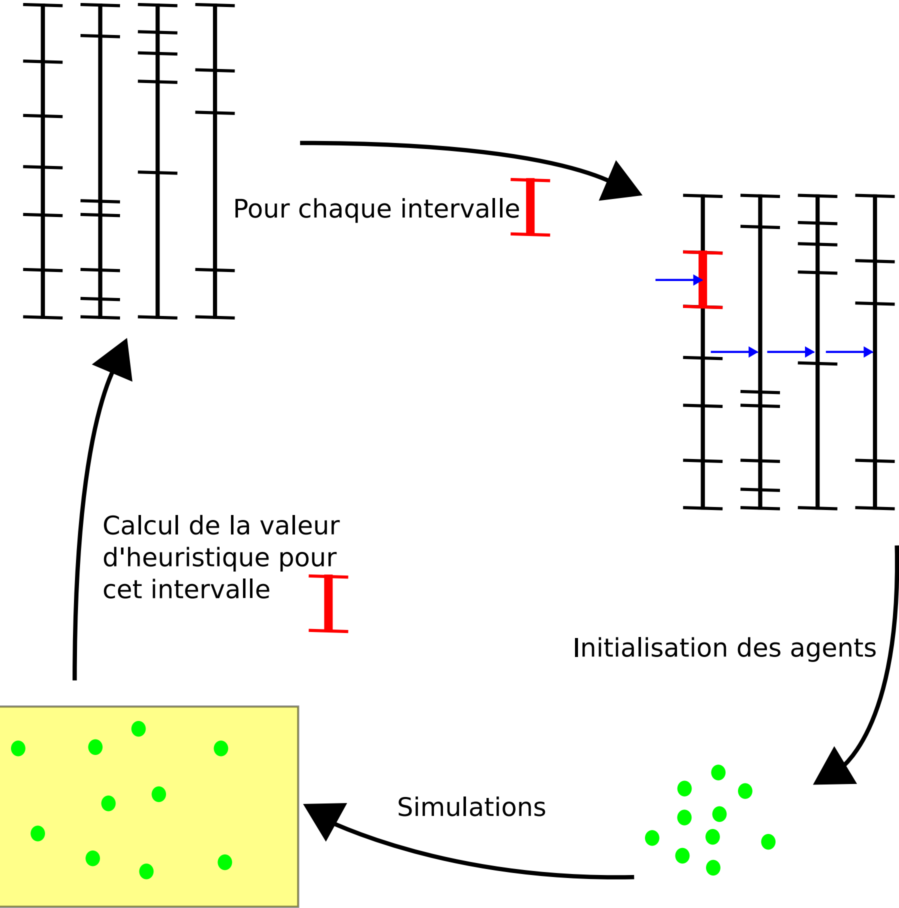


FIG. II.2.13 – *Principe de l'initialisation d'un modèle pour le calcul de la valeur d'heuristique d'un intervalle*

inférieure à la quantité moyenne de phéromones de tous les intervalles du paramètre i c'est-à-dire $\tau_{ij} < \overline{\tau_i} - \sigma_i$. Ce choix permet d'avoir des phases de découpage et de fusion beaucoup plus dynamique.

Pour résumer, il est possible de reprendre le schéma de l'algorithme *Ant Systems* :

- `ConstructionSolutionParFourmis()` : cette étape correspond à la phase d'initialisation du modèle avec le choix des valeurs des paramètres.
- `MiseAJourPheromones()` : cette étape correspond à la simulation du modèle et à la récompense des intervalles.
- `ActionsDémons()` : cette étape correspond à la mise à jour des valeurs d'heuristique.

La figure II.2.14 résume cette variante de la méthode s'inspirant des algorithmes de colonies de fourmis. Chaque fourmi est responsable du paramétrage d'un agent. Chaque fourmi voyage à travers les paramètres pour choisir un intervalle par paramètre. Le choix de cet intervalle dépend de la quantité de phéromones et de l'heuristique. À la fin du voyage des fourmis, nous obtenons les différents intervalles pour créer les agents.

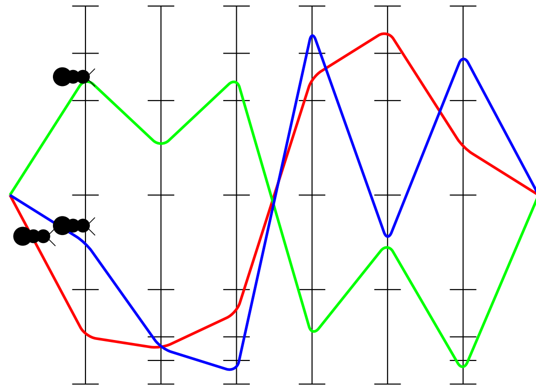


FIG. II.2.14 – *Schéma résumant la variante de la méthode découpage de paramètre s'inspirant des algorithmes de colonies de fourmis*

Cette variante a été également testée sur le calibrage du modèle de fourragement avec les mêmes conditions de tests. La figure II.2.15 page suivante montre l'évolution de la *fitness* en fonction du nombre de simulations. La *fitness* converge très rapidement avec peu de simulations. Le résultat le plus intéressant est réside dans le découpage en intervalles des paramètres. La figure II.2.16 page ci-contre montre l'évolution du découpage du paramètre `speed`. La figure II.2.17 page suivante montre l'évolution du découpage d'un paramètre artificiel. Avec cette variante, il y a beaucoup plus de phases de division et de fusion de paramètres : le découpage est beaucoup plus dynamique. Au niveau du paramétrage solution, les conclusions sont similaires à celle du modèle avec algorithme génétique.

Nous avons cherché à déterminer l'influence de la valeur des paramètres α et β qui contrôlent la phase de création des agents, sur l'efficacité de la méthode. La figure II.2.18 page 58 montre l'évolution des résultats de la méthode en fonction de α et de β comparée avec une courbe de référence. Cette courbe de référence représente l'utilisation de cette variante de la méthode sans utiliser l'heuristique, et sans réaliser de simulations supplémentaires pour le calcul de l'heuristique. Cette courbe s'appuie donc seulement sur les phéromones. Cette figure a été réalisée par l'intermédiaire de pseudo-simulations (expliqué dans l'annexe E page 143, plus particulièrement à la section E.5 page 149 pour la fonction mathématique) avec 26 exécutions pour chaque variante. Comme le montre la figure, les meilleurs résultats sont obtenus avec de faibles valeurs de β : cela montre que l'information heuristique ajoutée apparaît peu pertinente et qu'une méthode sans cette information (et toutes les simulations qui en découlent) est plus intéressante. Il est sûrement possible de réaliser un meilleur choix pour l'heuristique. Cependant, le calcul de l'heuristique est coûteux en simulations. Finalement, étant donné le coût de l'heuristique, et sa faible efficacité, la meilleure solution semble être d'appliquer cette méthode sans utiliser l'heuristique, en absence de meilleure heuristique.

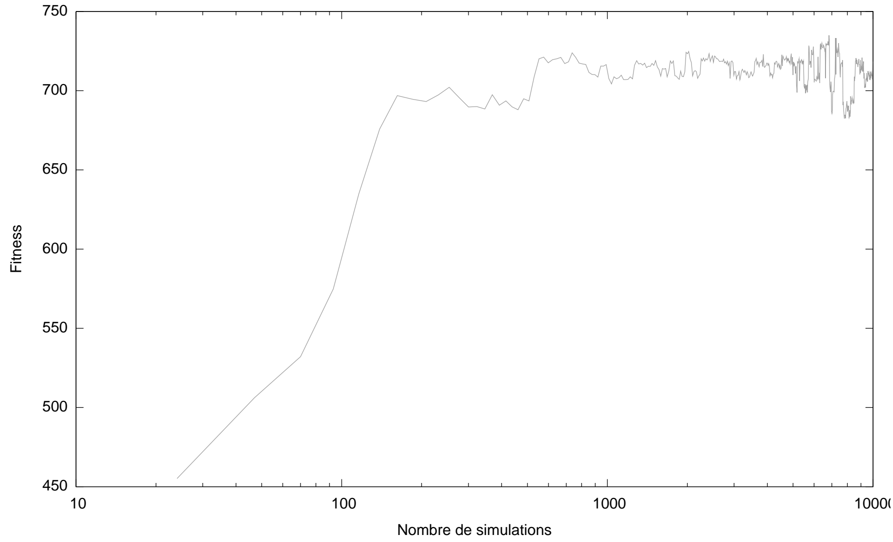


FIG. II.2.15 – *Évolution de la fitness en fonction du nombre de simulations avec la méthode découpage de paramètres avec inspiration des colonies de fourmis*

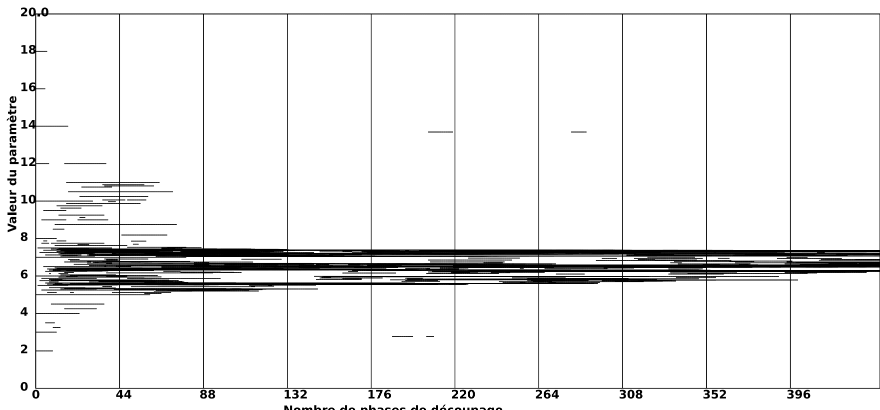


FIG. II.2.16 – *Évolution du découpage du paramètre* `speed` *avec la méthode découpage de paramètre avec inspiration des colonies de fourmis*

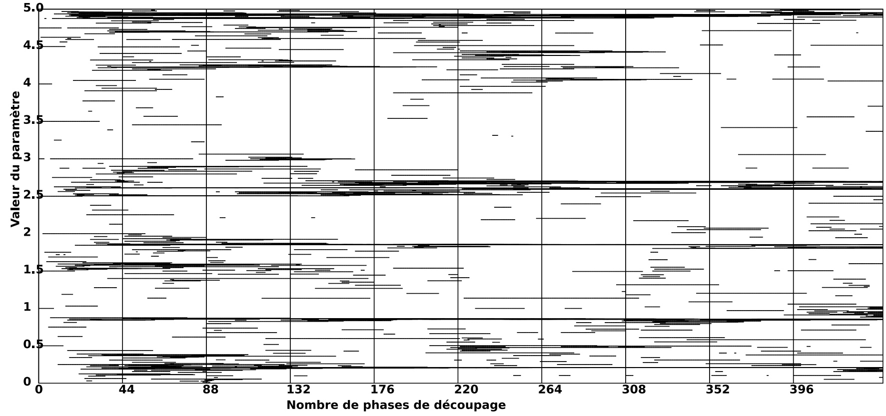


FIG. II.2.17 – *Évolution du découpage d'un paramètre artificiel avec la méthode découpage de paramètre avec inspiration des colonies de fourmis*

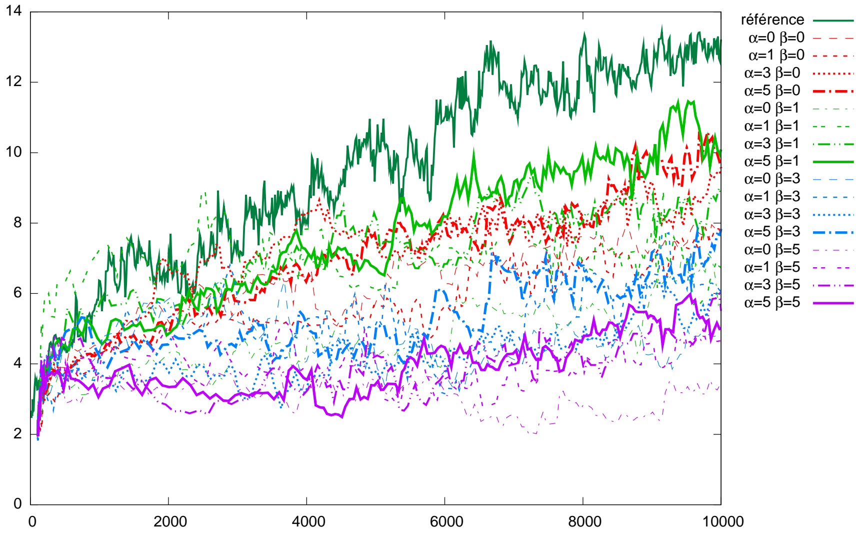


FIG. II.2.18 – *Évolution de α et de β*

II.2.3.4 Variante où le nombre de découpages est fixé

Un aspect négatif de ces méthodes de découpage de paramètres, et plus particulièrement la méthode s'inspirant des colonies de fourmis est l'obtention d'un très grand nombre d'intervalles. Si nous utilisons la première méthode présenté dans ce chapitre, il faut que chaque intervalle ait reçu un certain nombre de récompenses pour que la phase de découpage ait lieu : plus il y a d'intervalles, plus il est nécessaire de réaliser de simulations entre chaque phase de sélections.

Une idée est de limiter le nombre d'intervalles pour réduire le nombre de simulations entre deux phases de découpage. À partir de la variante avec ajout d'aléatoire, nous créons une nouvelle variante : le nombre maximal de découpages d'un paramètre est fixé. Initialement il n'y a pas de différence avec ajout d'aléatoire. C'est seulement quand le nombre maximal de découpages d'un paramètre que la méthode est différente : pour rajouter un nouvel intervalle, la méthode fusionne l'intervalle ayant la moyenne des récompenses avec l'intervalle voisin ayant la moyenne la plus basse.

Nous avons testé cette approche sur deux cas : 25 intervalles et avec 100 intervalles. Les figures II.2.19 page suivante et II.2.22 page 60 montrent l'évolution de la *fitness* en fonction du nombre de simulations pour les cas 25 intervalles et 100 intervalles. Les figures II.2.20 page suivante et II.2.23 page 61 montrent l'évolution du découpage du paramètre `speed` pour les cas 25 intervalles et 100 intervalles. Les figures II.2.21 page 60 et II.2.24 page 61 montrent l'évolution du découpage du paramètre artificiel pour les cas 25 intervalles et 100 intervalles. Du point de vue de l'évolution de la *fitness*, les résultats sont plus mauvais (dans le cas général, mais il y a des exemples où c'était le contraire). L'intérêt réside surtout au niveau du découpage des paramètres qui est beaucoup plus net.

II.2.4 Résultats

Dans cette section, nous comparons les résultats des différentes méthodes de découpage de paramètres avec la méthode algorithme génétique et une méthode test. Cette méthode test consiste à parcourir l'espace des paramètres de façon systématique : la méthode de découpage en 2-étapes ou n-étapes.

II.2.4.1 Méthode de découpage en 2-étapes ou n-étapes

Cette méthode consiste à parcourir l'espace des paramètres de façon systématique. Par exemple, si le modèle comporte trois paramètres, alors l'espace de recherche est un parallélépipède. Si la

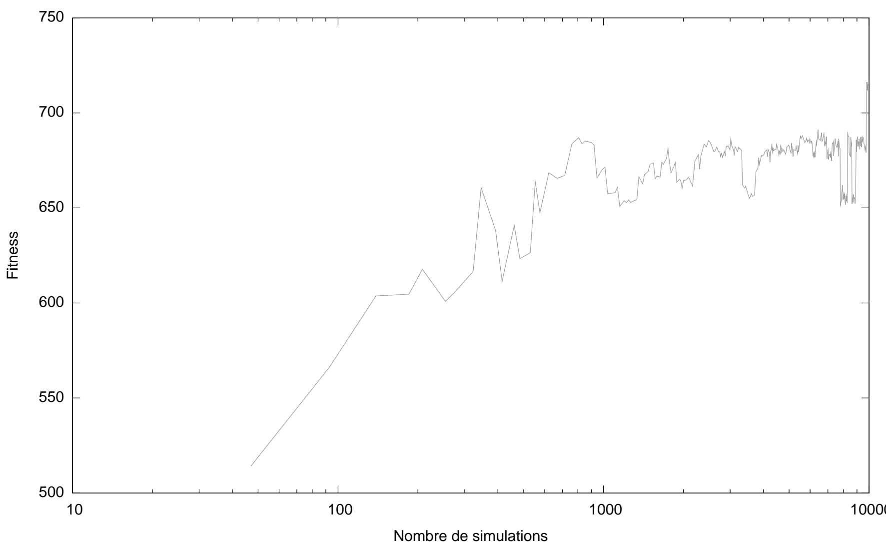


FIG. II.2.19 – *Évolution de la fitness en fonction du nombre de simulations avec la méthode découpage de paramètres en limitant à 25 intervalles par paramètres*

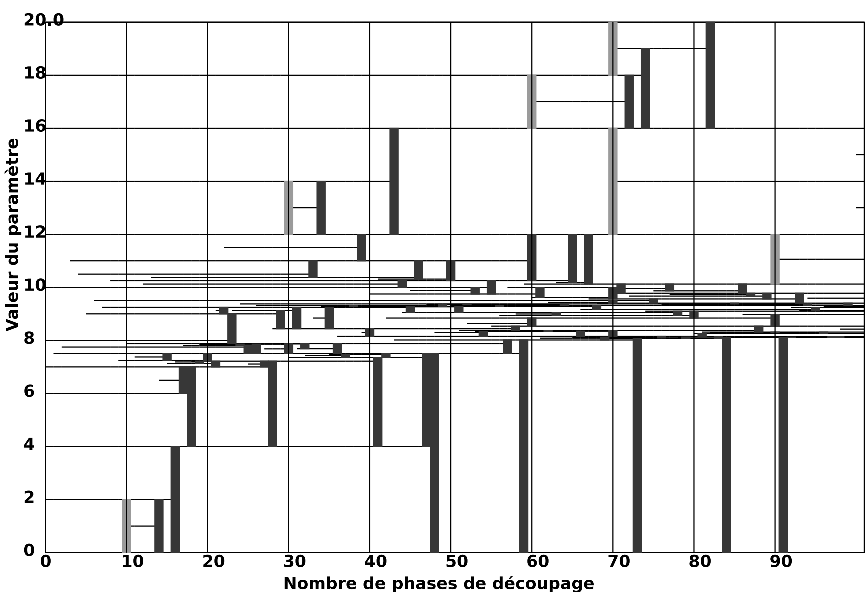


FIG. II.2.20 – *Évolution du découpage du paramètre* `speed` *avec la méthode découpage de paramètres en limitant à 25 intervalles par paramètre*

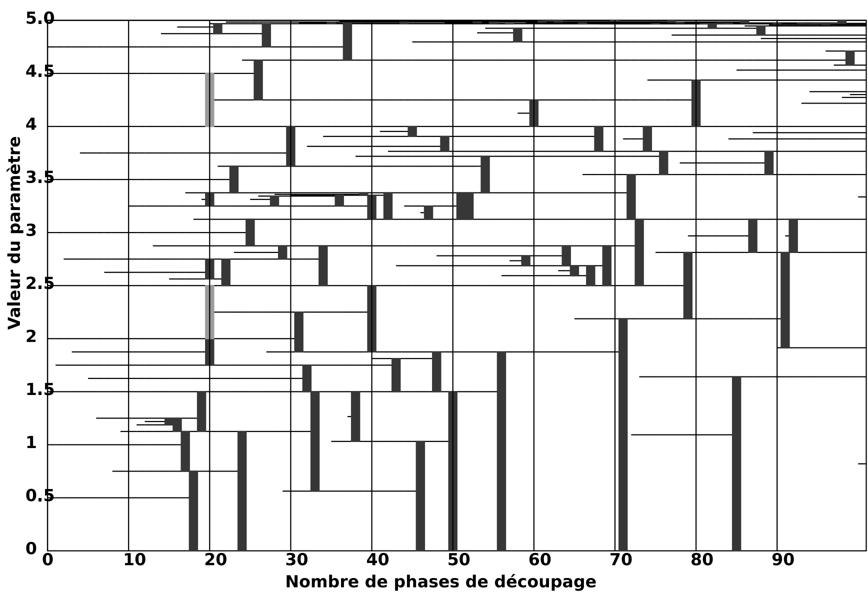


FIG. II.2.21 – *Évolution du découpage du paramètre artificiel avec la méthode découpage de paramètres en limitant à 25 intervalles par paramètre*

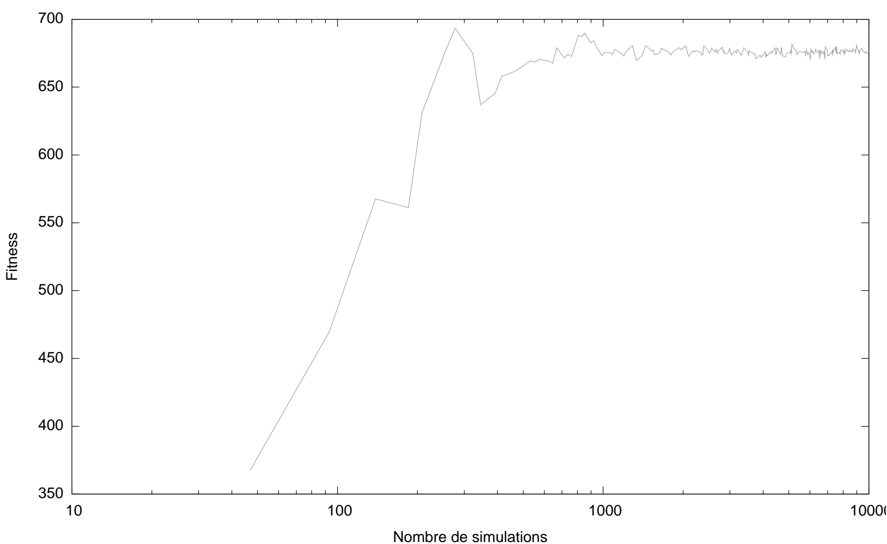


FIG. II.2.22 – *Évolution de la fitness en fonction du nombre de simulations avec la méthode découpage de paramètres en limitant à 100 intervalles par paramètres*

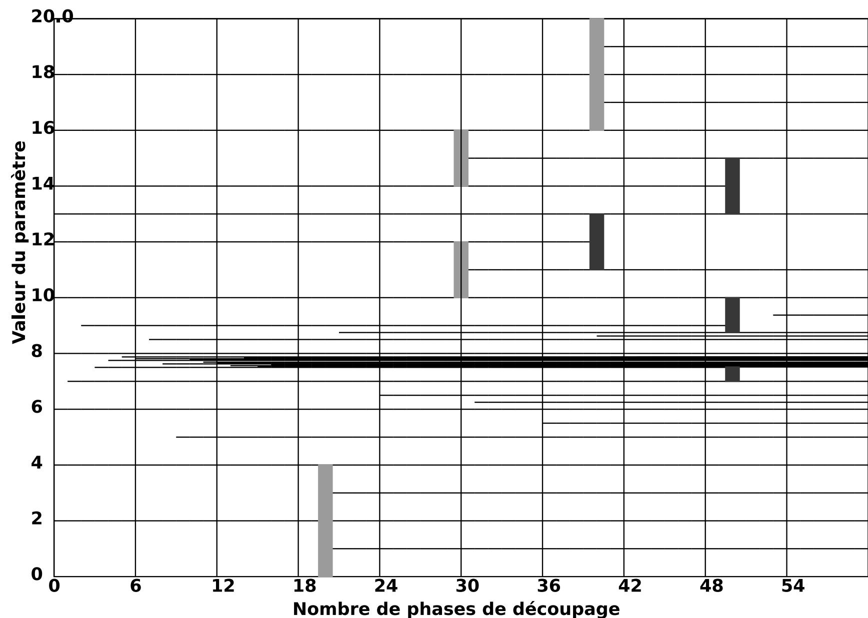


FIG. II.2.23 – *Évolution du découpage du paramètre* `speed` *avec la méthode découpage de paramètres en limitant à 100 intervalles par paramètre*

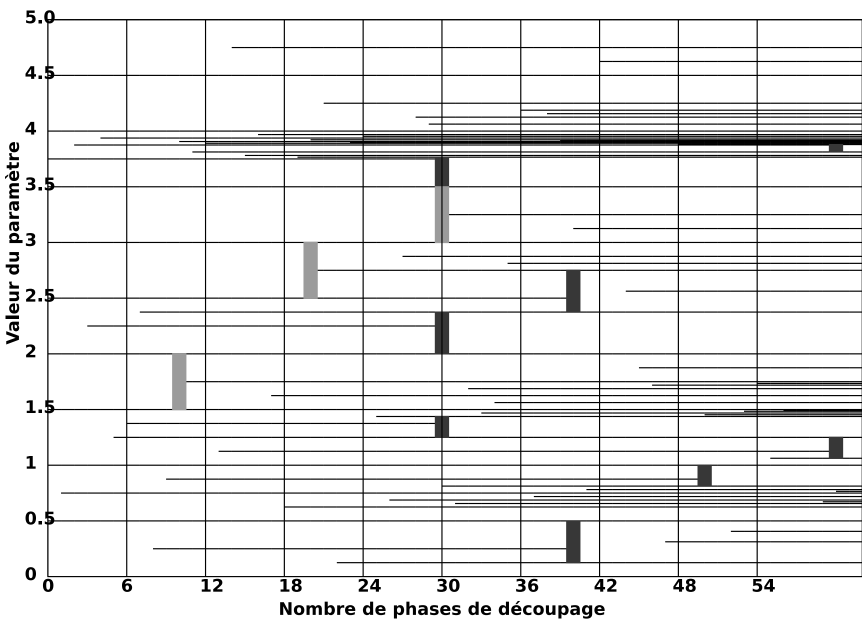
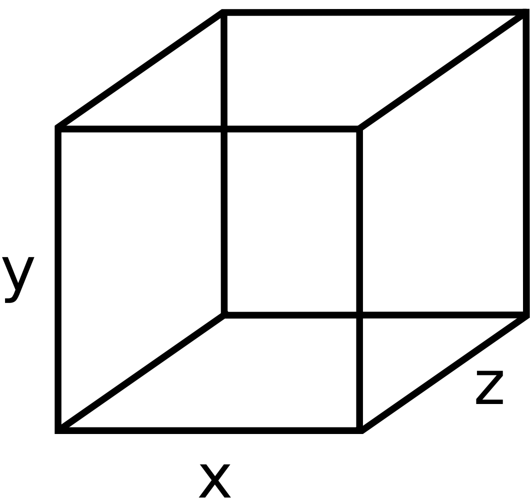
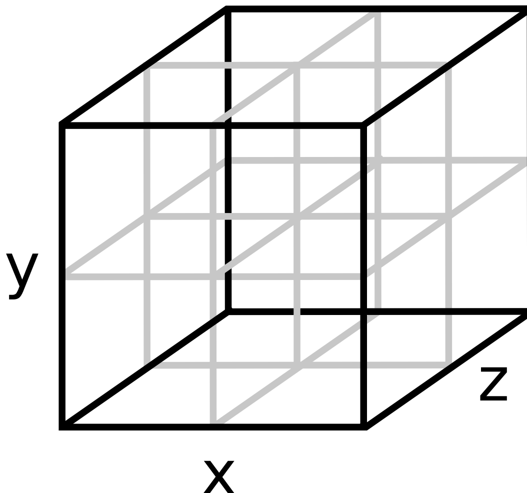


FIG. II.2.24 – *Évolution du découpage du paramètre artificiel avec la méthode découpage de paramètres en limitant à 100 intervalles par paramètre*

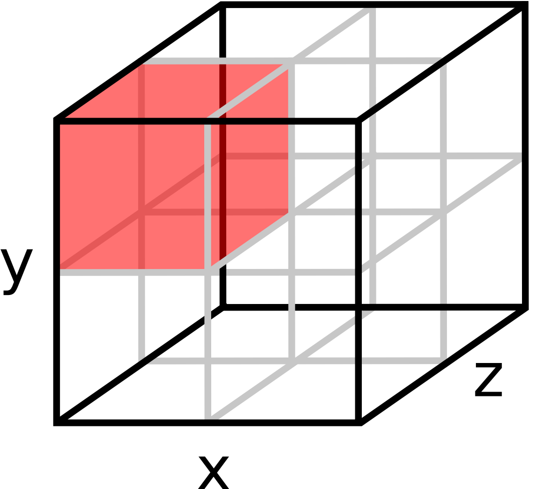
méthode de découpage en 2-étapes est appliquée, alors ce parallélépipède est découpé en quatre plus petit parallélépipèdes, et chaque parallélépipède va être évalué : une simulation est construite en paramétrant les agents avec les valeurs au centre du parallélépipède, elle est exécutée (voire plusieurs fois à cause de la stochasticité), et une *fitness* est calculée. Puis, le découpage recommence avec le parallélépipède ayant la meilleure *fitness*... La figure II.2.4.1 page 62 résume le principe. La méthode de découpage en n-étapes se déroule de la même manière que la méthode en 2-étapes sauf que l'espace est découpé en n au lieu de 2.



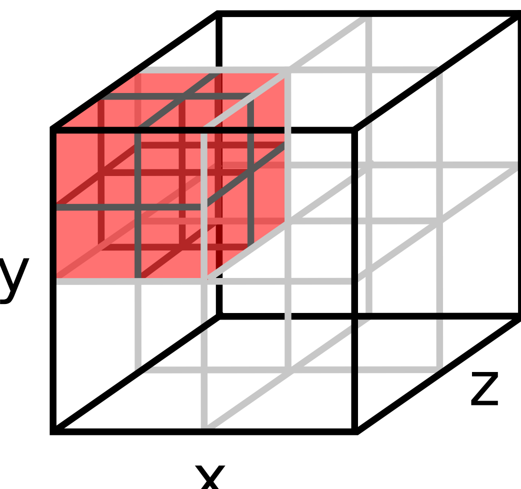
(a) Voici l'espace des paramètres initiale pour un modèle à trois paramètres.



(b) Chaque dimension de l'espace est découpé en 2 : on obtient 4 cubes. Chaque cube est évalué par une (ou des) simulation construite en paramétrant les agent avec les valeurs au centre du cube.



(c) Le cube ayant la meilleure *fitness* est sélectionné.



(d) Et l'algorithme recommence sur le cube sélectionné.

FIG. II.2.25 – *Principe de la méthode de découpage en 2-étapes*

Nous avons appliqué cette méthode sur le modèle « *Ants* » avec les mêmes conditions de test que précédemment, décrites à la sous-section II.1.2.1 page 35, en prenant $n = 3$ (découpage en trois sous-intervalles à chaque étape). La figure II.2.26 page suivante montre l'évolution de la *fitness* en fonction du nombre de simulations. La figure II.2.27 page ci-contre montre l'évolution du découpage du paramètre `speed`. La figure II.2.28 page 64 montre l'évolution du découpage du paramètre artificiel. Cette méthode nécessite un grand nombre de simulations pour obtenir des premiers résultats.

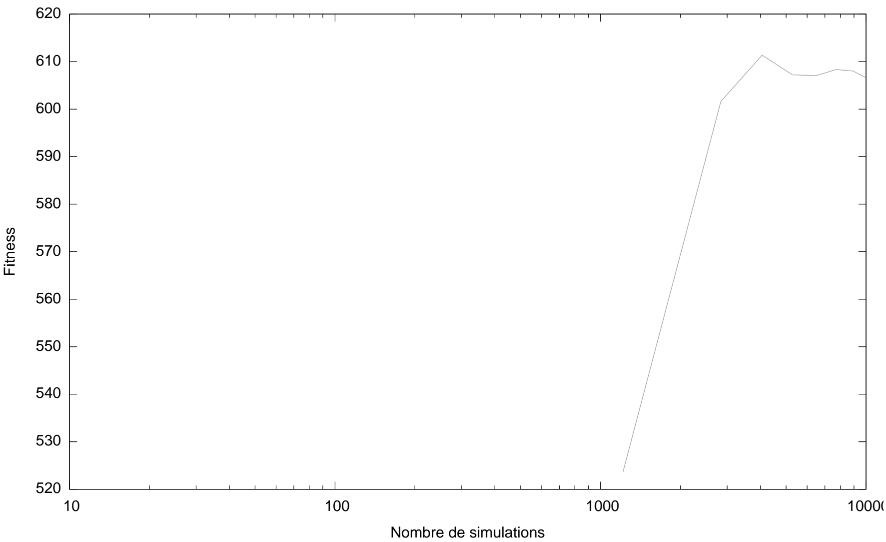


FIG. II.2.26 – *Évolution de la fitness en fonction du nombre de simulations avec la méthode découpage de paramètre en n-étapes dans le cas $n = 3$*

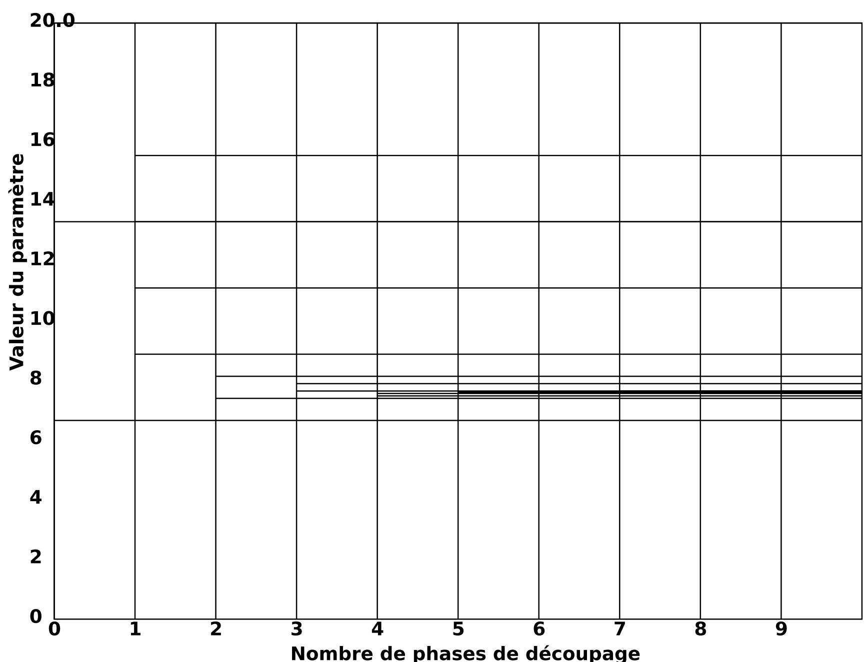


FIG. II.2.27 – *Évolution du découpage du paramètre* `speed` *avec la méthode découpage de paramètre en n-étapes dans le cas $n = 3$*

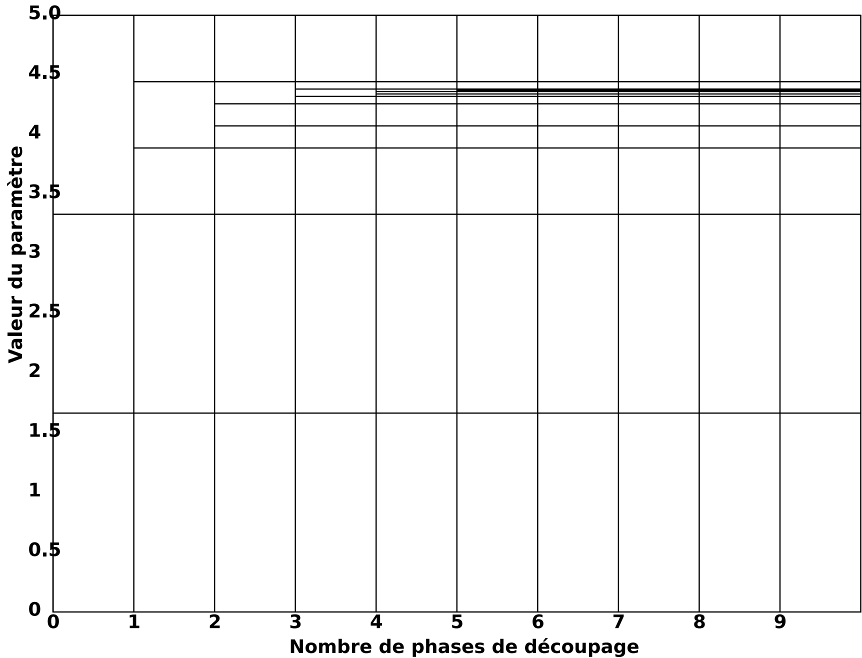


FIG. II.2.28 – *Évolution du découpage du paramètre* `foo` *avec la méthode découpage de paramètre en n-étapes dans le cas $n = 3$*

II.2.4.2 Comparaison

De nombreuses comparaisons ont été réalisées : nous ne présentons ici qu'un seul résultat de comparaison, illustrant de manière satisfaisante le comportement général de chaque méthode. La figure II.2.29 page 65 montre la comparaison des différentes méthodes présentées dans les précédentes sections. Les méthodes limitant le nombre d'intervalles ont de plus mauvais résultats en général, mais, dans quelques exemples précis, ont d'excellents résultats. La variante avec ajout d'aléatoire permet à la *fitness* de converger plus haut mais avec un démarrage plus lent par rapport à la méthode initiale. La méthode de découpage en n étapes a des résultats en net retrait par rapport aux autres méthodes : il faut un grand nombre de simulations pour obtenir des premiers résultats et la valeur de *fitness* obtenue est plus basse que les autres méthodes. La variante s'inspirant des algorithmes colonies de fourmis est la méthode la plus performante par rapport à toutes les autres méthodes : un démarrage plus rapide, une convergence plus haute, une meilleure stabilité.

II.2.5 Discussion

Nous avons présenté, dans ce chapitre, la méthode de découpage de paramètres et quelques variantes de cette méthode. L'idée de la création de cette méthode est d'avoir une méthode simple, et s'appuyant sur le modèle à base d'agents. Au final, la méthode de découpage de paramètres est une méthode relativement simple (même si la variante s'inspirant des algorithmes colonie de fourmis est un peu plus complexe). Elle utilise le fait qu'un modèle à base d'agents possède de nombreux agents identiques.

Les principaux avantages de cette méthode sont de fournir, en plus un jeu de paramètres solution, une espèce de « cartographie » de l'espace des paramètres. Cette cartographie permet de mesurer l'influence des paramètres. Si un paramètre a une faible influence, c'est-à-dire que quelque soit la valeur du paramètre, la valeur de la *fitness* change peu, alors le découpage obtenu serait relativement régulier, sans convergence. À l'inverse, si un paramètre a une forte influence, alors son découpage est irrégulier.

Un autre avantage est qu'avec peu de simulations, la valeur de *fitness* est très haute : cette caractéristique est très intéressante dans la cadre de simulations à base d'agents dont la durée d'exécution

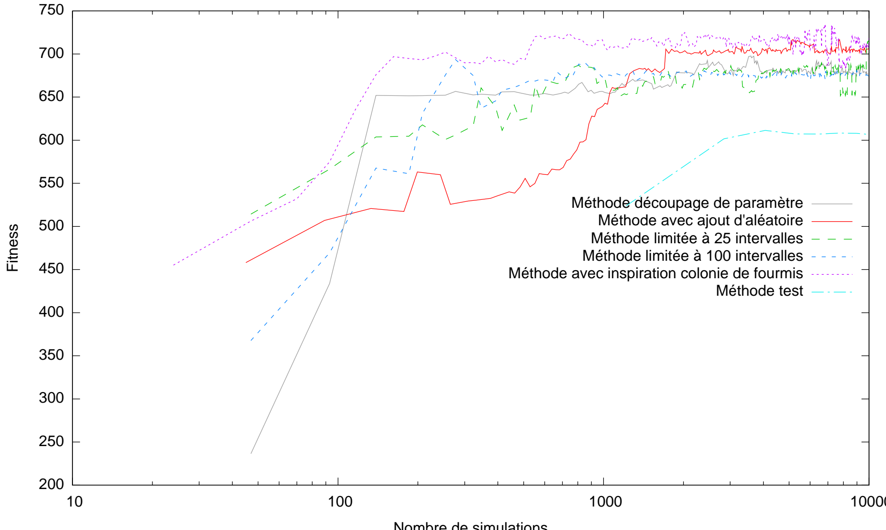


FIG. II.2.29 – *Comparaison des différentes méthodes de découpages de paramètres*

peut être longue.

Un autre point à ne pas oublier est qu'il est possible de faire du calcul distribué avec cette méthode : lors de la phase de génération de modèle, le programme génère plusieurs modèles, et chacun de ces modèles est simulé sur des processeurs différents.

Nous avons poser l'hypothèse que nous pouvions calibrer en considérant les paramètres indépendants : au vu des résultats, cette hypothèse semble réaliste.

Un autre point à ne pas oublier est que cette méthode peut s'appliquer dans le cas de modèle où les agents sont hétérogènes (voir l'annexe D page 137 pour voir un exemple).

Cependant, cette méthode, comme la précédente avec les algorithmes génétiques, requiert un grand nombre de simulations. Une idée est de modifier en ligne les agents durant la simulation afin de ne réaliser qu'une seule simulation.

II.3 Évolution d'agents

Pour explorer l'espace des paramètres, les précédentes méthodes multiplient le nombre de simulations à exécuter. Cette multiplication du nombre de simulations se traduit par un temps de calcul très important. L'idée de la méthode « évolution d'agents » est d'exécuter une seule simulation du modèle. Pour réaliser cette idée, au lieu de multiplier des simulations courtes, une seule simulation du modèle sera réalisée : cette simulation sera une simulation longue où les agents seront modifiés en ligne, c'est-à-dire qu'au cours de la simulation, les plus « mauvais » agents (en référence une *fitness* donnée) seront supprimés et de nouveaux agents seront ajoutés. Le but est d'obtenir au fur et à mesure une population d'agents qui s'améliore du point de vue de la *fitness*.

Cette démarche d'évolution d'agents peut s'apparenter à de l'apprentissage par renforcement, méthode décrite à la section I.3.5 page 26. Mais, dans le cadre de la thèse, les agents ont un comportement fixé par l'utilisateur, dont les seuls ajustements concernent les des paramètres : ils n'ont pas la possibilité d'évaluer la conséquence de leurs actions au niveau des récompenses. Dans l'apprentissage par renforcement, l'agent choisit sa décision en fonction de l'environnement et des récompenses espérées. Pour pouvoir utiliser l'apprentissage par renforcement, il faudrait complexifier le fonctionnement des agents, ce qui est en contradiction avec les objectifs initiaux : à partir d'un modèle à base d'agents, l'objectif consiste à calibrer le modèle sans avoir à le reconstruire.

Cette méthode d'évolution d'agents pourrait ressembler à la programmation génétique, méthode décrite à la section I.3.2 page 21 : cela revient en effet à appliquer une « sorte » d'algorithme génétique, non plus sur des paramètres, mais sur le comportement des agents. Il serait même possible d'aller plus loin en appliquant la programmation génétique, non plus sur un comportement générique des agents, mais sur le comportement individuel de chaque agent. À la différence de la méthode proposée dans ce chapitre, avec une telle méthode, il serait possible d'obtenir des comportements des agents « non réels » : par exemple, dans le modèle du fourragement par colonie de fourmis, il serait possible d'imaginer obtenir un modèle où les fourmis se passeraient la nourriture entre elles afin de la ramener à la fourmilière, au lieu d'effectuer des allers et retours entre les sources de nourriture et la fourmilière.

II.3.1 Description détaillée de la méthode

La méthode « évolution d'agents » se déroule en plusieurs étapes : une phase de simulation, une phase de récompense, et une phase de sélection. La figure II.3.1 page suivante montre un schéma général de la méthode : il y a d'abord une phase de simulation, puis les agents sont récompensés, et une phase de sélection se déroule finalement avec des suppressions et des ajouts d'agents. Cependant il existe différentes manières de concevoir une telle méthode. Une première idée est de faire un parallèle avec les algorithmes génétiques : au lieu d'avoir une population de modèles paramétrés, la méthode a une population d'agents au sein d'un modèle. Cette population d'agent est modifiée lors de phases de sélection espacées tous les n pas de simulation : c'est au cours de ces phases, que les plus mauvais agents sont supprimés et remplacés par de nouveaux agents. Une telle méthode pourrait se nommer l'*évolution par générations*. Une deuxième vision d'un tel algorithme est une évolution beaucoup plus continue : chaque agent aurait des points de vie, et à chaque action élémentaire, il perdrait ou gagnerait des points de vie. Quand son nombre de points de vie devient nul ou négatif, il est supprimé. Une telle méthode pourrait se nommer l'*évolution en continu*.

II.3.1.1 Évolution par générations

La méthode « évolution d'agent » peut se découper en deux variantes différentes : cette sous-section présente une variante de l'*évolution d'agent par générations*. La figure II.3.2 page 69 montre le principe de fonctionnement de cette variante. Au départ, un modèle est initialisé avec des agents dont les valeurs de paramètres sont choisies de manière aléatoire. Puis, le modèle ainsi paramétré est

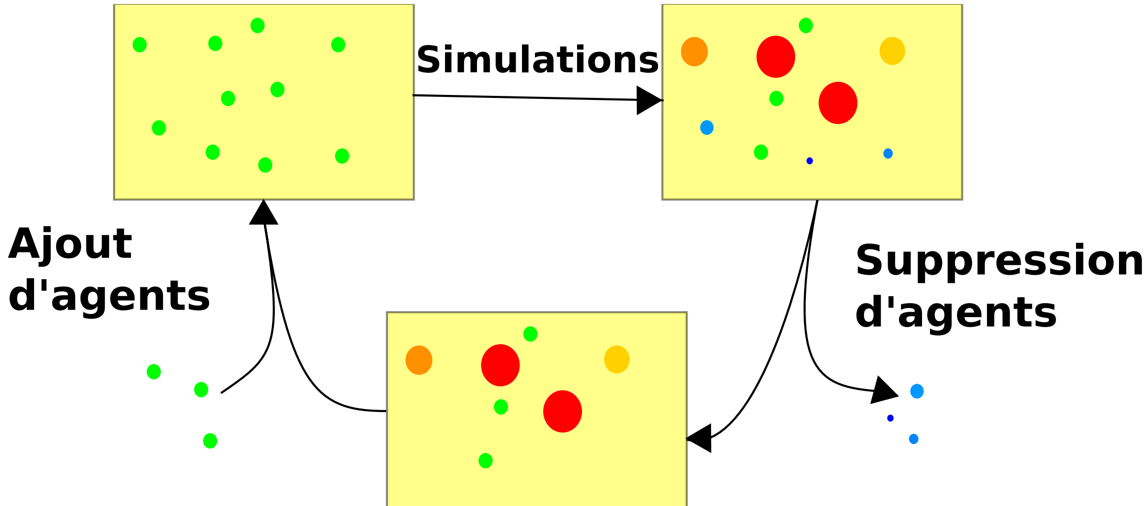


FIG. II.3.1 – *Schéma du principe de la méthode d'évolution d'agents*

simulé. Au cours de la simulation ou après la simulation, les agents reçoivent des récompenses. Ces récompenses dépendent du résultat d'une fonction de *fitness* calculée individuellement sur chaque agent. À partir de ces récompenses, les agents sont classés : les plus « mauvais » agents pour la *fitness* sont supprimés. De nouveaux agents, initialisés aléatoirement sont ajoutés. Avec cette nouvelle population, la phase de simulation recommence.

Général

Soit une simulation multi-agent $\mathcal{S}$ avec n agents a_i avec $i \in [\![0,n]\!]$ ($n \in \mathbb{N}^*$).
Chaque agent i possède un certain nombre de paramètres k^i ($k^i \in \mathbb{N}^*$) :

$$\forall i \in [\![0,n]\!] \; a_i = \left\{P_j^i | j \in [\![0,n]\!]\right\}$$

Supposons que pour chaque agent, une *fitness* $\mathcal{F}_i$ soit disponible.

Initialisation des agents

Les agents sont initialisés aléatoirement.

Simulation

Durant la phase de simulation, le modèle est simulé un certain nombre de pas de simulation dépendant du modèle multi-agent choisi. Il faut que le nombre de pas de simulations soit suffisant pour que la *fitness* de chaque agent soit satisfaisante c'est-à-dire que le comportement de l'agent ait duré pendant une durée suffisamment longue pour permettre d'évaluer une *fitness* pertinente. Par exemple, si nous reprenons le fourragement par les fourmis, si la quantité de nourriture ramenée à la fourmilière est très faible (quelques unités), la différence entre les « bons » agents et les « mauvais » agents ne sera pas significative. Ceci dépend du modèle multi-agent et de la fonction de *fitness* choisie. A contrario, si le nombre de pas de simulation est trop élevé, la simulation se poursuivra alors que la *fitness* des agents est déjà évaluée de manière satisfaisante, entraînant ainsi une perte de temps.

Récompense

Durant cette phase, les agents sont récompensés en relation avec une fonction de *fitness* individuelle.

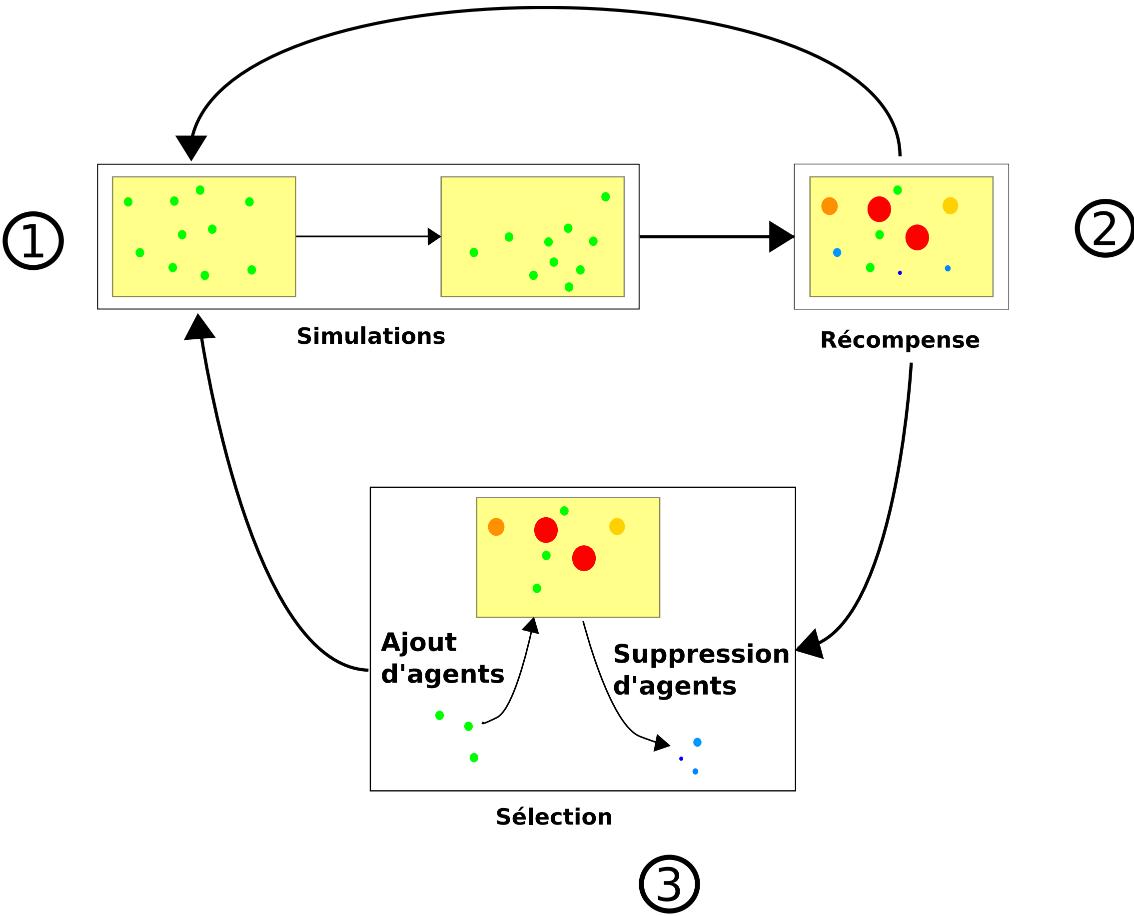


FIG. II.3.2 – *Schéma du principe de la méthode d'évolution d'agents par générations*

Sélection

Cette phase correspond à l'étape où des agents sont supprimés, et où de nouveaux agents sont ajoutés. Pour réaliser ce processus, les agents sont triés en fonction de leur *fitness*. Les plus « mauvais » agents (c'est-à-dire ceux qui ont la *fitness* la plus basse) sont supprimés. De nouveaux agents sont ajoutés dont les paramètres sont initialisés aléatoirement.

Le choix du nombre d'agents à supprimer dépend du modèle (et plus particulièrement du type de modèle). Si le modèle est un modèle où il n'y a pas de création ou de suppression d'agents, une proportion de 5% à 10% d'agents remplacés à chaque génération semble un choix intéressant, suite à différentes expérimentations. Si un grand nombre d'agents est remplacé à chaque génération, une grande partie de la population sera renouvelée à chaque sélection et donc, il y aura un problème de stabilité pour la convergence (et peut-être même la convergence sera compromise). La figure II.3.3 montre l'évolution de la *fitness* en fonction du nombre de générations et du taux de remplacement (ces résultats ont été obtenus par des pseudo-simulations à base de fonctions mathématiques, décrites à l'annexe E page 143, plus particulièrement à la section E.4 page 148 pour la fonction mathématique). De même, si peu d'agents sont supprimés par phase de sélection, alors il y aura une plus grande stabilité des résultats mais la durée de l'algorithme augmentera. Donc il faut trouver un juste milieu entre stabilité et rapidité.

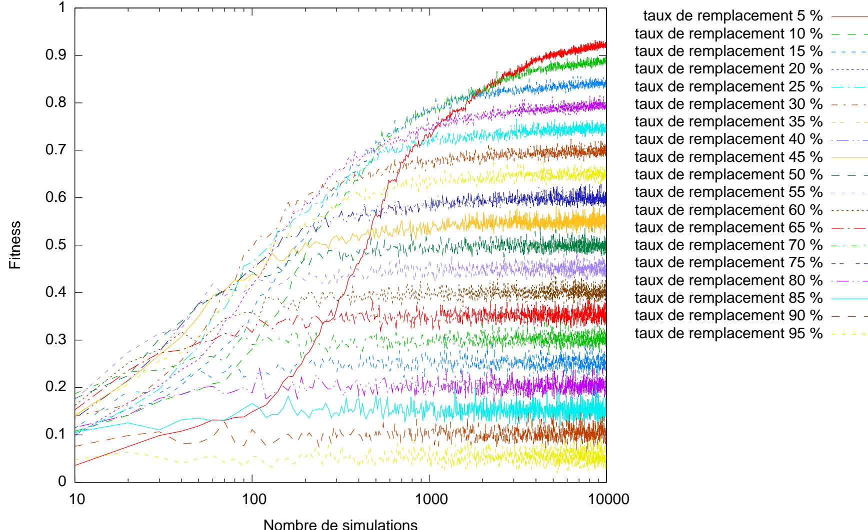


FIG. II.3.3 – *Évolution de la fitness en fonction du nombre de générations et du taux de remplacement*

Le choix d'ajouter des agents initialisés de façon aléatoire est pour éviter une convergence trop rapide, et surtout éviter une convergence vers des extremums locaux.

II.3.1.2 Évolution en continu

Après avoir présenté la variante d'*évolution par générations* dans la section précédente, nous décrivons la deuxième variante dans cette sous-section, l'*évolution d'agents en continu*. Le schéma II.3.4 page suivante résume le principe de fonctionnement de cette méthode. D'abord, un pas de simulation est exécuté, puis les points de vie des agents sont modifiés, et enfin, les agents, dont les points de vie deviennent nulles ou négatifs, sont supprimés.

Général

Soit une simulation multi-agent $\mathcal{S}$ avec n agents ($n \in \mathbb{N}^*$).

Chaque agent a_i avec $i \in [\![0,n]\!]$ possède des points de vie p_i ($p_i \in \mathbb{Z}$).

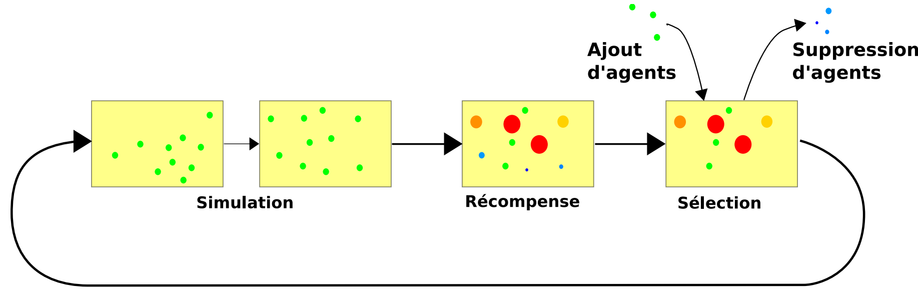


FIG. II.3.4 – *Schéma du principe de la méthode d'évolution d'agents en continu*

Initialisation des agents

Les agents sont initialisés avec des valeurs de paramètres choisies aléatoirement, avec un nombre de points de vie identiques pour tous les agents.

Simulation

Le modèle avec ses agents est simulé un pas.

Récompense

C'est durant cette phase que les agents sont récompensés : en fonction des actions réalisées par l'agent, il reçoit ou perd points de vie. Par exemple, pour le fourragement par une colonie de fourmi, chaque agent fourmi perd des points de vie à chaque déplacement, et ne gagne des points qu'il ramène une unité de nourriture à la fourmilière.

Sélection

La phase de sélection correspond à la phase où les agents sont supprimés ou ajoutés. Lorsque que le nombre de points de vie d'un agent devient nul ou négatif, l'agent est supprimé. Et à chaque agent supprimé est ajouté un nouvel agent dont les paramètres sont choisis aléatoirement Comme précédemment, l'initialisation est réalisé de manière aléatoire pour éviter la convergence vers des extremums locaux.

II.3.1.3 Problématiques de conception

Dans cette sous-section, nous allons nous intéresser à quelques points particuliers de la méthode évolution d'agents.

Choix de la fonction de *fitness*

Une des plus grandes difficultés de cette méthode est le choix de la fonction de *fitness*. C'est un des choix essentiels pour la méthode. Faire un mauvais choix pour la fonction de *fitness* entraîne un mauvais résultat de la méthode et donc un mauvais choix des paramètres. Généralement il est relativement facile de choisir une fonction de *fitness* du point de vue général du modèle : par exemple, pour le modèle « *Ants* », un bon modèle pourrait être un modèle qui ramène toute la nourriture au nid le plus rapidement possible. Mais du point de vue local, il est beaucoup plus dur de déterminer une *fitness* intéressante : par exemple, pour le modèle « *Ants* », la *fitness* correspondant au « temps moyen pour ramener la nourriture » est une *fitness* possible au niveau d'un agent, mais est-ce une *fitness* intéressante qui va mener à l'obtention d'un paramétrage intéressant au niveau global ? Est-ce qu'à partir de cette *fitness* locale, la méthode évolution d'agent va obtenir les mêmes résultats qu'avec la méthode algorithme génétique avec la *fitness* globale ?

En fonction du choix de la variante de la méthode, la manière de choisir la *fitness* est identique. La plus grande difficulté est de concevoir une fonction de *fitness* dont les récompenses peuvent être

individualisées au niveau local. Il est possible de concevoir la fonction de *fitness* de différentes façons possibles. Voici quelques possibilités de choix de fonction de *fitness*:

- locale à un agent individuel : lorsqu'un agent réalise une action, l'agent est récompensé individuellement. Par exemple, dans le modèle « *Ants* », une fourmi qui ramène une unité de nourriture est récompensée.
- locale à un groupe d'agents : par exemple, une récompense est attribuée à un même type d'agent ou/et attribuée à un groupe d'agents qui ont coopéré pour réaliser une tâche. Par exemple, pour le fourragement, quand une fourmi ramène de la nourriture à la fourmilière, nous récompensons toutes les fourmis qui ont « participé » à cette récupération de nourriture, c'est-à-dire l'ensemble des fourmis qui ont déposé des phéromones sur le chemin entre la fourmilière et la zone où la nourriture a été prise.
- spatiale : une récompense est « attribuée » à une zone précise de l'environnement : tous les agents se trouvant sur cette zone de l'environnement sont récompensés. Par exemple, dans le modèle « *Ants* », à chaque unité de nourriture ramenée, les agents fourmis se trouvant sur des zones de régions de l'environnement contenant un certain seuil de phéromone sont récompensées.

Il est possible d'imaginer encore d'autres variantes en combinant les précédentes, par exemple une *fitness* spatiale propre à un groupe d'agents.

Après avoir déterminé une *fitness* individuelle, il est intéressant de calculer une *fitness* globale au modèle. Cette *fitness* globale permettra de visualiser, de quantifier l'évolution de l'algorithme. Elle est surtout intéressante dans l'exploitation des résultats (voir la section II.3.1.4 page ci-contre). En fonction de la variante de la méthode choisie, deux cas se présentent. En ce qui concerne le cas de l'*évolution par générations*, une façon simple d'obtenir cette *fitness* globale est de calculer la moyenne des *fitness* individuelles des « meilleurs » agents (le nombre dépend du nombre d'agents remplacés à chaque génération : une possibilité est de choisir environ de 50% à 75% des « meilleures » agents lors du calcul de la moyenne. Ce choix semble conduire à des résultats intéressants.) :

$$\text{Fitness Globale} = \frac{\displaystyle\sum_{e_i \in \mathcal{P}_{\text{meilleur}}(Agents)} \mathcal{F}_{e_i}}{|\mathcal{P}_{\text{meilleur}}(Agents)|}$$

En ce qui concerne le cas de l'*évolution en continu*, le calcul de la *fitness* globale est un peu plus compliqué : la *fitness* globale pourrait être la somme des *fitness* (c'est-à-dire des points de vie) sur les x derniers pas de simulations.

Critère d'arrêt

Un paramètre important de la méthode est le nombre de générations de l'algorithme avant de calculer le résultat. La question qui se pose est de savoir quand la méthode évolution doit s'arrêter. Une première idée est de choisir un nombre fixe de générations d'évolution. Tout au long de la thèse, c'est ce choix qui a été fait : étant donné un modèle multi-agent, l'objectif est de rechercher les « bons » paramètres en un minimum de simulations. Ainsi, l'expérimentateur choisit le nombre de simulations (c'est-à-dire le nombre de générations de l'algorithme) (par exemple, dont le temps de calcul correspond a une journée).

Une deuxième idée serait de s'intéresser à la convergence de la *fitness* individuelle du meilleur agent (ou des meilleurs) au fil des générations. Quand cette *fitness* converge, alors il est possible d'arrêter la méthode évolution d'agent. C'est la même idée qui a été développée dans la méthode découpage de paramètres dans la sous-section II.2.1.6 page 49. Il est possible de faire de même pour la *fitness* globale : quand la *fitness* globale converge, alors il est possible d'arrêter l'évolution des agents.

Un autre critère d'arrêt serait l'homogénéisation de la population d'agent. Quand la population devient homogène, à chaque sélection les plus mauvais agents sont en majorité ceux qui ont été créés à la précédente sélection pour l'*évolution d'agent par générations*, ou les agents détruits sont ceux qui viennent juste d'être créés, la méthode a « convergé », et donc on peut arrêter de faire évoluer les agents.

Ajout et suppression d'agents

L'étape de suppression d'un agent est une phase délicate. En fonction des situations, supprimer un agent peut poser des difficultés, qui peuvent être, par exemple :

- interactions entre agents : par exemple, dans le modèle « *Ants* », s'il est autorisé l'échange de nourriture entre les fourmis, un agent à supprimer peut se trouver en train d'interagir avec un autre.
- interactions entre agents et environnement : par exemple, dans le modèle « *Ants* », s'il est autorisé l'échange de nourriture entre les fourmis, un agent à supprimer peut se trouver en train d'interagir avec un autre.

Le problème survient quand un agent à supprimer interagit avec l'environnement ou avec d'autres agents. Pour supprimer un tel agent, il faut supprimer une à une toutes les interactions de l'agent en faisant des choix : par exemple, dans le modèle « *Ants* », la nourriture portée par l'agent fourmi est renvoyée dans la zone où elle a été prélevée.

L'ajout d'agents est beaucoup moins problématique. Le seul point auquel il faut faire attention est leur emplacement.

Il est possible d'avoir aussi des difficultés avec l'environnement : par exemple, des ressources de l'environnement peuvent disparaître. Cela peut se voir dans le modèle « *Ants* » où les zones de nourriture disparaissent au fur et à mesure de leur exploitation : il faut donc les réapprovisionner durant la simulation.

II.3.1.4 Calcul des solutions

Un des aspects les plus importants de la méthode est le calcul du paramétrage solution. Une première approche est de choisir le paramétrage solution comme similaire au paramétrage du meilleur agent. Cependant, du fait de la stochasticité de la simulation, la valeur de la *fitness* du meilleur agent peut être surévaluée par rapport aux autres agents. Ainsi, cette approche n'apparaît pas comme un choix raisonnable. Une proposition alternative est de considérer le paramétrage solution comme le paramétrage des « meilleurs » agents (moyenne et écart-type). Mais le terme « meilleurs » agents peut poser problème sur sa signification : si la population finale est homogène, le choix des « meilleurs » agents est simple. Par contre si la population n'est pas homogène, il faudra distinguer des groupes, donc distinguer plusieurs paramétrages solution différents.

De plus il se peut que pour certains paramètres, il n'y ait pas de convergence vers une valeur. Ce problème est similaire à celui rencontré avec la méthode découpage de paramètres pour la non-convergence du découpage de certains paramètres. De la même manière, cette non-convergence est le signe d'un paramètre avec une influence faible sur la simulation.

À la fin de la méthode, une *fitness* globale finale peut être calculée. Le principe de calcul est de simuler le modèle avec des agents paramétrés avec le paramétrage solution.

II.3.2 Résultats

Cette section présente trois résultats d'application de la méthode évolution d'agents.

II.3.2.1 Modèle « *Traffic 2 Lanes* »

Dans cet exemple, nous allons nous intéresser à calibrer le modèle « *Traffic 2 Lanes* » de la plateforme `NetLogo`. Ce modèle est décrit en détail à la section B.2 page 128 représente une autoroute à deux voies où circulent des agents voitures. Ces voitures peuvent accélérer, freiner ou changer de voie. Les paramètres dont l'espace va être exploré sont :

- `speed-up`,
- `slow-down`,
- `speed-limit`,
- `look-ahead`.

La fonction de *fitness* est la somme sur tous les pas de simulation des vitesses instantanées supérieures à 1. La simulation compte 54 agents.

La figure II.3.5 montre l'évolution de la *fitness* du meilleur agent au fur et à mesure des pas de simulations : en très peu de pas de simulation, le modèle a été calibré. Les agents solutions sont des voitures qui ont une accélération et un freinage faibles, mais qui ont une vitesse limite haute, et qui regardent loin devant eux.

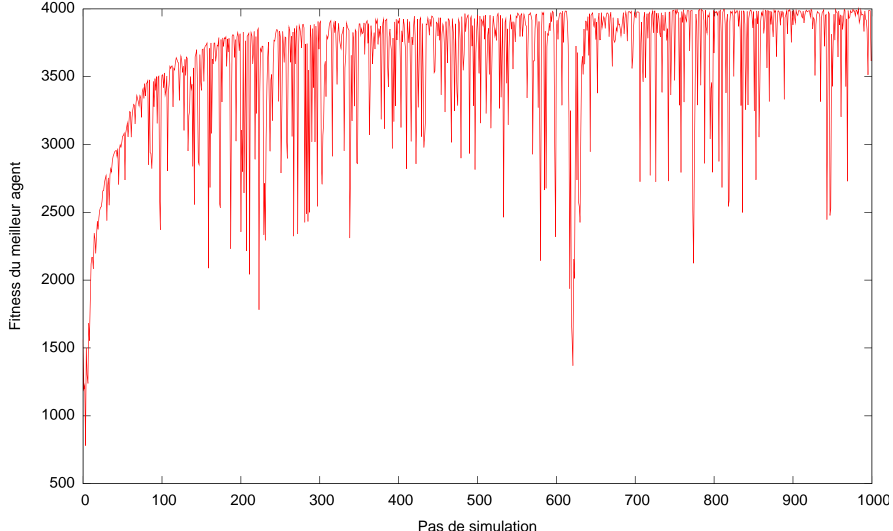


FIG. II.3.5 – *Évolution de la fitness en fonction du nombre de pas de simulation pour le modèle « Traffic 2 Lanes » par la méthode évolution d'agents*

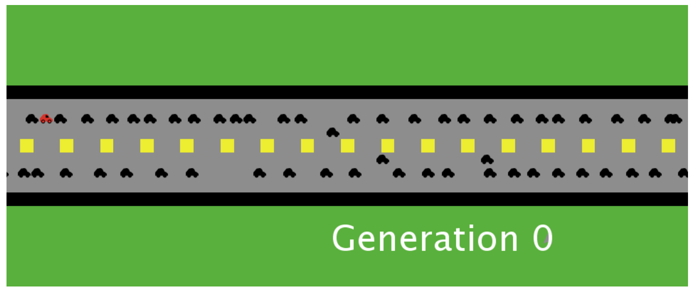


FIG. II.3.6 – *Le modèle « Traffic 2 Lanes » optimisé par la méthode évolution d'agents*

II.3.2.2 Modèle « *Divide the cake* »

Nous nous intéressons ensuite dans cet exemple à calibrer le modèle « *Divide the cake* » de la plate-forme `NetLogo`. Ce modèle évolutionnaire est composé de trois types d'agents, caractérisés par un appétit, se disputant une ressource commune. La survie d'un agent dépend des agents situés sur la même zone de l'environnement que lui. Ce modèle est décrit plus en détail à la section B.4 page 130. La méthode va seulement jouer sur le paramètre `appetite`. Initialement, il y 200 agents. Le but du calibrage est de maximiser le nombre d'agents présents sur l'environnement. Pour cela, la *fitness* sera une *fitness* collective à un groupe d'agents : nombre d'agents identiques (ayant un même ancêtre commun).

La figure II.3.7 page suivante montre l'évolution des populations d'agents au cours de l'algorithme. Avec cette *fitness*, ce sont les agents égoïstes qui dominent l'environnement. Or, l'objectif de cette exploration de paramètres était de trouver un jeu de paramètres pour optimiser le nombre d'agents présents. Intuitivement et expérimentalement, ce n'est pas une population composée d'agents égoïstes qui maximise le nombre d'agents, mais une population composée majoritairement d'agents équitables. Ainsi, cet exemple illustre le problème du choix de la fonction de *fitness*

au niveau local : un mauvais choix au niveau local peut entraîner des résultats sous-optimaux au niveau global.

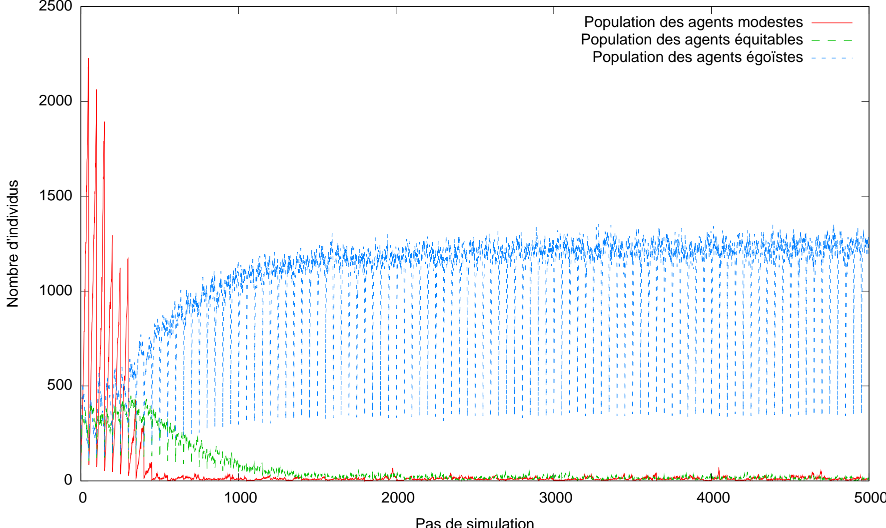


FIG. II.3.7 – *Évolution des populations d'agents en fonction du nombre de pas de simulation pour le modèle « Divide the cake » par la méthode évolution d'agents*

II.3.2.3 Modèle « *Ants* »

Les deux exemples précédents étaient des exemples bien choisis pour la méthode évolution d'agents : de nombreux agents, simulation qui n'a pas de fin, ajout et suppression d'agents faciles. Pour pouvoir comparer cette méthode avec les autres méthodes, nous utilisons le modèle « *Ants* » dans les mêmes conditions de test, décrites à la sous-section II.1.2.1 page 35. La méthode *évolution d'agent par générations* va être employée : le modèle est simulé durant 500 pas de simulation. Durant la simulation, les agents sont récompensés proportionnellement au nombre d'unités de nourriture qu'ils rapportent à la fourmilière. À la fin des 500 pas de simulation, une phase de sélection est exécutée : les plus « mauvais » agents disparaissent et de nouveaux agents sont ajoutés. Puis recommence la phase de simulation. L'algorithme s'arrête au bout de $10\ 000 \times 500$ pas de simulation. La *fitness* globale est la somme des unités de nourriture ramenées à la fourmilière durant un cycle de simulation.

La figure II.3.9 montre l'évolution de la *fitness* globale au fur et à mesure des simulations. Très vite, la *fitness* converge, mais à des valeurs moins hautes qu'avec les autres méthodes. De plus, elle est moins stable en valeur. Par ailleurs cette méthode a été un peu « dénaturée » pour avoir un exemple comparable aux autres méthodes. L'idée de base de cette méthode est d'avoir une seule simulation continue. Ici, pour l'adapter au modèle, la méthode exécute 500 pas de simulation, puis les plus « mauvais » agents sont supprimés et de nouveaux agents sont ajoutés, ensuite l'environnent est réinitialisé : suppression des phéromones et initialisation des sources de nourriture. Dans ce cas-là, nous perdons l'idée d'une seule simulation continue et nous avons plutôt une juxtaposition de simulations. Il serait sûrement possible d'avoir de meilleurs résultats en changeant les conditions de test.

La figure II.3.10 page ci-contre montre l'évolution des valeurs du paramètre `speed`, et de l'écart-type. Assez vite, il y a convergence des valeurs avec une homogénéisation de la population d'agents. La figure II.3.11 page suivante montre l'évolution des valeurs du paramètre artificiel, qui n'a aucune influence sur le modèle, et de son écart-type. Il n'y a pas de convergence, et la population d'agents pour ce paramètre ne devient pas homogène. Au niveau des résultats que montre le tableau II.3.1 page ci-contre, les conclusions sont relativement similaires aux conclusions obtenues avec les résultats des précédentes méthodes, mais avec des valeurs de vitesses et de perceptions plus faibles.

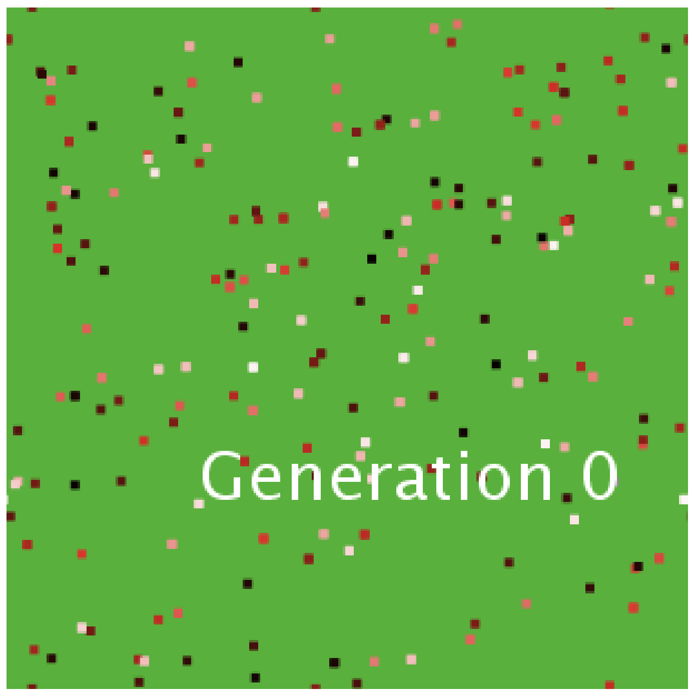


FIG. II.3.8 – *Le modèle « Divide the cake » optimisé par la méthode évolution d'agents*

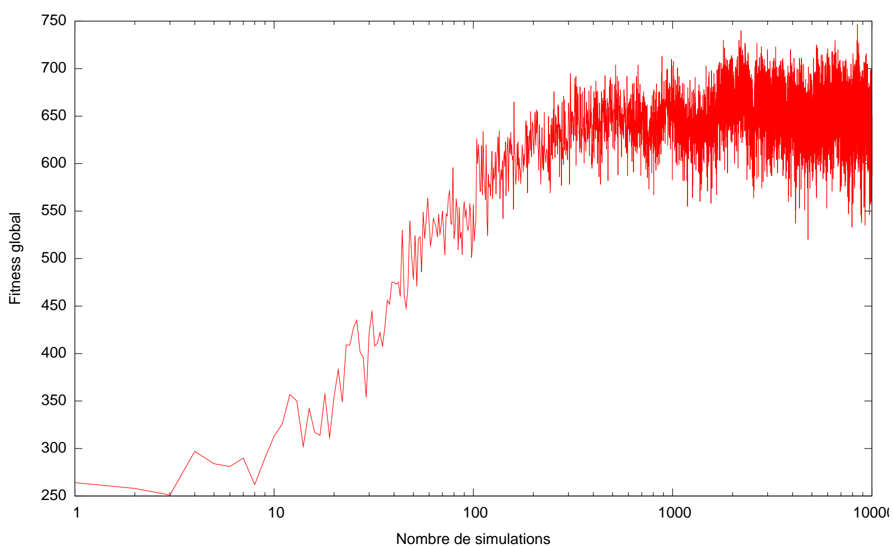


FIG. II.3.9 – *Évolution de la fitness globale en fonction du nombre de simulations pour le modèle « Ants » par la méthode évolution d'agents*

nom du paramètre	valeur
`speed`	6,44
`patch_ahead`	7,730
`angle_vision`	239.257
`drop_size`	127.912

TAB. II.3.1 – *Paramétrage solution obtenu avec la méthode évolution d'agents*

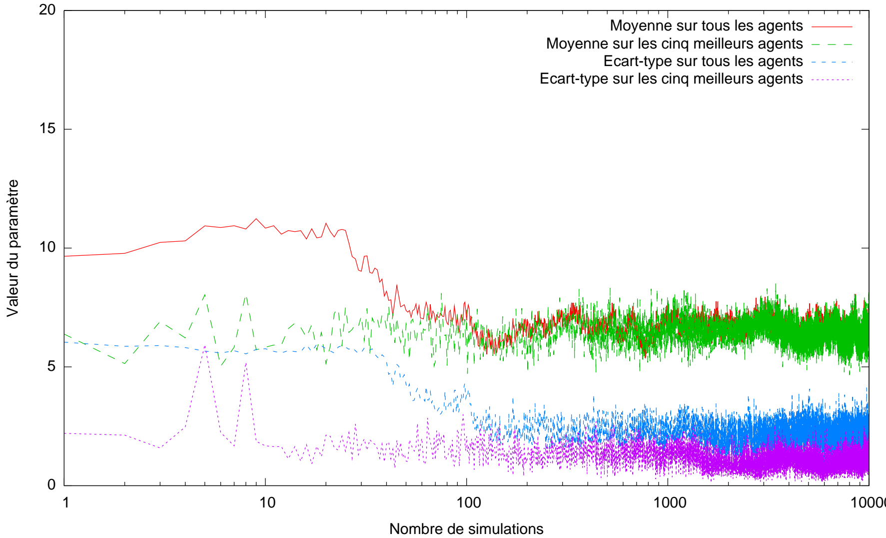


FIG. II.3.10 – *Évolution des valeurs du paramètre* `speed` *en fonction du nombre de simulations par la méthode évolution d'agents*

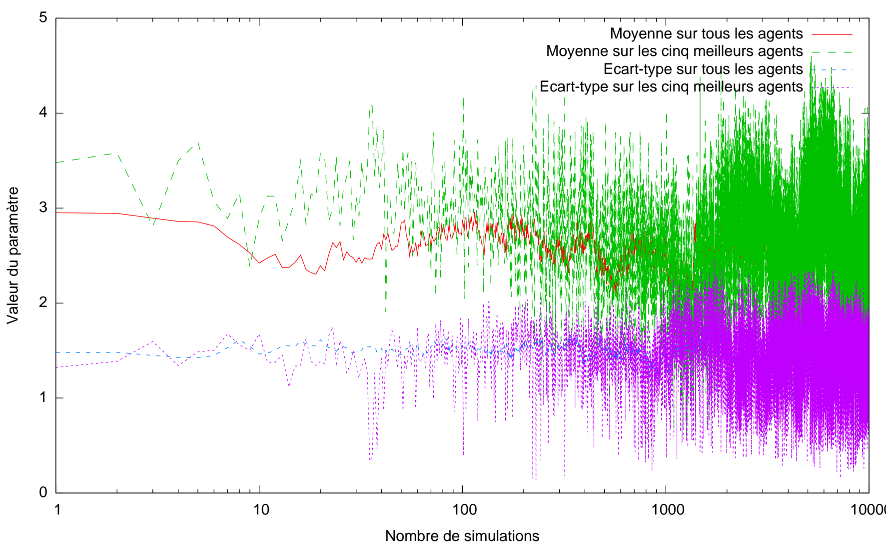


FIG. II.3.11 – *Évolution des valeurs du paramètre artificiel en fonction du nombre de simulations par la méthode évolution d'agents*

II.3.3 Discussion

Nous avons décrit dans ce chapitre la méthode d'évolution d'agents. L'idée de cette méthode est de modifier les agents au cours de la simulation. Cette méthode a de très bons résultats, en peu de pas de simulation, et est peu sensible à la stochasticité.

Cependant cette méthode présente beaucoup d'inconvénients. D'abord, cette méthode ne fonctionne que sur les paramètres internes aux agents, non sur les paramètres globaux au modèle. Il serait possible d'intégrer la gestion de ces paramètres globaux à cette méthode (comme, par exemple, au lieu de faire évoluer une simulation, en faire évoluer plusieurs simulations, avec des phases de partage d'agents...) ; mais il y aurait la perte de l'intérêt principale de cette méthode d'effectuer une seule simulation. De même, il sera très difficile de réaliser du calcul distribué avec cette méthode : l'efficacité de cette méthode en terme de nombre de pas de simulation à exécuter sur un seul processeur n'est pas forcément si rapide en terme de temps par rapport aux deux précédentes méthodes, qui sont utilisables en calcul distribué. De plus, du point de vue de la programmation, cette méthode est plus complexe : dans cette méthode, il faut pouvoir supprimer des agents précis durant la simulation, modifier l'environnement et donc il faut ajouter et modifier dans le modèle un grand nombre de fonctions.

La difficulté la plus grande de cette méthode est le choix de la fonction de *fitness*. Généralement les objectifs d'un utilisateur sont exprimés d'un point de vue global : le modèle doit ressembler à des données globales réelles, et donc choisir une fonction de *fitness* locale est délicat. Un choix de *fitness* locale, apparaissant intuitivement intéressant au niveau local, peut se révéler une mauvaise option : le comportement global obtenu est inverse du comportement souhaité. Le choix de la fonction de *fitness* est à mettre en corrélation avec le type de modèle : cette méthode a tendance à favoriser les agents qui essaient de maximiser leur récompense au niveau individuel, et donc à favoriser des agents aux comportements égoïstes. Mais, si le type de modèle est plutôt coopératif, et en plus, si le comportement idéal est plutôt coopératif, alors la méthode, favorisant plutôt les agents égoïstes, va amener un résultat aberrant. Le choix de la fonction de *fitness* est réellement la plus grande difficulté de cette méthode.

Il reste des aspects à explorer dans cette méthode. Par exemple, nous avons le choix d'ajouter des agents qui sont initialisés de façon aléatoire visant à éviter une convergence trop rapide, et surtout éviter une convergence vers des extremums locaux. Un autre choix aurait pu être fait comme initialiser les nouveaux agents de la même manière que les « meilleurs » agents (avec plus ou moins d'aléatoire (à la manière des mutations pour un algorithme génétique), et avec plus ou moins de mélange (à la manière des recombinaisons pour un algorithme génétique)).

II.4 Discussion

Dans ce chapitre, nous nous intéressons à comparer les méthodes proposées dans les trois précédents chapitres afin de montrer les caractéristiques essentielles de chacune. Puis nous proposons de nouvelles méthodes en utilisant les points forts de chacune des méthodes précédentes.

Mais en guise d'introduction à ce chapitre, nous pouvons nous demander quelle est le statut ou la validité de modèles dont le paramétrage a été obtenu par les différentes méthodes décrites précédemment. Les approches que nous avons développées utilisent le modèle conçu par le modélisateur ; elles ne jouent que sur la valeur de certains paramètres dans des plages données, c'est-à-dire dans l'espace des paramètres fixé d'avance par le modélisateur. Elles ne créent pas de nouveaux comportement par rapport aux comportements (visibles ou cachés) déjà présents dans le modèle. Si nous avions proposé des approches pouvant modifier le comportement des agents (par exemple, en utilisant la programmation génétique, décrite à la section I.3.2 page 21), nous aurions pu obtenir des comportements des agents, totalement nouveaux, non prévus, voire aberrants par rapport aux systèmes réels : par exemple, dans le fourragement par une colonie de fourmis, des fourmis gigantesques qui n'ont pas besoin de déplacer pour la recherche de nourriture. Le statut du modèle ainsi calibré par nos approches a le même statut que le modèle non calibré. Les méthodes correspondent à la phase où le modélisateur calibre manuellement son modèle, à la différence que nos approches automatisent cette phase d'exploration de l'espace des paramètres. La validité du modèle paramétré via nos approches est la même que le modèle calibré « à la main ». Cependant, si nous obtenons un comportement non prévu par le modélisateur, mais optimal par rapport à l'objectif du calibrage, ce n'est pas la méthode qu'il faut remettre en cause, mais plutôt le modèle sous trois points :

- soit le modélisateur essaye de valider ce comportement à travers de nouvelles expérimentations sur le système réel, ou à travers de nouvelles hypothèses ou de nouveaux modèles ;
- soit il modifie son modèle car celui-ci comporte des comportements aberrants par rapport à la réalités ;
- soit il change son objectif de calibrage ou fonction de *fitness* car celui-ci ne semble pas correspondre à ses attentes.

II.4.1 Comparaison des méthodes

Nous allons comparer dans cette section les différentes méthodes, d'abord au niveau des résultats, puis au niveau des caractéristiques.

II.4.1.1 Comparaison par rapport aux résultats

Nous récapitulons dans cette sous-section les différents résultats sur l'exploration de l'espace des paramètres du modèle « Ants ». Les conditions de tests sont les conditions exposées à la sous-section II.1.2.1 page 35. La figure II.4.1 page suivante montre les différents résultats de chaque méthode. En résumant les différentes conclusions exposées tout au long des trois précédents chapitres, la méthode à base d'algorithme génétique et la méthode découpage de paramètres s'inspirant des colonies de fourmis ont des résultats similaires avec un grand nombre de simulations. Mais, avec peu de simulations, la méthode découpage de paramètres s'inspirant des colonies de fourmis est meilleure que celle à base d'algorithme génétique. Ces deux méthodes-là sont beaucoup plus performantes que les autres méthodes. La méthode évolution d'agents a des résultats en retrait, cependant, pour pouvoir la comparer aux autres méthodes, cette méthode a été dénaturée (voir la sous-section II.3.2.3 page 76).

De nombreuses comparaisons ont été réalisées : par exemple, nous présentons un autre calibrage de modèle dans l'annexe D page 137. Nous avons aussi réalisés des tests sur des « pseudo-simulations » à partir de fonctions mathématiques décrits dans l'annexe E page 143. Nous n'avons présenté ici qu'un seul résultat représentatif des nombreuses comparaisons réalisées.

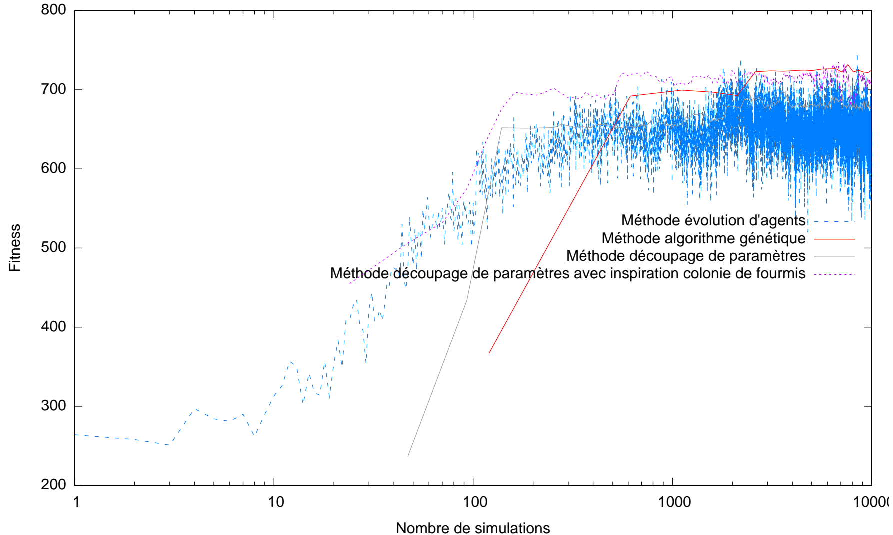


FIG. II.4.1 – *Comparaison des différentes méthodes sur le modèle « Ants »*

II.4.1.2 Au niveau des caractéristiques

Le tableau II.4.1 récapitule les différentes caractéristiques de chaque méthode. Chacune de ces méthodes a des avantages et des inconvénients.

Caractéristiques	Méthode « algorithmes génétiques »	Méthode « découpage de paramètres »	Méthode « évolution d'agents »
Modification du modèle	Ajout de quelques fonctions simples dans le modèle (voir l'annexe C page 133)	Ajout de quelques fonctions simples dans le modèle (voir l'annexe C page 133)	Ajout de fonctions qui peuvent être complexes, comme par exemple la suppression d'un agent qui porte un élément de l'environnement
Calcul distribué	Très simple à mettre en place	Très simple à mettre en place	Très difficile à mettre en place
Connaissance sur l'algorithme	Nécessite de connaître les algorithmes génétiques sauf si on utilise comme outil tout fait de calibrage à la manière de l'outil *Optimization* de la plate-forme `AnyLogic`. Mais le risque est d'avoir des problèmes similaires à ceux d'`AnyLogic` décrit à la section I.2.3 page 18.	Simple	Très simple
Paramètres de la méthode	Les paramètres de l'algorithme génétique sont assez nombreux : nombre de chromosomes, taux de mutation...	Peu de paramètres : savoir quand fusionner et découper les paramètres...	Peu de paramètres : Quand récompenser les agents, et quand les supprimer et les ajouter...

suite à la page suivante

suite de la page précédente			
Caractéristiques	Méthode « algorithmes génétiques »	Méthode « découpage de paramètres »	Méthode « évolution d'agents »
Choix de la *fitness*	Choix de la *fitness* globale	Choix de la *fitness* globale	Choix de la *fitness* locale : c'est un choix plus compliqué, comme expliqué dans la sous-sous-section II.3.1.3 page 71
Principaux avantages	Besoin d'aucune connaissance sur le modèle : vue en boîte noire du modèle à base d'agents	Obtention d'une « pseudo » cartographie de l'espace des paramètres	Stochasticité et rapidité
Principaux désavantages	Vision boîte noire : oubli complet que le modèle est un modèle à base d'agents. En une simulation, on évalue un seul jeu de paramètres	Optimisation de chaque paramètre individuellement en oubliant qu'il peut y avoir des liens entre les paramètres	Choix local de la *fitness*

TAB. II.4.1 – *Tableau récapitulatif des différentes méthodes*

II.4.2 Mélange des méthodes

Comme le montre la sous-section précédente, les méthodes ont toutes des points positifs et des points négatifs. Une idée serait de mélanger les différentes méthodes afin de prendre les avantages de chacune pour réaliser une nouvelle méthode encore plus performante.

II.4.2.1 Découpage de paramètres puis algorithmes génétiques

L'intérêt de la méthode découpage de paramètres est d'avoir de bons résultats avec peu de simulations. Cependant, avec un grand nombre de simulations, la méthode algorithmes génétiques a des résultats meilleurs. L'idée est de commencer par un découpage de paramètres puis de continuer, à partir du découpage en intervalles de chaque paramètre obtenu avec la méthode algorithme génétique. Initialiser un algorithme génétique de façon non-aléatoire peut en effet améliorer les résultats de l'algorithme [Kallel et Schoenauer, 1997].

Cette méthode se déroule en deux phases : dans une première phase, la méthode découpage de paramètre est employée : elle permet de faire un découpage relativement grossier de l'espace des paramètres. Puis, dans une seconde phase, c'est la méthode à base d'algorithmes génétiques qui est utilisée. À la différence de la méthode traditionnelle, la population de chromosomes n'est pas initialisée de façon aléatoire : l'initialisation de cette population va dépendre du découpage de l'espace des paramètres, obtenu précédemment. Pour l'initialisation d'un gène d'un chromosome, le découpage en intervalles correspondant au paramètre codé par le gène est utilisé de la manière suivante : on choisit un intervalle parmi tous les intervalles du paramètre avec une probabilité uniforme ; puis on choisit une valeur parmi toutes les valeurs de l'intervalle sélectionné avec une probabilité uniforme. De plus, une subtilité peut être ajoutée dans l'algorithme génétique au niveau de la mutation : la mutation correspond à la modification d'un gène dans un chromosome et donc d'un paramètre. Ainsi, il est possible de s'appuyer sur le découpage de paramètres lors d'une mutation de la même manière que lors de la phase d'initialisation des chromosomes : choix d'un intervalle de façon aléatoire avec

une probabilité uniforme, puis choix d'un élément dans l'intervalle de façon aléatoire avec une probabilité uniforme. Cette modification de la mutation n'est pas sans risque : elle peut mener l'algorithme vers des optimums locaux. La figure II.4.2 page ci-contre résume le principe de cette méthode.

Cette méthode a été testée dans des conditions similaires, décrites à la sous-section II.1.2.1 page 35. La méthode « découpage de paramètres » a été utilisée pendant 1 000 simulations, puis la phase utilisant les algorithmes génétiques a été utilisée durant les 9 000 simulations restantes. Les mutations utilisées par l'algorithme génétique sont restées traditionnelles, c'est-à-dire complètement aléatoires sans utilisation du découpage en intervalles des paramètres. La figure II.4.3 page 84 montre le résultat, en comparant avec la méthode à base d'algorithmes génétiques. Il est possible de remarquer le changement de type d'algorithmes sur l'évolution de la *fitness* aux environs de 1000 pas de simulation. Les résultats de cette méthode sont équivalents à ceux de la méthode à base d'algorithmes génétiques, pour un grand nombre de simulations. Et, pour un faible nombre de simulations, les résultats de cette méthode sont similaires à ceux de la méthode découpage de paramètres. Donc cette méthode retrouve les principales caractéristiques des deux méthodes dont elle dérive : un démarrage rapide et une convergence très haute avec un grand nombre de simulations.

Cependant, de nombreux tests ont été effectués. Si le découpage obtenu initialement est mauvais, alors les performances de l'algorithme génétique sont très mauvaises, d'autant plus si les mutations utilisent ce découpage. Par contre, si le découpage obtenu initialement est très bon, alors les performances de l'algorithme génétique sont excellentes : les résultats de cette méthode sont supérieurs aux autres méthodes. Mais, le découpage obtenu initialement est en généralement moyen car la méthode découpage de paramètres est appliquée sur un petit nombre de simulations. De plus, il y a un autre paramètre à étudier, à savoir le rapport entre le nombre de simulations pour la partie découpage de paramètres et le nombre de simulations pour la partie algorithme génétique. La figure II.4.4 page 84 montre les résultats de différentes exécutions de la méthode où le nombre de simulations alloué à la partie découpage de paramètres varie. Ce test repose sur des « pseudo-simulations » fondé sur une fonction mathématique, décrits à l'annexe E page 143, plus particulièrement à la section E.3 page 147 pour la fonction mathématique. Chaque courbe correspond à la moyenne de 26 exécutions de la méthode représentée par la courbe. Quelque soit le choix, les résultats au final sont équivalents. Mais, peu de simulations allouées au découpage de paramètres semble être un bon choix permettant combiner au mieux les avantages de deux méthodes : démarrage rapide pour le découpage de paramètres, convergence haute de l'algorithme génétique.

II.4.2.2 Algorithmes génétiques et évolution d'agents

Il est possible aussi d'imaginer une méthode mêlant les algorithmes génétiques et l'évolution d'agents. Au lieu de réaliser un algorithme génétique sur les paramètres du modèle de façon globale, une idée serait de réaliser un algorithme génétique sur les agents. Un modèle à base d'agents comporte un grand nombre d'agents identiques : en tenant compte de cette particularité, dans cette méthode, chaque agent aura un jeu de paramètres différent. Un modèle paramétré sera représenté par un chromosome dont les gènes représenteront les agents (c'est-à-dire qu'un gène sera un jeu de paramètres d'un agent). Ainsi, la phase de recombinaison correspondra à un échange d'agents paramétrés entre deux modèles, et la phase de mutation correspondra à la modification d'un agent dans le modèle. À la fin de l'algorithme génétique, il faudra caractériser la population d'agents solutions : si l'algorithme génétique a convergé, une hypothèse à vérifier est que la population d'agent a aussi convergé vers un type d'agent (voire plusieurs, mais en nombre restreint). En effet, si l'algorithme a convergé, lors de la phase de mutation et de recombinaison, les nouveaux modèles descendants sont moins bons que les modèles parents, et donc l'ajout de variabilité dans les modèles parents n'améliore pas le modèle, ainsi, les modèles parents devraient être constitués d'une population relativement homogène. La figure II.4.5 page 85 résume le principe de cette méthode.

Cette méthode a été testée dans des conditions similaires, décrites à la sous-section II.1.2.1 page 35. La figure II.4.6 page 86 montre les résultats. Ceux-ci sont très mauvais. La population d'agents ne s'est pas homogénéisée. Une explication possible est que pour l'algorithme génétique, au lieu d'avoir 4 paramètres à optimiser, il a 25 agents-paramètre c'est-à-dire $25 \times 4 = 100$ paramètres. En partant de populations d'agents initialisées complètement aléatoirement, le temps de convergence est très long.

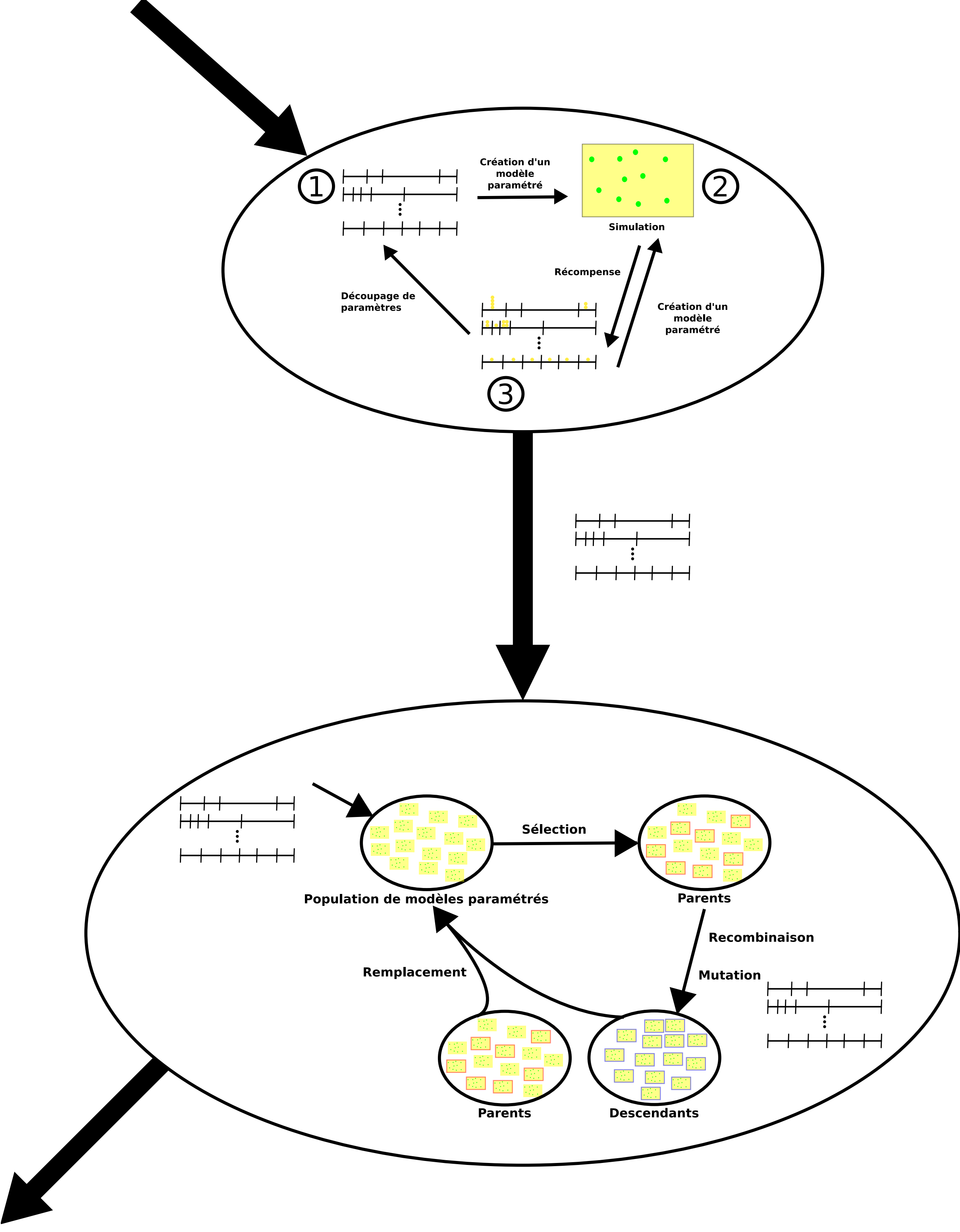


FIG. II.4.2 – *Principe de la méthode découpage de paramètres puis algorithmes génétiques*

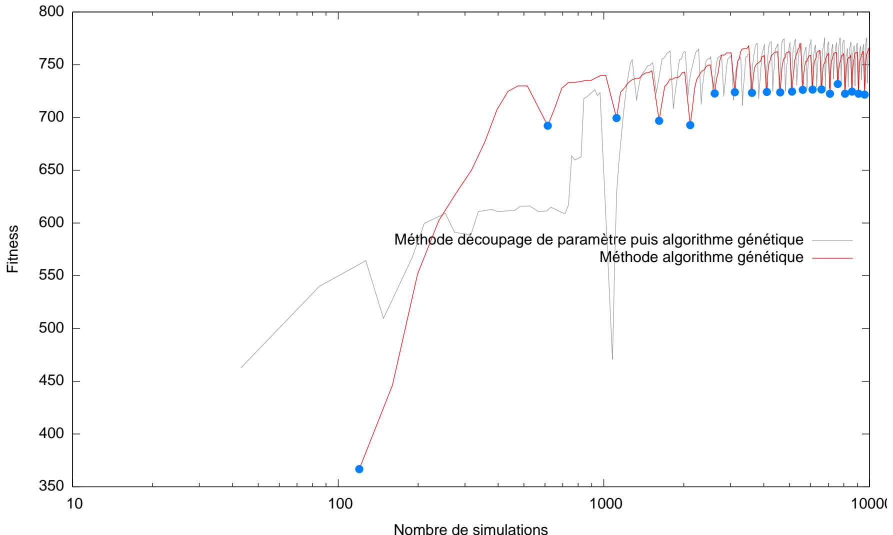


FIG. II.4.3 – *Évolution de la fitness globale en fonction du nombre de simulations pour le modèle « Ants » par la méthode découpage de paramètres puis algorithmes génétiques*

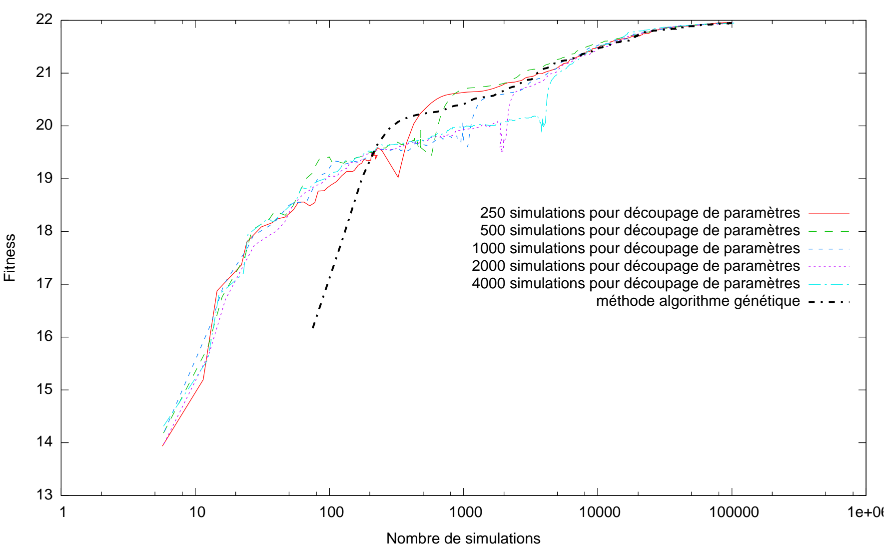


FIG. II.4.4 – *Évolution de la fitness globale en fonction du nombre de simulations et en fonction du nombre de simulations alloués à la partie découpage de paramètres par la méthode découpage de paramètres puis algorithmes génétiques*

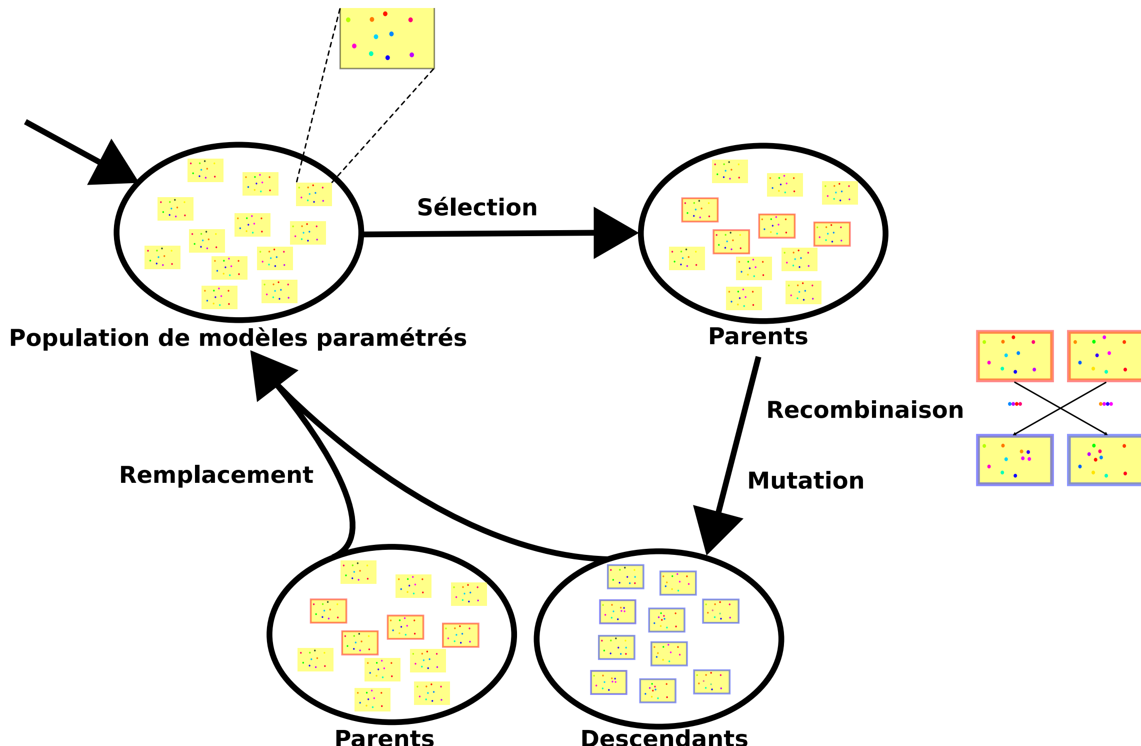


FIG. II.4.5 – *Principe de la méthode algorithmes génétiques et évolution d'agents*

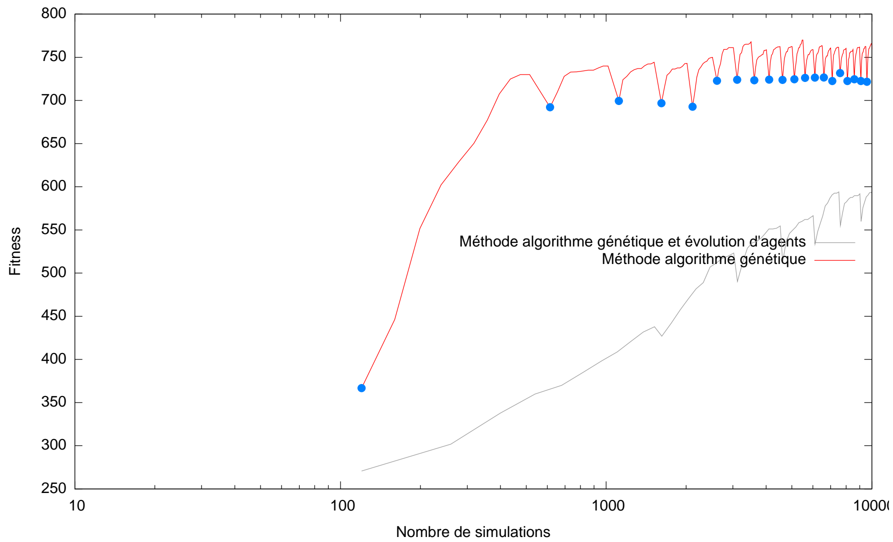


FIG. II.4.6 – *Évolution de la fitness globale en fonction du nombre de simulations pour le modèle « Ants » par la méthode algorithmes génétiques et évolution d'agents*

II.4.2.3 Découpage de paramètres puis algorithmes génétiques et évolutions d'agents

De même, il est possible d'imaginer de mélanger les deux précédentes méthodes : l'intérêt est d'homogénéiser la population d'agents pour la partie « algorithmes génétiques et évolutions d'agents » afin d'accélérer l'algorithme génétique.

D'abord, la méthode découpage de paramètres est exécutée. Puis, à partir du découpage relativement grossier de l'espace des paramètres, la méthode mêlant algorithmes génétiques et évolution d'agents est utilisée. L'initialisation des agents utilise le découpage en intervalles des paramètres obtenu lors de la première partie de la méthode : choix d'un intervalle de façon aléatoire avec une probabilité uniforme, puis choix d'un élément dans l'intervalle choisi de façon aléatoire avec une probabilité uniforme. De la même manière, les mutations peuvent utiliser aussi le découpage en intervalles des paramètres. La figure II.4.7 page 87 résume le principe de cette méthode. Les résultats ne sont pas très enthousiasmants : le temps de convergence de la partie algorithmes génétiques et évolution d'agents reste très long.

II.4.2.4 Discussion

Dans cette section, nous avons proposé quelques idées pour combiner les différentes méthodes proposées dans les chapitres précédents. Après quelques tests de faisabilité, l'intérêt de ces différentes combinaisons n'apparaissent pas flagrants. Mélanger les différentes méthodes ajoute de nouveaux paramètres à la méthode. Le choix de ces nouveaux paramètres et la mise au point de ces méthodes hybrides seront sûrement plus long.

Ces méthodes hybrides peuvent apporter des avantages supplémentaires (par exemple, la première méthode combine l'efficacité de la méthode découpage de paramètres pour peu de pas de simulations, et la performance de la méthode algorithme génétique avec un grand nombre de simulations) mais aussi apporter des inconvénients (par exemple, difficultés de mise au point). En essayant de capturer les avantages de différentes méthodes, nous récupérons aussi les désavantages en les mélangeant. Finalement une méthode simple est sûrement préférable. Avec peu de simulations, la méthode découpage de paramètres semble préférable. Par contre, avec un grand nombre de simulations, la méthode à base d'algorithmes génétiques semble préférable.

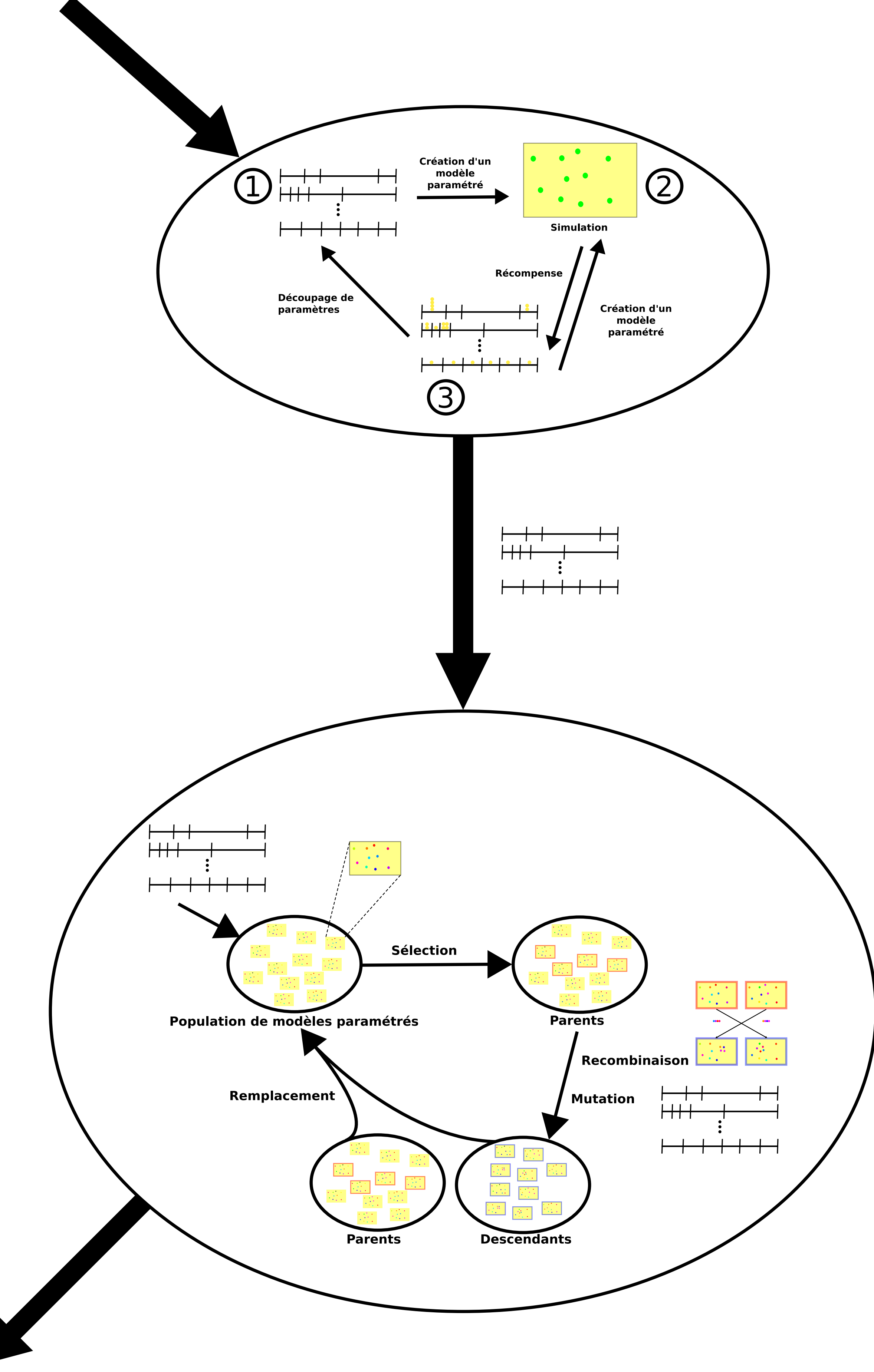


FIG. II.4.7 – *Principe de la méthode découpage de paramètres puis algorithmes génétiques et évolution d'agents*

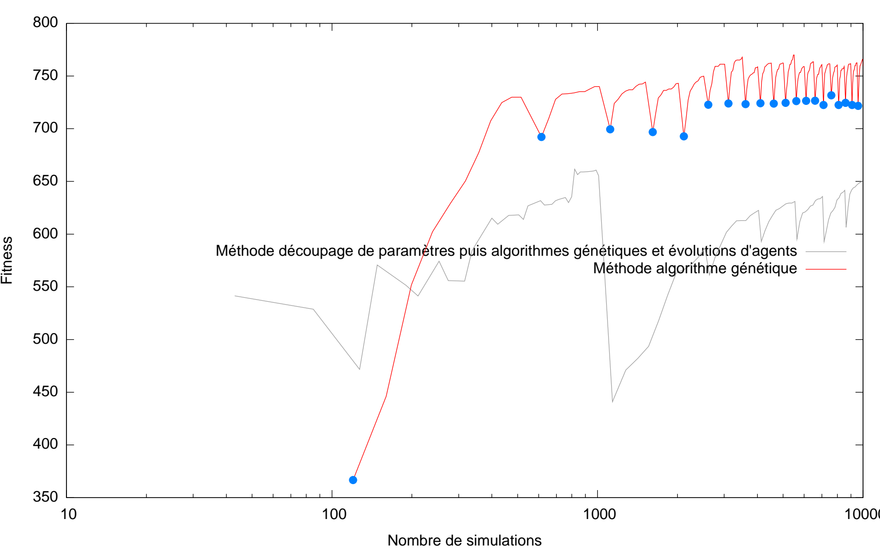


FIG. II.4.8 – *Évolution de la fitness globale en fonction du nombre de simulations pour le modèle « Ants » par la méthode découpage de paramètres puis algorithmes génétiques et évolution d'agents*

II.5 Applications

Ce chapitre présente deux applications à des cas réels des méthodes d'exploration de l'espace des paramètres.

II.5.1 Hyperstructure

Les hyperstructures [Amar et al., 2002a, Amar et al., 2002b] sont des complexes de molécules qui se forment dans le but de réaliser une fonction. De nombreuses hypothèses existent autour des hyperstructures. L'existence et les propriétés de ces hyperstructures ne sont pas clairement établies et sont l'objet de débats. Concernant leur existence, des études ont pu montrer l'existence de petits complexes de 2 molécules [Mitchell, 1996]. Les hypothèses qui pourraient justifier l'existence de telles hyperstructures mettent en avant l'amélioration de l'efficacité des fonctions auxquelles elles participent. Les cascades de réactions enzymatiques permettent d'illustrer ce fait.

Les enzymes [1] sont des protéines [2] qui agissent sur des espèces chimiques (substrats) pour les tranformer. Les produits obtenus peuvent être réutilisés par d'autres enzymes : ainsi de suite, ce qui peut conduire à des cascades de réactions.

Les enzymes peuvent aussi se lier afin de former des complexes (voir figure II.5.1 page suivante), ce qui conduit à une modification des réactions enzymatiques selon le principe suivant :

- l'enzyme $E1$ se lie avec son substrat $S1$, il le transforme en $S2$ qui est le substrat de l'enzyme $E2$;
- étant lié à $E2$, l'enzyme $E1$ passe $S2$ à $E2$;
- l'enzyme $E2$ transforme $S2$ en $S3$.

Si l'arrivée de $S1$ peut se faire en continu, alors la formation de $S3$ se fait en continu : c'est ce que l'on appelle une réaction enzymatique canalisée. Si le complexe ne s'était pas formé, alors $S2$ aurait été libéré dans la cellule et aurait diffusé. Il aurait fallu plus de temps pour qu'elle trouve l'enzyme $E2$ afin de former $S3$.

Plus la chaîne enzymatique est longue, plus le gain de performance doit être grand.

De plus, un des intérêts de ces hyperstructures est la réduction du nombre de molécules actives nécessaires pour le fonctionnement de la cellule. Par exemple, pour les hyperstructures enzymatiques, il y aurait moins de substrats intermédiaires dans la cellule (avec les hyperstructures, tous les substrats intermédiaires réagissent toute de suite avec les enzymes) que dans une cellule dépourvue d'hyperstructures. Cela a un intérêt quand les substances intermédiaires sont labiles (c'est-à-dire se dégradent rapidement) ou nocives pour la cellule.

Une autre propriété des hyperstructures est un aspect régulateur [Thellier et al., a, Thellier et al., b]. Il est ainsi possible d'obtenir des comportements cinétiques sigmoïdes, ou linéaires sur une vaste plage de concentrations de substrats pour les hyperstructures enzymatiques.

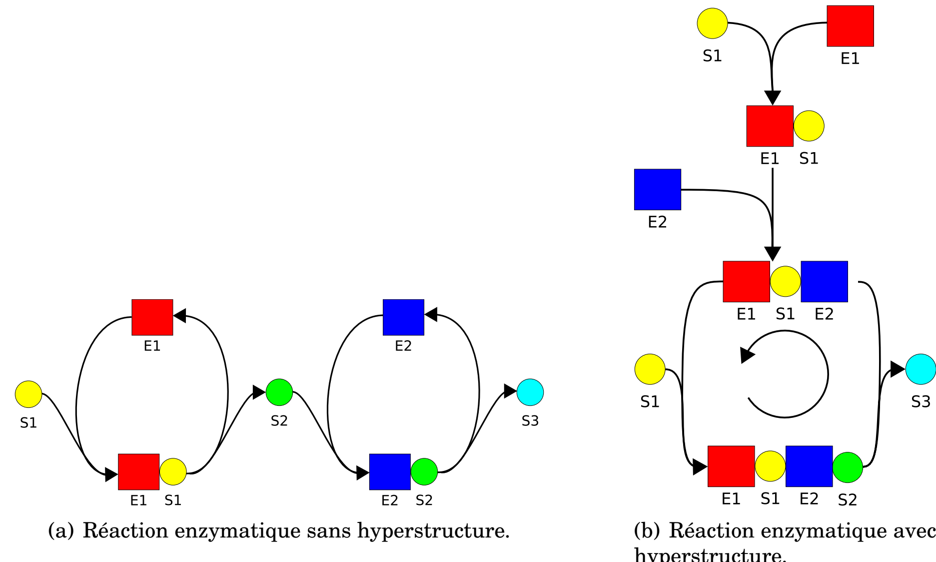
Enfin, les hyperstructures permettent une plus grande flexibilité pour la cellule lors du changement du milieu : changement de concentration de substrat (saturation ou peu de substrat) , changement de type des substrats (une partie de l'hyperstructure peut être commune à plusieurs substrats initiaux)...

L'hyperstructure apparaît comme un objet biologique intéressant. Cependant, l'existence et les propriétés des hyperstructures restent des hypothèses.

---

1. Les enzymes sont des protéines permettant d'accélérer des millions de fois les réactions chimiques du métabolisme se déroulant dans le milieu cellulaire. Les enzymes agissent à faible concentration et elles se retrouvent intactes en fin de réaction: ce sont des catalyseurs.

2. Une protéine est un assemblage (ou séquence) d'acides aminés (au moins 50 acides aminés) liés par des liaisons peptidiques. Les propriétés des acides aminés [3] gouvernent la structure de la protéine. Les protéines ont des fonctions très diverses : certaines pourront avoir une fonction structurale (elles participent à la cohésion structurale des cellules entre elles), enzymatique (elles catalysent les réactions chimiques de la matière vivante) ou encore une fonction de messager (pour les protéines impliquées dans des processus de signalisation cellulaire).



(a) Réaction enzymatique sans hyperstructure.

(b) Réaction enzymatique avec hyperstructure.

FIG. II.5.1 – *Complexe enzymatique*

Le simulateur d'interactions Protéine-Protéine

Pour étudier les hyperstructures, il existe un simulateur d'interactions Protéine-Protéine [Amar et al., 2002c]. Ce simulateur permet d'étudier la formation, le mouvement et la dissociation d'agglomérats de molécules.

Le simulateur modélise une cellule virtuelle comme un espace en 3 dimensions entouré par une membrane de forme ovoïde. La cellule est initialement remplie par des molécules de types différents. Quand la simulation commence, ces molécules diffusent et interagissent selon les règles de réaction décrites par le modèle.

Le simulateur est un automate stochastique régi par des règles de réaction entre molécules.

À chaque pas de simulation (appelé génération) est appliqué le processus suivant :

- une molécule source S est choisie (de façon aléatoire pour éviter les artefacts) ;
- une direction L est choisie, et le simulateur vérifie s'il y a une molécule T proche de S dans la direction L ;
- s'il y a une molécule T, et s'il existe une règle de réaction entre une molécule du type de T et une molécule du type de S, alors cette règle est appliquée selon une probabilité représentant la cinétique de la réaction ;
- s'il n'y a pas de molécule T proche de S, alors la molécule S peut bouger dans la direction L selon une probabilité représentant la vitesse de diffusion.

Quand toutes les molécules ont réagi selon le processus précédent, la génération actuelle est finie et une autre peut commencer. Le temps d'une génération simulée correspond à 100 μs, ce qui correspond au temps moyen pour qu'une protéine puisse se déplacer d'une distance de 10 nm (ce qui représente environ le diamètre d'une protéine).

Pour le simulateur, il faut définir les conditions initiales :

- les molécules et leur type ;
- les vitesses de déplacement des molécules ;
- les quantités initiales des molécules.

Le simulateur implémente quatre types de règles d'interactions entre deux molécules : ($\star$ correspond a un complexe, + correspond a une collision entre deux molécules pour le membre gauche)

- Réaction : `S` réagit avec `T` pour produire deux autres types de molécules `S'` et `T'` ;

$$\mathtt{S} + \mathtt{T} \longrightarrow \mathtt{S'} + \mathtt{T'}$$

- Association : `S` se lie avec `T` pour produire le complexe `S-T` ;

$$\mathtt{S} + \mathtt{T} \longrightarrow \mathtt{S} \star \mathtt{T}$$

– Dissociation : Le complexe S−T peut se casser et libérer les molécules S et T ;

$$S \star T \longrightarrow S + T$$

– Catalyse : Le complexe S−T peut se transformer en S′−T′.

$$S \star T \longrightarrow S' \star T'$$

Comme dans la réaction et dans la catalyse, l'association et la dissociation peuvent changer le type de molécules. Chaque règle a une probabilité donnée d'exécution ce qui correspond, lors de longues simulations, à la cinétique de réaction. Pour l'association, un nombre maximum de liens peut être spécifié.

Voici un exemple de règle d'intéraction :

$$A + B \longrightarrow C \star B \; [0.5]$$

Lors d'une collision entre les molécules A et B, il y a une probabilité de 0.5 pour que ces deux molécules réagissent pour former le complexe C * B.

II.5.1.1 Le système de glucose phosphotransférase et de glycolyse dans *Escherichia coli*

L'application biologique utilisée dans notre étude de l'exploration des paramètres est un modèle simulant le système de glucose phosphotransférase et de glycolyse dans la bactérie *Escherichia coli*.

La glycolyse, appelée voie D'EMBEN-MEYEROFF-PARNAS, est une voie métabolique essentielle dans la vie d'une cellule. Elle dégrade le glucose avec production d'ATP et de métabolites intermédiaires qui peuvent être repris dans d'autres voies métaboliques.

Le système de glucose phosphotransférase (*sugar phosphotransferase system PTS*) permet le passage du glucose à travers la membrane de la bactérie.

Glycolyse

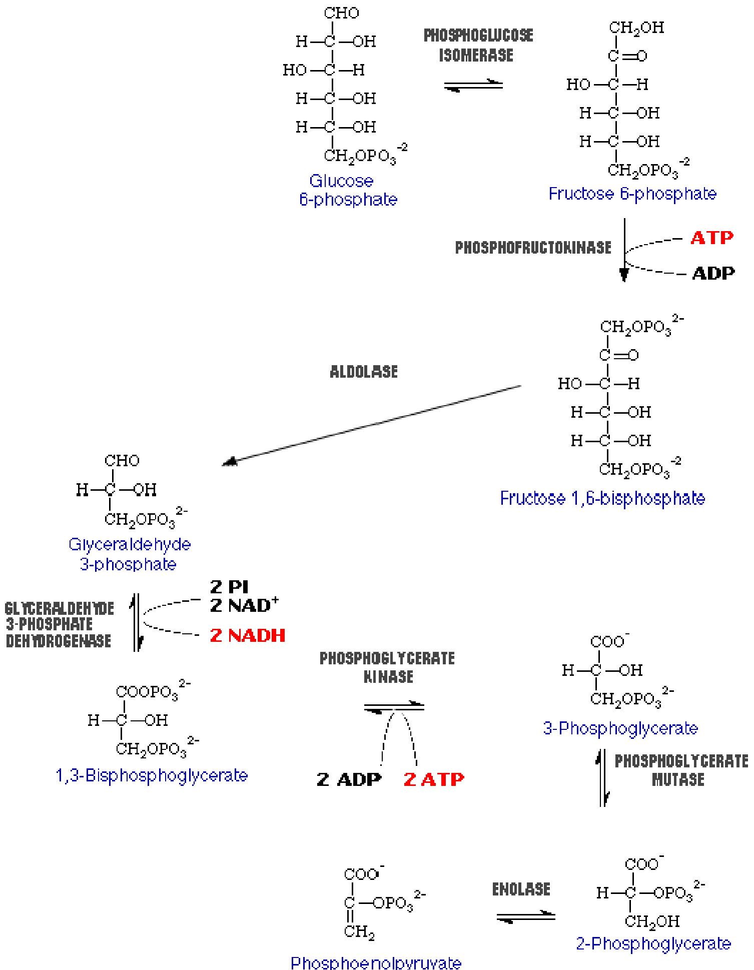
La glycolyse (figure II.5.2 page suivante) est une chaîne métabolique qui permet à une cellule d'extraire de l'énergie directement du glucose. La glycolyse est le phénomène de morcellement du glucose en molécules plus simples, sous l'action de multiples enzymes. Elle transforme le glucose en pyruvate. Cette petite chaîne fournit peu d'énergie : seulement deux molécules d'ATP pour une molécule de glucose dégradée en deux molécules de pyruvate. La glycolyse se déroule en plusieurs étapes, et plus particulièrement :

- Le glucose 6-phosphate (*P_Gluc*) se transforme par action de la phosphoglucose isomérase (*PGI*) en fructose 6-phosphate (*F6P*) ;
- Le fructose 6-phosphate (*F6P*) se transforme par action de la phosphofructokinase (*PFK*) en fructose 1,6-bisphosphate (FBP) en consommant un ATP ;
- Le fructose 1,6-bisphosphate (*FBP*) se transforme par action d'une aldolase (*Aldolase*) en glycéraldehyde 3-phosphate (*GAP*) ;
- Le glycéraldehyde 3-phosphate (*GAP*) se transforme par action de la glycéraldehyde 3-phosphate dehydrogénase (*GAPDH*) en 1,3-bisphophoglycérate (*BPG*) ;
- Le 1,3 Bisphophoglycérate (*BPG*) se transforme par action de la phosphoglycérate kinase (*PGK*) en 3-phosphoglycérate (*PG3*) libérant 2 ATP ;
- Le 3-phosphoglycérate (*PG3*) se transforme par action de la phosphoglycérate mutase (*PGM*) en 2-Phosphoglycérate (*PG2*) ;
- Le 2-phosphoglycérate (*PG2*) se transforme par action de l'énolase (*Enolase*) en phosphoenolpyruvate (*PEP*).

Glucose phosphotransférase

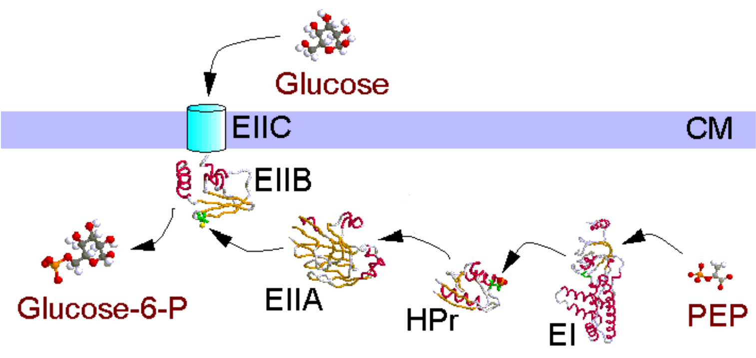
Le système de glucose phosphotransférase(figure II.5.3) qui catalyse le transfert du groupe phosphoryle du phosphoenolpyruvate (PEP) au glucose permet la translocation des molécules de glucose à travers la membrane bactérienne. Ce système se déroule en 5 étapes requérant 4 protéines :

- La phosphoenolpyruvate (*PEP*) réagit avec l'enzyme 1 (*EI*). On obtient de la pyruvate (*Pyr*) et le transfert d'un phosphate à l'enzyme 1 (*P_EI*) ;



source : http://biotech.icmb.utexas.edu/glycolysis/pathway.html

FIG. II.5.2 – *Schéma représentant la Glycolyse*



source : http://www.biologie.uni-hamburg.de/b-online/chimes/kanal/transp/pts/pts.htm

FIG. II.5.3 – *Schéma de la glucose phosphotransférase*

- L'enzyme 1 réagit avec la protéine histidine contenant un transporteur de phosphate (*histidine containing phosphocarrier protein*) (*HPr*). Le groupe phosphate passe de l'enzyme 1 à *HPr* (*P_HPr*) ;
- La protéine *P_HPr* réagit avec l'enzyme 2 A (*EIIA*). Le groupe phosphate est transféré de *P_HPr* à l'enzyme 2 A (*P_EIIA*) ;
- L'enzyme 2 A phosphorisée réagit avec le complexe enzymatique moléculaire (Glucose perméase) (*EIIBC*). Le complexe (*P_EIIBC*) est phosphorylé ;
- Le complexe membranaire phosphorylé réagit avec le glucose se trouvant à l'extérieur de la membrane. Il fait passer le glucose de l'autre coté de la membrane en le phosphorylant : on obtient du glucose 6-phosphate (*P_Gluc*).

Récapitulatif

En résumé (voir figure II.5.4), en ce qui concerne les réactions prises en compte pour le modèle, la glycolyse compte 7 réactions mettant en jeu 7 enzymes et 7 substrats. Pour la glucose phosphotransférase, elle comprend 5 réactions autour de 7 molécules. De plus, ces deux chaînes métaboliques sont reliées. Le glucose 6-phosphate utilisé en entrée de la glycolyse est fabriqué par la glucose phosphotransférase. De même, le phosphoenolpyruvate utilisé comme substrat de la glucose phosphotransferase est fabriqué par la glycolyse. Ainsi, il est obtienu un cycle de réactions.

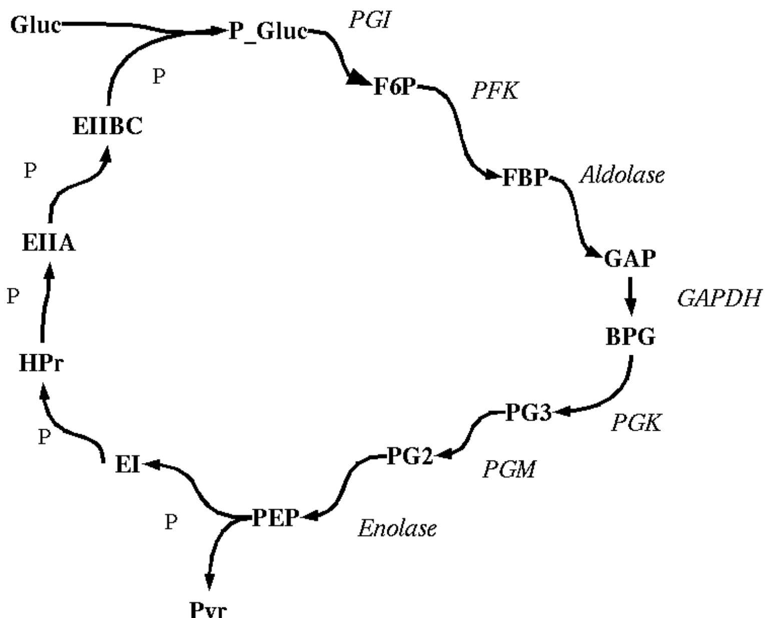


FIG. II.5.4 – *Schéma récapitulatif des réactions*

Finalement, le modèle étudié est constitué de 12 réactions mettant en jeu 21 molécules.

Des observations biologiques [Amar et al., 2002c] ont montré qu'il existe des complexes de deux enzymes dans la glycolyse et la phosphotransférase dans *Escherichia coli* dont on suspecte qu'il s'agit d'hyperstructures. Alors il serait intéressant de tester les hypothèses des hyperstructures : amélioration des performances, aspects régulateurs... De plus, au niveau de la flexibilité, des connaissances biologiques affirment qu'une partie des enzymes de la glycolyse et de la phosphotransférase peuvent être communes à plusieurs sucres. Seules quelques enzymes sont spécifiques à un sucre.

Ainsi, sur ce cas biologique, il parait opportun de vérifier les hypothèses des hyperstructures à partir de leurs propriétés. L'hypothèse la plus intéressante à tester est l'amélioration de l'efficacité de la réaction par les hyperstructures et la détermination des types d'hyperstructures les plus efficaces s'il y a amélioration.

II.5.1.2 Résultats

La question biologique intéressante est de savoir s'il est possible de vérifier les hypothèses des hyperstructures à partir de leurs propriétés. Ainsi, pour vérifier cette hypothèse, les paramètres intéressants à faire varier sont les coefficients d'association et de dissociation entre enzymes. Et la fonction de *fitness* sera la quantité de pyruvate produit au bout de 500000 pas de simulations. De nombreux tests ont été réalisés : en partant d'un modèle de référence, en supposant le substrat intermédiaire labile (susceptible d'être dégradé par des enzymes présentes dans le milieu), avec des faibles concentrations d'enzymes, avec l'ajout d'enzymes « gloutons »... Ici, deux résultats sont présentés : un avec un modèle normal, et un en supposant le substrat intermédiaire labile. La figure II.5.5 montre le résultat de l'exploration de paramètres avec la méthode à base d'algorithme génétique comparé avec un modèle sans hyperstructure, et avec un modèle avec de grandes hyperstructures. Et la figure II.5.6 montre la même comparaison que précédemment mais dans le cadre où les métabolites intermédiaires sont labiles. À chaque fois, l'exploration de l'espace des paramètres a trouvé un modèle meilleur que les deux modèles de références. Ce modèle, résultat de l'exploration de l'espace des paramètres, a les mêmes caractéristiques dans les deux cas : des petites hyperstructures (2 à 3 enzymes) très dynamiques (en permanence en formation et destruction).

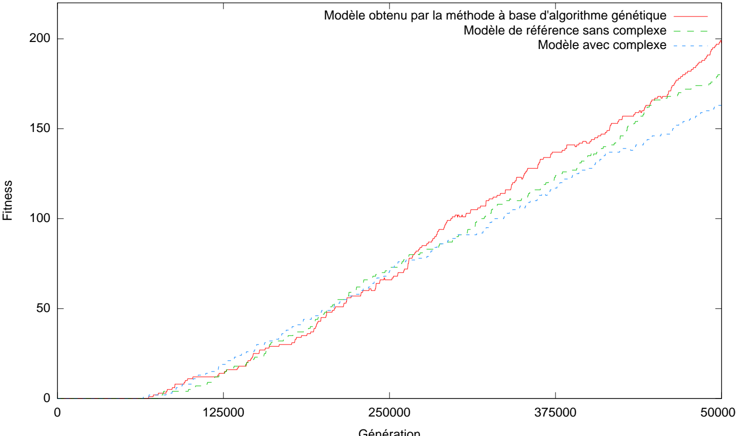


FIG. II.5.5 – *Comparaison de résultats sur les hyperstructures.*

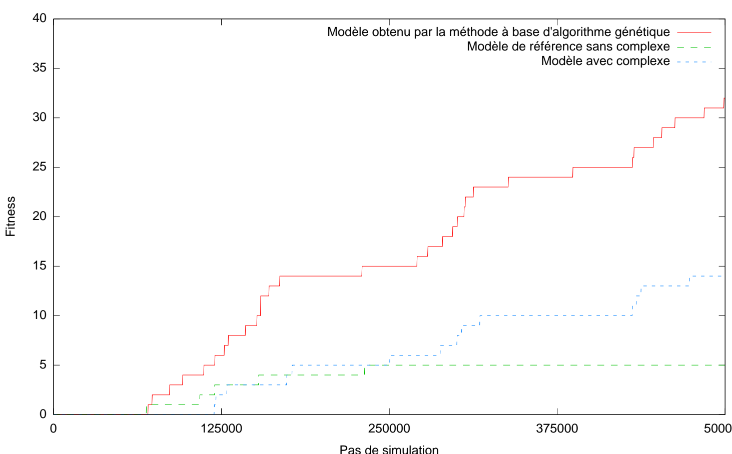


FIG. II.5.6 – *Comparaison de résultats sur les hyperstructures dans le cadre où les métabolites intermédiaires sont labiles.*

II.5.2 Hétérogénéité dans les cultures *in vitro* de populations de cellules clonales

Cette application se fonde sur l'étude de l'origine de l'hétérogénéité dans les cultures *in vitro* de populations de cellules clonales[Stockholm et al., 2007].

L'hétérogénéité phénotypique dans une population de cellules génétiquement homogènes est fréquemment observée dans les cultures *in vitro* de cellules. Dans les systèmes cellulaires *in vitro* de mammifères, l'émergence spontanée d'hétérogénéité phénotypique est plutôt la règle que l'exception. Par exemple, deux sous-populations apparaissent spontanément dans la prolifération de cellules myogéniques de souris *C2C12* : la population principale (*MP*[4] représentant 95 à 99% de la population totale et la population restante *SP*[5] représentant 5 à 1% de la population totale. À partir de ces observations, Paldi et Stockholm [Stockholm et al., 2007] ont échafaudé deux hypothèses de fonctionnement : une hypothèse de fonctionnement intrinsèque et une hypothèse de fonctionnement extrinsèque.

Pour vérifier leurs hypothèses, ils ont créé trois modèles à base d'agents sur la plate-forme `NetLogo`: un modèle intrinsèque, un modèle extrinsèque et un modèle hybride. Le modèle représente la croissance de cellules *C2C12* : il y a des cellules de type A représentant la population *SP*, des cellules de type B représentant la population *MP*. La figure II.5.7 montre le modèle dans la plate-forme `NetLogo`.

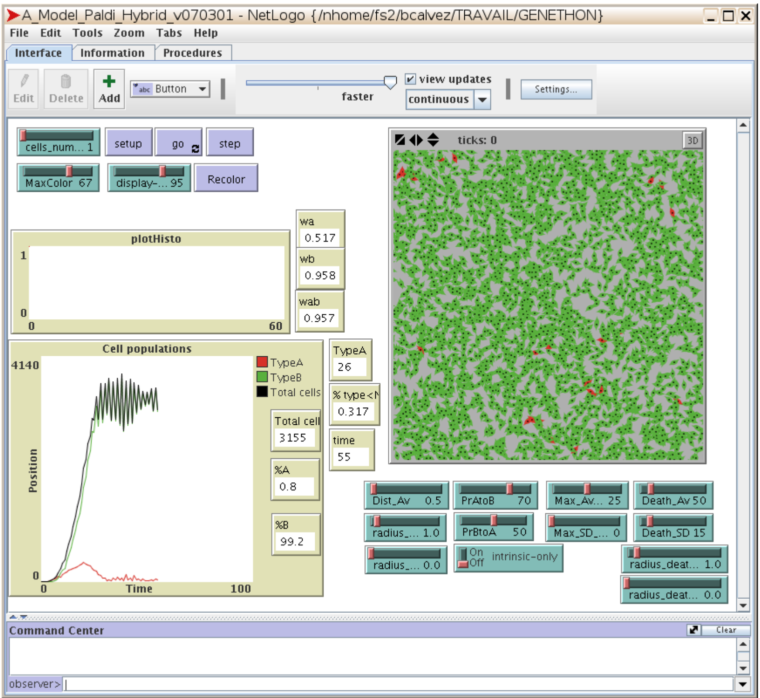


FIG. II.5.7 – *Image du modèle représentant une population de cellule C2C12*

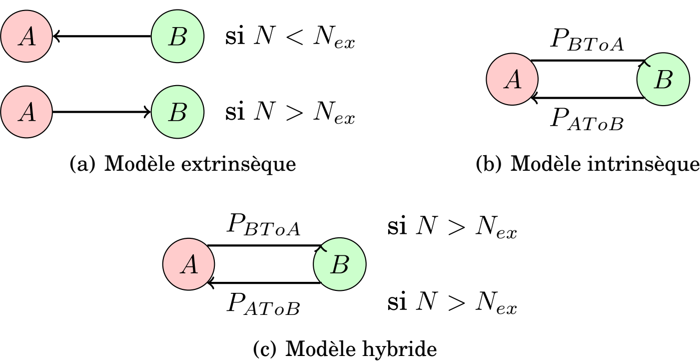
Dans le modèle intrinsèque, une cellule de type A a une probabilité fixe P_{AtoB} de se transformer en cellule de type B. De la même manière, une cellule de type B a une probabilité fixe P_{BtoA} de se transformer en cellule de type A.

Dans le modèle extrinsèque, le changement de phénotype est la conséquence de changements dans le micro-environnement cellulaire. Des cellules de type A se changent en type B si le nombre de voisins dans un rayon R autour d'elles est supérieur à N_ex. De même, des cellules de type B se changent en type A si le nombre de voisins dans un rayon R autour d'elles est inférieur N_ex.

Le modèle hybride est un modèle mixant les deux précédents modèles.

La figure II.5.8 page suivante résume les différentes hypothèses de fonctionnement.

4. pour *main population*
5. pour *side population*



(a) Modèle extrinsèque

(b) Modèle intrinsèque

(c) Modèle hybride

FIG. II.5.8 – *Principe des différentes hypothèses*

La question est de savoir quelle est l'hypothèse de fonctionnement « correcte » c'est-à-dire en concordance avec les observations des cellules *C2C12*. Une particularité dans les observations des cellules *C2C12* est dans la distribution du nombre de voisins de la population *SP* : cette distribution est caractérisé par deux pics, comme la figure II.5.9 le montre. Le but de l'exploration de l'espace des paramètres est, à partir du modèle hybride, de jouer sur les paramètres propres aux modèles intrinsèque et extrinsèque (12 paramètres) afin de retrouver une distribution du nombre de voisins de la population *SP* avec deux pics.

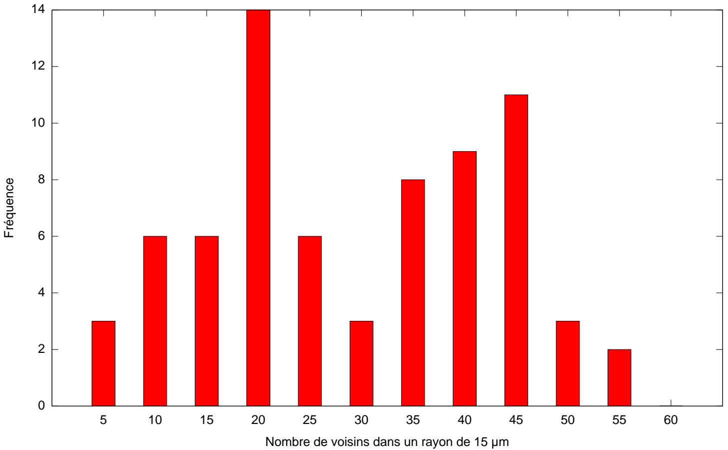


FIG. II.5.9 – *Histogramme de la distribution du nombre de voisins de la population SP*

Pour explorer cet espace, nous avons choisi une fonction de *fitness* qui se découpe en quatre points :

– Caractérisation de l'histogramme de la distribution du nombre de voisins de la population *SP* avec deux pics : l'hypothèse est que plus un modèle présente deux pics plus fortement marqués dans l'histogramme de la distribution du nombre de voisins de la population *SP*, plus le modèle est considéré comme intéressant.
– Proportion de cellules du type A et du type B : nous essayons via ce point d'éviter d'obtenir des modèles où la proportion dans le modèle simulé serait totalement différente des observations du système réel.
– Nombre de cellules au total : ce choix est pour éviter d'obtenir des modèles avec un très grand nombre de cellules.
– Stabilité du nombre de cellules sur 20 pas de simulation : ce point est pour favoriser les modèles stables. Une population de cellules est relativement stable en nombre d'individus entre deux pas de simulations.

Le modèle est simulé pendant 2500 pas de simulations.

Nous présentons quelques résultats. La figure II.5.10 page ci-contre montre l'évolution de la *fitness* par la méthode à base d'algorithmes génétiques. La figure II.5.11 page 98 montre l'évolution du

découpage du paramètre `radius_av` par la méthode de découpage de paramètres. Cependant ces résultats ne sont que des résultats très préliminaires, avec peu de signification. Actuellement, le modèle présente des problèmes importants de stabilité : avec un même jeu de paramètres, nous pouvons obtenir des résultats très différents, voire opposés. La stochasticité du modèle est très forte. Multiplier le nombre de simulations sur un modèle avec un même jeu de paramètres pour obtenir une *fitness* précise n'a peut-être aucune signification : le modèle a un comportement tellement chaotique à certains paramétrages que le calcul d'une *fitness* n'a pas de sens. Il y a un travail d'amélioration de la stabilité du modèle à réaliser.

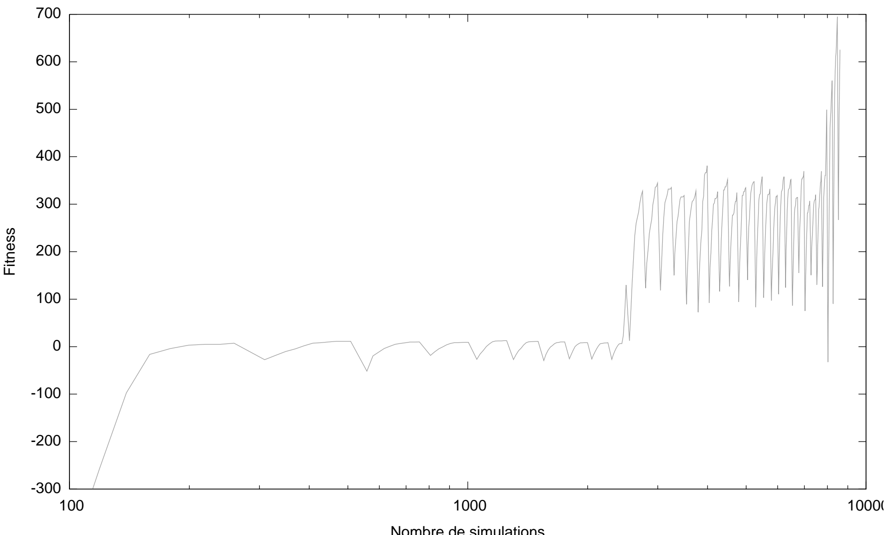


FIG. II.5.10 – *Évolution de la fitness du modèle sur l'hétérogénéité dans les cultures in vitro de populations de cellules clonales par la méthode à base d'algorithmes génétiques*

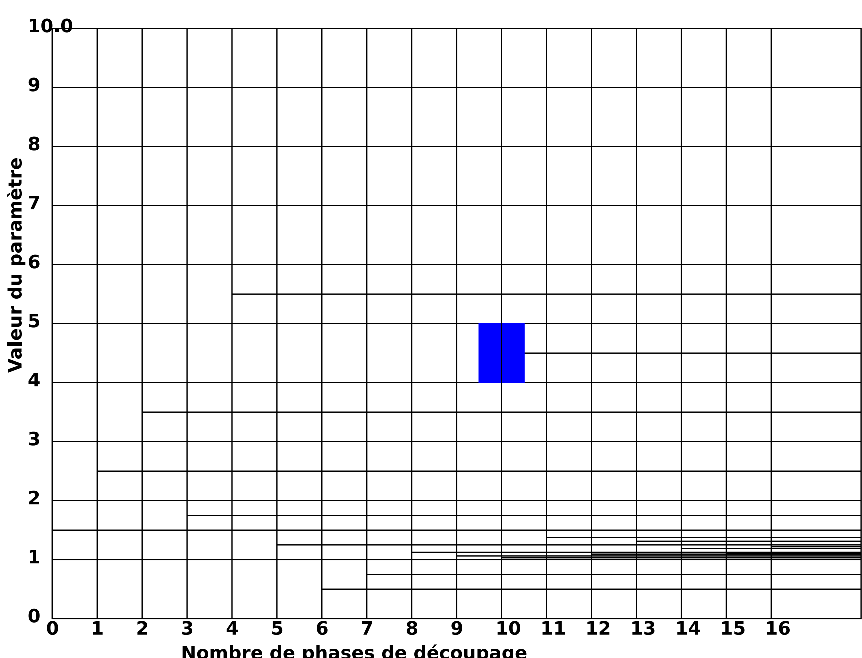


FIG. II.5.11 – *Évolution du découpage du paramètre* `radius_av` *du modèle sur l'hétérogénéité dans les cultures in vitro de populations de cellules clonales par la méthode de découpage de paramètres*

Conclusion

Dans ce manuscrit, nous nous sommes placés dans le cadre de la simulation de modèles à base d'agents pour les systèmes complexes. Nous nous sommes intéressés à la conception de ces modèles, et plus particulièrement à leur calibrage, visant à trouver un ou des jeux de paramètres satisfaisants tout en minimisant le nombre de simulations nécessaires. Comme nous l'avons montré, cette phase de calibrage est à la fois une étape essentielle dans la conception d'un modèle multi-agent et un problème très difficile, pour lequel peu de solutions générales et satisfaisantes ont été proposées.

L'approche que nous avons proposée consiste à aborder ce problème comme un problème d'optimisation, en évaluant la qualité d'un modèle paramétré par une fonction objectif ou *fitness* dépendant des résultats de la simulation du modèle et des questions posées par le modélisateur. L'objectif est donc reformulé en un objectif d'optimisation de cette fonction. Nous avons développé trois méthodes distinctes réalisant une exploration automatique de l'espace de paramètres, décrites dans le seconde partie de ce manuscrit.

- La première approche consiste à appliquer à ce problème un algorithme d'optimisation « classique », en considérant que le modèle à base d'agents peut être vu comme une boîte noire dont les entrées sont les valeurs des paramètres, et dont la sortie est la valeur de la *fitness*, calculée après la simulation du modèle ainsi paramétré ;
- La deuxième approche résulte du souhait d'avoir une méthode à la fois plus systématique et de convergence plus rapide. Cette méthode, dite de « découpage de paramètres » ou « Adaptation Dichotomique Adaptative », explore l'espace des paramètres de manière adaptative : chaque paramètre est découpé en un ensemble d'intervalles, les intervalles a priori les plus intéressants étant successivement découpés de plus en plus finement, ceux qui sont a priori moins intéressants étant fusionnés ;
- La troisième approche, dite d' « évolution d'agents », consiste à paramétrer individuellement chacun des agents, en modifiant leur paramétrage dynamiquement. Cela permet de n'avoir qu'une seule simulation au cours de laquelle le paramétrage des agents évolue jusqu'à obtenir une dynamique globale satisfaisante du système.

Nous comparons les caractéristiques de ces différentes approches dans le chapitre II.4. La première méthode, basée sur l'utilisation des algorithmes d'optimisation classiques, correspond à une approche très générale qui fonctionne bien dans la plupart des situations, l'inconvénient principal étant le nombre souvent important de simulations à réaliser avant d'obtenir des résultats corrects. À l'opposé, la troisième méthode, dite d' « évolution d'agents », est potentiellement très rapide puisque l'optimisation peut s'effectuer au cours d'une seule simulation. Cependant elle présente l'inconvénient majeur de nécessiter une adaptation fine et délicate du modèle, et de reposer sur le choix d'une fonction de *fitness* locale aux agents, également problématique : de même qu'il n'est pas simple de savoir a priori quel comportement individuel d'agent conduira à une dynamique globale particulière, de même il n'est pas simple de savoir a priori quelle *fitness* locale conduira à une *fitness* globale satisfaisante. La deuxième méthode, dite de « découpage de paramètres » se positionne quant à elle de manière intermédiaire entre les deux autres. Comme la première, c'est une méthode très générale, parallélisable, et qui se base sur une mesure de *fitness* globale, relativement simple à mettre en oeuvre. Comme la troisième, elle utilise un paramétrage différencié des différents agents, ce qui permet d'explorer en parallèle différents paramétrages, et accélère la convergence initiale de l'optimisation en comparaison avec la première méthode. Enfin, contrairement aux deux autres qui fournissent seulement un paramétrage solution, cette méthode fournit une cartographie de l'espace des paramètres qui ouvre la voie à une étude plus systématique du paramétrage des modèles.

Toutes les méthodes ont été implantées en java de manière générique, utilisable en ligne de commande avec des fichiers de configuration en `xml` : l'annexe C page 133 en détaille l'implantation qui représente environ 10000 lignes de code.

Différentes perspectives s'ouvrent à présent pour prolonger le travail que nous venons d'exposer dans ce manuscrit. Ces perspectives peuvent se ranger dans trois grandes catégories :

- le développement et la mise au point des méthodes proposées ;
- l'aide au modélisateur ;
- l'application à la simulation de systèmes complexes et à la conception de systèmes à base d'agents.

Développement et mise au point des méthodes

Une perspective très générale qui concerne les différentes approches correspond à l'étude de la convergence de l'optimisation. Dans le cadre de la thèse, nous avons travaillé en nous basant sur des modèles simples, voire sur des fonctions mathématiques à optimiser, de manière à obtenir rapidement des résultats dans la mise au point de nos méthodes. Nous avons donc fixé comme critère d'arrêt de l'optimisation, un nombre fixe de simulations à réaliser, en l'occurrence 10000, ce qui constitue une valeur irréaliste si l'on doit optimiser des modèles dont la durée de simulation se compte en heures voire en jours, même en parallélisant les méthodes. Il est donc important de disposer d'un contrôle fin du processus d'optimisation pour savoir s'il a convergé ou s'il faut poursuivre. En particulier, dans la méthode de « découpage de paramètres », on peut parfaitement envisager de suivre cette convergence, paramètre par paramètre, de manière à stopper l'optimisation sur certains paramètres pour lesquels on aurait détecté une convergence, tout en la continuant sur d'autres paramètres pour lesquels l'algorithme n'aurait pas encore convergé vers une valeur unique.

Un deuxième aspect en liaison avec le développement de nos méthodes concerne l'analyse des résultats. Une fois que les algorithmes ont tourné sur le calibrage d'un modèle, il reste en effet un travail important de dépouillement des résultats. Dans notre manuscrit, nous avons expliqué comment le paramétrage solution était construit à l'issue de l'optimisation, tout en soulignant la possibilité d'effectuer des choix différents. Il serait important de pouvoir disposer d'outils d'analyse des populations de modèles obtenus à la fin de l'optimisation de manière à pouvoir affiner les critères de choix du paramétrage solution. En particulier, dans le cadre de la méthode de « découpage de paramètres », nous avons souligné la « cartographie » de l'espace des paramètres à laquelle cette méthode donnait accès, mais il reste à développer les outils permettant d'en tirer réellement partie, par exemple en donnant la possibilité de visualiser cet espace, en plusieurs dimensions (2D, 3D) et de manière interactive.

Aide au modélisateur

Du point de vue de l'aide au modélisateur, un premier point à explorer est d'ordre méthodologique : il est facile de voir, dans la synthèse ci-dessus qu'aucune des trois méthodes ne peut être présentée comme supérieure aux autres dans tous les cas, quelque soit le modèle étudié et quelque soit la question posée par le modélisateur. Il est donc important de pouvoir donner à ce dernier les éléments de choix lui permettant, en fonction du modèle et des questions posées, de déterminer la méthode a priori la plus à même de lui renvoyer une réponse satisfaisante en un temps raisonnable. Les quelques pistes données dans le manuscrit devront donc être complétées pour obtenir une méthodologie plus complète et systématique.

Pour le néophyte, l'utilisation de l'implantation réalisée durant la thèse reste relativement compliquée. L'aide au modélisateur passe donc également par le développement d'une interface graphique, permettant d'utiliser les méthodes présentées dans ce manuscrit de manière aussi simple que possible, sans connaissance particulière des algorithmes sous-jacents. Celle-ci doit permettre de paramétrer et lancer le calibrage, de manière couplée avec un simulateur. Par défaut, on peut imaginer que ce simulateur soit `NetLogo` mais on peut aussi envisager de définir une `API` et un format de description des paramètres permettant d'adapter le système pour utiliser tout type de plate-forme de simulation.

Application à la simulation et à la conception de systèmes multi-agents

Nous sommes engagés actuellement dans différents projets visant à appliquer nos méthodes pour le calibrage et la mise au point de modèles à base d'agents, notamment dans le domaine de la biologie. L'un de ces projets concerne le prolongement du travail présenté dans ce manuscrit sur la différentiation cellulaire. Un deuxième projet, en collaboration avec l'équipe *Dynamic* d'IBISC, concerne la modélisation de la croissance tumorale, l'étude des différents modes de migration cellulaire, et de l'alternance des phases de prolifération et de migration. Dans ce cadre, nous sommes confrontés à la problématique spécifique de l'occurrence de phénomènes rares : l'échappement d'une cellule cancéreuse de la tumeur principale, conduisant à la constitution de métastases est un phénomène rare se produisant une fois sur 10000. Dans ces conditions, un phénomène pourra ne pas être observé, non pas parce que le modèle est mal paramétré mais parce que le phénomène n'a pas encore eu le temps de se produire. Il faudra donc probablement adapter nos méthodes pour prendre en compte cette caractéristique spécifique.

Enfin, les méthodes que nous avons développées se situent très en aval dans la chaîne de conception d'un modèle à base d'agents, c'est-à-dire une fois que le modèle a été entièrement conçu, que les agents ont été choisis, leurs comportements déterminés et les paramètres contrôlant ce comportement identifiés. Pour maximiser l'efficacité des méthodes que nous avons développées, il serait sans doute profitable d'intervenir plus en amont dans cette chaîne de conception. Des travaux existent (par exemple, calibrage en boîte blanche I.2.1 page 14), mais cela reste très complexe à utiliser. L'objectif à terme pourrait être de créer une plate-forme de simulation où l'utilisateur créerait des agents, préciserait leur comportement dans un formalisme plus abstrait qu'un langage de programmation, et expliciterait les caractéristiques attendues du point de vue de la dynamique globale de la simulation. Le système aurait alors la charge d'instancier et de paramétrer automatiquement le comportement des agents de manière à obtenir la dynamique globale souhaitée. Un tel type de système, qui constitue une perspective à plus long terme, aurait un intérêt non seulement pour la simulation de systèmes complexes mais plus largement pour la conception de tous types de systèmes multi-agents.

Bibliographie

[Ackley, 1987] David H. Ackley (1987). *A connectionist machine for genetic hillclimbing*. Kluwer Academic Publishers, Norwell, MA, USA. http://portal.acm.org/citation.cfm?id=SERIES9588.40713&coll=GUIDE&dl=ACM&CFID=521902&CFTOKEN=68138139.

[Amar et al., 2002a] Patrick Amar, Pascal Ballet, Georgia Barlovatz-Meimon, Arndt Benecke, Gilles Bernot, Yves Bouligand, Paul Bourguine, Franck Delaplace, Jean-Marc Delosme, Maurice Demarty, Itzhak Fishov, Eric Fourmentin-Guilbert, Joe Fralick, Jean-Louis Giavitto, Bernard Gleyse, Christophe Godin, Roberto Incitti, François Képès, Catherine Lange, Lois Le Sceller, Corinne Loutellier, Olivier Michel, Franck Molina, Chantal Monnier, René Natowicz, Vic Norris, Nicole Orange, Helene Pollard, Derek Raine, Camille Ripoll, Josette Rouviere-Yaniv, Milton Saier jnr., Paul Soler, Pierre Tambourin, Michel Thellier, Philippe Tracqui, Dave Ussery, Jean-Pierre Vannier, Jean-Claude Vincent, Philippa Wiggins, et Abdallah Zemirline (2002a). Hyperstructures. In *Proceedings of the Dieppe spring school on Modelling and simulation of biological processes in the context of genomics*, pages 137–159.

[Amar et al., 2002b] Patrick Amar, Pascal Ballet, Georgia Barlovatz-Meimon, Arndt Benecke, Gilles Bernot, Yves Bouligand, Paul Bourguine, Franck Delaplace, Jean-Marc Delosme, Maurice Demarty, Itzhak Fishov, Jean Fourmentin-Guilbert, Joe Fralick, Jean-Louis Giavitto, Bernard Gleyse, Christophe Godin, Roberto Incitti, François Képès, Catherine Lange, Lois Le Sceller, Corinne Loutellier, Olivier Michel, Franck Molina, Chantal Monnier, René Natowicz, Vic Norris, Nicole Orange, Helene Pollard, Derek Raine, Camille Ripoll, Josette Rouviere-Yaniv, Milton Saier jnr., Paul Soler, Pierre Tambourin, Michel Thellier, Philippe Tracqui, Dave Ussery, Jean-Claude Vincent, Jean-Pierre Vannier, Philippa Wiggins, et Abdallah Zemirline (2002b). Hyperstructures, genome analysis and I-cell. *Acta Biotheoretica*, 50:357–373.

[Amar et al., 2002c] Patrick Amar, Gilles Bernot, et Victor Norris (2002c). Modelling and Simulation of Large Assemblies of Proteins. In *Proceedings of the Dieppe spring school on Modelling and simulationof biological processes in the context of genomics*, pages 36–42.

[Antunes et al., 2006a] Luis Antunes, João Balsa, Ana Respício, et Helder Coelho (2006a). Tactical exploration of tax compliance decisions in multi-agent based simulation. In *MABS'06 : Seventh International Workshop on Multi-Agent-Based Simulation*.

[Antunes et al., 2006b] Luis Antunes, Helder Coelho, João Balsa, et Ana Respício (2006b). E*plore v.0: Principia for strategic exploration of social simulation experiments design space. In *The First World Congress on Social Simulation*. labmag.di.fc.ul.pt/guess/publications/antunes2006wcss.pdf.

[Arostegui et al., 2006] Jr Arostegui, Sukran N. Kadipasaoglu, et Basheer M. Khumawala (2006). An empirical comparison of tabu search, simulated annealing, and genetic algorithms for facilities location problems. *International Journal of Production Economics*, 103(2):742–754. http://www.sciencedirect.com/science/article/B6VF8-4K66DYW-2/2/1432230e34ae52539bd2825f21cd7760.

[Axtell et al., 1996] Robert Axtell, Robert Axelrod, Joshua M. Epstein, et Michael D. Cohen (1996). Aligning simulation models: A case study and results. *Computational & Mathematical Organization Theory*, 1(2):123–141. http://dx.doi.org/10.1007/BF01299065.

[Back et al., 1995] T. Back, T. Beielstein, B. Naujoks, et J. Heistermann (1995). Evolutionary algorithms for the optimization of simulation models using pvm. In J. Dongarra, M. Gengler, B. Tourancheau, , et X. Vigouroux, editeur, *EuroPVM '95: Second European PVM Users' Group Meeting*, pages 277–282. Hermes. http://citeseer.ist.psu.edu/back95evolutionary.html.

[Beyer, 2000] H.ans-Georg Beyer (2000). Evolutionary algorithms in noisy environments: Theoretical issues and guidelines for practice. *Computer Methods in Applied Mechanics and Engineering*, 186:239–267. http://citeseer.ist.psu.edu/beyer98evolutionary.html.

[Bianchi et al., 2007a] Carlo Bianchi, Pasquale Cirillo, Mauro Gallegati, et Pietro A. PVagliasindi (2007a). Validation and calibration in agent-based models: An investigation of the cats model. *Journal of Econimic Behaviour and Organization*. forthcoming.

[Bianchi et al., 2007b] Carlo Bianchi, Pasquale Cirillo, Mauro Gallegati, et Pietro Vagliasindi (2007b). Validating and calibrating agent-based models: A case study. *Computational Economics*. http://dx.doi.org/10.1007/s10614-007-9097-z.

[Bilchev et Parmee, 1995] George Bilchev et Ian C. Parmee (1995). The ant colony metaphor for searching continuous design spaces. In Terence C. Fogarty, editor, *Evolutionary Computing, AISB Workshop*, volume 993 of *Lecture Notes in Computer Science*, pages 25–39. Springer.

[Bird, 1993] Shawn D. Bird (1993). Toward a taxonomy of multi-agent systems. *International Journal of Man-Machine Studies*, 39(4):689–704. http://www.sciencedirect.com/science/article/B6WGS-45PTH1T-T/2/035ab35f03f5b922938eda5d01c0dea3.

[Blum, 2005] Christian Blum (2005). Ant colony optimization: Introduction and recent trends. *Physics of Life Reviews*, 2(4):353–373. http://www.sciencedirect.com/science/article/B75DC-4HM7S2N-1/2/b515e1932596d4416a1c8bec13dadab0.

[Bonabeau, 2002] Eric Bonabeau (2002). Agent-based modeling: Methods and techniques for simulating human systems. *PNAS*, 99(90003):7280–7287. http://www.pnas.org/cgi/content/abstract/99/suppl_3/7280. http://dx.doi.org/10.1073/pnas.082080899.

[Bonabeau et al., 1999] Eric Bonabeau, Marco Dorigo, et Guy Theraulaz (1999). *Swarm Intelligence From Natural to Artificial Systems*. Oxford University Press.

[Branke et Schmidt, 2005] J. Branke et C. Schmidt (2005). Faster convergence by means of fitness estimation. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 9(1):13–20. http://dx.doi.org/10.1007/s00500-003-0329-4.

[Brueckner et Parunak, 2003] Sven A. Brueckner et H. Van Dyke Parunak (2003). Resource-aware exploration of the emergent dynamics of simulated systems. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 781–788, New York, NY, USA. ACM Press. http://www.newvectors.net/staff/parunakv/AAMAS03APSE.pdf. http://dx.doi.org/http://doi.acm.org/10.1145/860575.860701.

[Burks, 1971] Arthur Walter Burks (1971). *Essays on Cellular Automata*. University of Illinois Press. http://www.amazon.fr/exec/obidos/ASIN/0252000234/citeulike04-21.

[Bäck et al., 1997] Thomas Bäck, Ulrich Hammel, et Hans Paul Schwefel (1997). Evolutionary computation: Comments on the history and current state. *IEEE Trans. Evolutionary Computation*, 1(1):3–17. http://citeseer.ist.psu.edu/601414.html.

[Büche et al., 2004] Dirk Büche, Nicol N. Schraudolph, et Petros Koumoutsakos (2004). Accelerating evolutionary algorithms with gaussian process fitness function models. *Systems, Man and Cybernetics, Part C, IEEE Transactions on*, 35(2):183–194. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1424193. http://dx.doi.org/10.1109/TSMCC.2004.841917.

[Calderoni, 2002] Stéphane Calderoni (2002). *Éthologie Artificielle et Contrôle Auto-Adaptatif dans les Systèmes d'Agents Réactifs*. Thèse, Université de La Réunion.

[Calvez, 2004] Benoît Calvez (2004). Exploration de l'espace des paramètres d'une simulation multiagent. Rapport de Master, Université d'Évry Val d'Essonne.

[Cantu-Paz et Goldberg, 2000] Erick Cantu-Paz et David E. Goldberg (2000). Efficient parallel genetic algorithms: theory and practice. *Computer Methods in Applied Mechanics and Engineering*, 186(2-4):221–238. http://www.sciencedirect.com/science/article/B6V29-40CRYKF-6/2/9ec1963003d8d255648b3d7878df6272.

[Cantú-Paz, 2004] Erick Cantú-Paz (2004). Adaptive sampling for noisy problems. In *Genetic and Evolutionary Computation – GECCO 2004*, lncs, pages 947–958. Springer. http://www.springerlink.com/content/e1237bu775d7ulr1.

[Castle et Crooks, 2006] Christian Castle et Andrew Crooks (2006). Principles and concepts of agent-based modelling for developing geospatial simulations. Technical report, Centre for Advanced Spatial Analysis, University College London.

[Cattaneo et al., 2006] Gianpiero Cattaneo, Alberto Dennunzio, et Fabio Farina (2006). A full cellular automaton to simulate predator-prey systems. In *7th International Conference on Cellular Automata, for Research and Industry, ACRI 2006*, pages 446–451. El Yacoubi, Samira and Chopard, Bastien and Bandini, Stefania. http://dx.doi.org/10.1007/11861201_52.

[Chiaberge et al., 1994] M. Chiaberge, J.J. Merelo, L.M. Reyneri, A. Prieto, et L. Zocca (1994). A comparison of neural networks, linear controllers, genetic algorithms and simulated annealing

for real time control. In *ESANN'1994 proceedings - European Symposium on Artificial Neural Networks*, pages 205–210. http://polimage.polito.it/~marcello/pubbli.htmlhttp://www.dice.ucl.ac.be/esann/proceedings/papers.php?ann=1994.

[Coquillard et Hill, 1997] Patrick Coquillard et David Hill (1997). *Modélisation et simulation d'écosystèmes*. MASSON.

[Cramer, 1985] Nichael Lynn Cramer (1985). A representation for the adaptive generation of simple sequential programs. In John J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA. http://www.sover.net/~nichael/nlc-publications/icga85/index.html.

[De Jong, 1999] Ken De Jong (1999). Genetic algorithms: A 30 year perspective. Festschrift in honor of John H. Holland. http://www.cscs.umich.edu/jhhfest/.

[De Wolf et al., 2005] T. De Wolf, G. Samaey, T. Holvoet, et D. Roose (2005). Decentralised autonomic computing: Analysing self-organising emergent behaviour using advanced numerical methods. In *ICAC 2005*, pages 52–63. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1498052.

[Deneubourg et al., 1990] Deneubourg, S. Aron, S. Goss, et J. M. Pasteels (1990). The self-organizing exploratory pattern of the argentine ant. *Journal of Insect Behavior*, 3(2):159–168. http://dx.doi.org/10.1007/BF01417909.

[Dorigo et al., 2006] Marco Dorigo, Mauro Birattari, et Thomas Stützle (2006). Ant colony optimization–artificial ants as a computational intelligence technique. Technical report, IRIDIA, Institut de Recherches Interdisciplinaires IRIDIA UNIVERSITE LIBRE DE BRUXELLES. http://iridia.ulb.ac.be/IridiaTrSeries/IridiaTr2006-023r001.pdf.

[Dorigo et Blum, 2005] Marco Dorigo et Christian Blum (2005). Ant colony optimization theory: A survey. *Theoretical Computer Science*, 344(2-3):243–278. http://www.sciencedirect.com/science/article/B6V1G-4GSJTVN-2/2/2c61a7f14dd9850df1982ca35800c0d9. http://dx.doi.org/http://dx.doi.org/10.1016/j.tcs.2005.05.020.

[Dorigo et Di Caro, 1999] Marco Dorigo et Gianni Di Caro (1999). *The Ant Colony Optimization Meta-Heuristic*, chapter New Ideas in Optimization, pages 11–32. McGraw-Hill. http://www.idsia.ch/~gianni/Papers/OptBook.ps.gz.

[Dorigo et al., 1999] Marco Dorigo, Gianni Di Caro, et Luca M. Gambardella (1999). Ant algorithms for discrete optimization. *Artificial Life*, 5(2):137–172. ftp://iridia.ulb.ac.be/pub/mdorigo/journals/IJ.23-alife99.pdf.

[Dorigo et al., 1991] Marco Dorigo, Vittorio Maniezzo, et Alberto Colorni (1991). Positive feedback as a search strategy. *Technical Report No. 91-016, Politecnico di Milano, Italy, 1991*. http://iridia.ulb.ac.be/~mdorigo/ACO/publications.html.

[Dorigo et al., 1996] Marco Dorigo, Vittorio Maniezzo, et Alberto Colorni (1996). The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics*, 26(1):29–41. citeseer.ist.psu.edu/dorigo96ant.html.

[Dorigo et of Socha, 2007] Marco Dorigo et Krzyszt of Socha (2007). *Approximation Algorithms and Metaheuristics*, chapter An Introduction to Ant Colony Optimization. CRC Press. http://iridia.ulb.ac.be/IridiaTrSeries/IridiaTr2006-010r001.pdf.

[Dorigo et Stützle, 2002] Marco Dorigo et Thomas Stützle (2002). *The Ant Colony Optimization Metaheuristic: Algorithms, Applications, and Advances*, chapter Handbook of Metaheuristics. Springer. http://iridia.ulb.ac.be/~stuetzle/publications/ACO.ps.gz. http://dx.doi.org/10.1007/0-306-48056-5_9.

[Drogoul, 1993] Alexis Drogoul (1993). *De La Simulation Multi-Agent A La Résolution Collective de Problèmes*. Thèse, Université Paris VI.

[Duboz, 2004] Raphaël Duboz (2004). *Intégration de modèles hétérogènes pour a modélisation et la simulation de systèmes complexes*. Thèse, Université du Littoral - Côte d'Opale.

[Edmonds et Bryson, 2004] Bruce Edmonds et Joanna J. Bryson (2004). The insufficiency of formal design methods - the necessity of an experimental approach for the understanding and control of complex mas. pages 938–945. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1373612.

[Epstein et Axtell, 1996] Joshua M. Epstein et Robert L. Axtell (1996). *Growing Artificial Societies: Social Science from the Bottom Up*. The MIT Press. http://www.amazon.ca/exec/obidos/redirect?tag=citeulike04-20{\&}path=ASIN/0262550253.

[Fehler et al., 2004] Manuel Fehler, Franziska Klügl, et Frank Puppe (2004). Techniques for analysis and calibration of multi-agent simulations. In Marie Pierre Gleizes, Andrea Omicini, et Franco Zambonelli, editeur, *ESAW*, volume 3451 of *Lecture Notes in Computer Science*, pages 305–321. Springer. http://www.irit.fr/ESAW04/PAPERS/ESAW04_Paper13.pdf. http://dx.doi.org/http://dx.doi.org/10.1007/11423355_22.

[Fehler et al., 2006] Manuel Fehler, Franziska Klügl, et Frank Puppe (2006). Approaches for resolving the dilemma between model structure refinement and parameter calibration in agent-based simulations. In Hideyuki Nakashima, Michael P. Wellman, Gerhard Weiss, et Peter Stone, editeur, *AAMAS*, pages 120–122. ACM. http://portal.acm.org/citation.cfm?doid=1160633.1160651. http://dx.doi.org/http://doi.acm.org/10.1145/1160633.1160651.

[Ferber, 1995] Jacques Ferber (1995). *Les Systèmes multi-agents: Vers une intelligence collective*. InterEditions. http://www.amazon.fr/exec/obidos/ASIN/2729606653/citeulike04-21.

[Franconi et Jennison, 1997] Luisa Franconi et Christopher Jennison (1997). Comparison of a genetic algorithm and simulated annealing in an application to statistical image reconstruction. *Statistics and Computing*, 7(3):193–207. http://dx.doi.org/10.1023/A:1018538202952.

[Franklin et Graesser, 1997] Stan Franklin et Art Graesser (1997). Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Intelligent Agents III Agent Theories, Architectures, and Languages*, pages 21–35. Springer. http://dx.doi.org/10.1007/BFb0013570.

[Ganguly et al., 2003] Niloy Ganguly, Biplab K. Sikdar, Andreas Deutsch, Geoffrey Canright, et P. Pal Chaudhuri (2003). A survey on cellular automata. Technical report, Centre for High Performance Computing, Dresden University of Technology.

[Gardner, 1970] M. Gardner (1970). The fantastic combinations of john conway's new solitaire game "life". *Scientific American*, 223:120–123.

[Ghosh et al., 2005] Ranadhir Ghosh, Moumita Ghosh, John Yearwood, et Adil Bagirov (2005). Comparative analysis of genetic algorithm, simulated annealing and cutting angle method for artificial neural networks. In *Machine Learning and Data Mining in Pattern Recognition*, pages 62–70. J. G. Carbonell and J. Siekmann. http://dx.doi.org/10.1007/11510888_7.

[Gilbert et Bankes, 2002] Nigel Gilbert et Steven Bankes (2002). Platforms and methods for agent-based modeling. *Proceedings of the National Academy of Sciences*, 99:7197–7198. http://www.pnas.org/cgi/reprint/99/suppl_3/7197. http://dx.doi.org/10.1073/pnas.072079499.

[Ginot et Monod, 2006] Vincent Ginot et Hervé Monod (2006). *Modélisation et simulation multi-agents, applications pour les Sciences de l'Homme et de la Société*, chapter Explorer les modèles par simulation : application aux analyses de sensibilité, pages 61–87. Hermes-Sciences & Lavoisier.

[Glover, 1986] Fred Glover (1986). Future paths for integer programming and links to artificial intelligence. *Comput. Oper. Res.*, 13(5):533–549. http://dx.doi.org/10.1016/0305-0548(86)90048-1.

[Goldberg et al., 1992] David E. Goldberg, Kalyanmoy Deb, et James H. Clark (1992). Genetic algorithms, noise, and the sizing of populations. *complex systems*, 6(4):333–362.

[Gonçalo, 2005] Neto Gonçalo (2005). From single-agent to multi-agent reinforcement learning: Foundational concepts and methods. Instituto de Sistemas e Robotica Instituto Superior Tecnico. Learning Theory Course, http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/.

[Goss et al., 1989] S. Goss, S. Aron, J. Deneubourg, et J. Pasteels (1989). Self-organized shortcuts in the argentine ant. *Naturwissenschaften*, 76(12):579–581. http://dx.doi.org/10.1007/BF00462870. http://dx.doi.org/10.1007/BF00462870.

[Grassé, 1959] Plerre-P Grassé (1959). La reconstruction du nid et les coordinations interindividuelles chez bellicositermes natalensis etcubitermes sp. la théorie de la stigmergie: Essai d'interprétation du comportement des termites constructeurs. *Insectes Sociaux*, 6(1):41–80. http://dx.doi.org/10.1007/BF02223791.

[Greening, 1990] Daniel R. Greening (1990). Parallel simulated annealing techniques. *Physica D: Nonlinear Phenomena*, 42(1-3):293–306. http://www.sciencedirect.com/science/article/B6TVK-46G4VN3-W/2/8a8fbd42950aa1aa900e8abb321b7578.

[Hare et Deadman, 2004] M. Hare et P. Deadman (2004). Further towards a taxonomy of agent-based simulation models in environmental management. *Mathematics and Computers in Simulation*, 64(1):25–40. http://dx.doi.org/10.1016/S0378-4754(03)00118-6.

[Ho et Pepyne, 2002] Y. C. Ho et D. L. Pepyne (2002). Simple explanation of the no-free-lunch theorem and its implications. *Journal of Optimization Theory and Applications*, V115(3):549–570. http://dx.doi.org/10.1023/A:1021251113462.

[Holland, 1975] John Holland (1975). *Adaptation In Natural and Artificial Systems.* University of Michigan Press.

[Holland, 1962] John H. Holland (1962). Outline for a logical theory of adaptive systems. *J. ACM*, 9(3):297–314. http://dx.doi.org/10.1145/321127.321128.

[Hutzler, 2007] Guillaume Hutzler (2007). Cellular automata and agent-based approaches for the modelling and simulation of biological systems: application to the lambda phage. In *Modeling Complex Systems in the Context of Genomics - Tutorial on Modeling*, pages 215–232. Evry Spring School 2007.

[Ingu et Takagi, 1999] T. Ingu et H. Takagi (1999). Accelerating a ga convergence by fitting a single-peak function. *Fuzzy Systems Conference Proceedings, 1999. FUZZ-IEEE '99. 1999 IEEE International*, 3:1415–1420 vol.3. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=790111.

[Jin, 2005] Y. Jin (2005). A comprehensive survey of fitness approximation in evolutionary computation. *Soft Comput.*, 9(1):3–12. http://dx.doi.org/10.1007/s00500-003-0328-5.

[Jin et al., 2000] Yaochu Jin, Markus Olhofer, et Bernhard Sendhoff (2000). On evolutionary optimization with approximate fitness functions. In L. Darrell Whitley, David E. Goldberg, Erick Cantú-Paz, Lee Spector, Ian C. Parmee, et Hans-Georg Beyer, editeur, *Proceedings of the Genetic and Evolutionary Computation Conference GECCO*, pages 786–793. Morgan Kaufmann. 1-55860-708-0, http://www.honda-ri.org/intern/literature/hriliterature/Jin00/view.

[Jin et al., 2002] Yaochu Jin, M. Olhofer, et B. Sendhoff (2002). A framework for evolutionary optimization with approximate fitness functions. *Evolutionary Computation, IEEE Transactions on*, 6(5):481–494. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1041556. http://dx.doi.org/10.1109/TEVC.2002.800884.

[Kaelbling et al., 1996] Leslie Pack Kaelbling, Michael L. Littman, et Andrew W. Moore (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285.

[Kallel et Schoenauer, 1997] Leila Kallel et Marc Schoenauer (1997). Alternative random initialization in genetic algorithms. In Thomas Bäck, editor, *Proceedings of the 7th International Conference on Genetic Algorithms (ICGA)*, pages 268–275. Morgan Kaufmann.

[Keerthi et Ravindran, 1995] S. Keerthi et B. Ravindran (1995). A tutorial survey of reinforcement learning. http://citeseer.ist.psu.edu/50807.html.

[Kevrekidis et al., 2004] Ioannis G. Kevrekidis, William C. Gear, et Gerhard Hummer (2004). Equation-free: The computer-aided analysis of complex multiscale systems. *AIChE Journal*, 50(7):1346–1355. http://dx.doi.org/10.1002/aic.10106.

[Kirkpatrick et al., 1983] S. Kirkpatrick, C. D. Gelatt, et M. P. Vecchi (1983). Optimization by simulated annealing. *Science*, 220(4598):671–680. http://dx.doi.org/10.1126/science.220.4598.671.

[Klein et al., 2005] François Klein, Christine Bourjot, et Vincent Chevrier (2005). Dynamic design of experiment with MAS to approximate the behavior of complex systems. In *Multi-Agents for modeling Complex Systems (MA4CS'05) Satellite Workshop of the European Conference on Complex Systems (ECCS'05)*. http://hal.inria.fr/inria-00000805/en/.

[Klein et al., 2006] François Klein, Christine Bourjot, et Vincent Chevrier (2006). Approche expérimentale pour la compréhension des systèmes multi-agents réactifs. In *Journées Francophones sur les Systèmes Multi-Agents - JFSMA 2006*. http://hal.inria.fr/inria-00104308/en/.

[Klein, 2002] John Klein (2002). breve: a 3d simulation environment for multi-agent simulations and artificial life. www.spiderland.org.

[Koza, 1992] John R. Koza (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

[Koza et al., 1992] John R. Koza, Jonathan Roughgarden, et James P. Rice (1992). Evolution of food-foraging strategies for the caribbean anolis lizard using genetic programming. *Adaptive Behavior*, 1(2):171–199. http://dx.doi.org/10.1177/105971239200100203.

[Krishnan et Purdy, 2006] Rajesh Krishnan et Carla C. Purdy (2006). Comparison of simulated annealing and genetic algorithm for characterizing and controlling biological pathways. In *Proceeding ANNIE 2006*. ASME Press.

[Lahtinen et al., 1996] Jussi Lahtinen, Petri Myllymäki, Tomi Silander, et Henry Tirri (1996). Empirical comparison of stochastic algorithms in a graph optimization problem. In Jarmo T. Alander, editor, *Proceedings of the Second Nordic Workshop on Genetic Algorithms and their Applications (2NWGA)*, Proceedings of the University of Vaasa, Nro. 13, pages 45–60, Vaasa (Finland). University of Vaasa. ftp://ftp.uwasa.fi/cs/2NWGA/*.ps.Z.

[Lotka, 1925] Alfred James Lotka (1925). *Elements of Physical Biology*. Dover Publication.

[Luke et al., 2005] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, et Gabriel Balan (2005). Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527. http://dx.doi.org/10.1177/0037549705058073.

[Macal et North, 2005] Charles M. Macal et Michael J. North (2005). Tutorial on agent-based modeling and simulation. In *WSC '05: Proceedings of the 37th conference on Winter simulation*, pages 2–15. Winter Simulation Conference.

[Manikas et Cain, 1996] Theodore W. Manikas et James T. Cain (1996). Genetic algorithms vs. simulated annealing: A comparison of approaches for solving the circuit partitioning problem. Technical report, Department of Electrical Engineering, The University of Pittsburgh. www.ee.utulsa.edu/~tmanikas/Pubs/gasa-TR-96-101.pdf.

[Marco-Blaska et Désidéri, 1999] Nathalie Marco-Blaska et Jean-Antoine Désidéri (1999). Numerical solution of optimization test-cases by genetic algorithms. Technical report, Inria. ftp://ftp-sop.inria.fr/pastis/RR/RR-3622.ps.gz.

[Metropolis et al., 1953] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, et Edward Teller (1953). Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092. http://scitation.aip.org/getabs/servlet/GetabsServlet?prog=normal\&id=JCPSA6000021000006001087000001\&idtype=cvips\&gifs=yes. http://dx.doi.org/10.1063/1.1699114.

[Minar et al., 1996] N. Minar, R. Burkhart, C. Langton, et M. Askenazi (1996). The swarm simulation sys tem: a toolkit for building muli-agent simulations. RReport 96-06-042. Sante Fe Institute, www.swarm.org.

[Minsky, 1965] Marvin L. Minsky (1965). Matter, mind and models. In *Proceeding International Federation of Information Processing Congress*, pages 45–49.

[MIT, 2000] MIT (2000). Starlogo. MIT, http://education.mit.edu/starlogo/.

[Mitchell, 1996] Colin G. Mitchell (1996). Identification of a multienzyme complex of the tricarboxylic acid cycle enzymes containing citrate synthase isoenzymes from *Pseudomonas aeruginosa*. *Biochemical Journal*, 313:769–774. www.biochemj.org/bj/313/0769/3130769.pdf, www.biochemj.org/bj/313/0769/3130769.pdf.

[Mitchell et Taylor, 1999] Melanie Mitchell et Charles E. Taylor (1999). Evolutionary computation: An overview. *Annual review of ecology and systematics*, 30:593–616. http://cat.inist.fr/?aModele=afficheN&cpsidt=1229427.

[Moncion et al., 2006] Thomas Moncion, Guillaume Hutzler, et Patrick Amar (2006). Verification of biochemical agent-based models using petri nets. In *ABModSim 2006 (International Symposium on Agent Based Modeling and Simulation)*.

[Narzisi et al., 2006] Giuseppe Narzisi, Venkatesh Mysore, et Bud Bud Mishra (2006). Multi-objective evolutionary optimization of agent-based models: An application to emergency response planning. In B. Kovalerchuk, editor, *The IASTED International Conference on Computational Intelligence (CI 2006)*. http://www.actapress.com/Abstract.aspx?paperId=29076.

[North et al., 2005] M.J. North, T.R. Howe, N.T. Collier, et J.R. Vos (2005). Repast simphony runtime system. In *Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*. http://repast.sourceforge.net/.

[opttek, 1992] opttek (1992). Optquest®. http://www.opttek.com/products/optquest.html.

[Panait et Luke, 2005] Liviu Panait et Sean Luke (2005). Cooperative multi-agent learning: The state of the art. *Autonomous Agents and Multi-Agent Systems*, 11(3):387–434. http://dx.doi.org/10.1007/s10458-005-2631-2.

[Parisey et al., 2007] Nicolas Parisey, Marie Beurton-Aimar, et Charles Lalès (2007). Cookbook to design mas for molecular simulations. In *Modeling Complex Systems in the Context of Genomics - Tutorial on Modeling*, pages 215–232. Evry Spring School 2007.

[Parker et al., 2003] Dawn C. Parker, Steven M. Manson, Marco A. Janssen, Matthew J. Hoffmann, et Peter Deadman (2003). Multi-agent systems for the simulation of land-use and land-cover change: A review. *Annals of the Association of American Geographers*, 93(2):314–337. http://dx.doi.org/10.1111/1467-8306.9302004.

[Parunak et al., 1998] H. Van Dyke Parunak, Robert Savit, et Rick L. Riolo (1998). Agent-based modeling vs. equation-based modeling: A case study and users' guide. pages 10–25. http://dx.doi.org/10.1007/10692956_2.

[Pritsker, 1979] A. Alan B. Pritsker (1979). Compilation of definitions of simulation. *SIMULATION*, 33(2):61–63. http://sim.sagepub.com. http://dx.doi.org/10.1177/003754977903300205.

[Project, 2007] The Wolfram Demonstrations Project (2007). 3d boid model. The Wolfram Demonstrations Project, http://demonstrations.wolfram.com/3DBoidModel/.

[Railsback et al., 2006a] Steven F. Railsback, Steven L. Lytinen, et Stephen K. Jackson (2006a). Agent-based simulation platforms: Review and development recommendations. Revised manuscript submitted to Simulation, July 2006, http://www.humboldt.edu/~ecomodel/documents/ABMPlatformReview.pdf.

[Railsback et al., 2006b] Steven F. Railsback, Steven L. Lytinen, et Stephen K. Jackson (2006b). Agent-based simulation platforms: Review and development recommendations. *SIMULATION*, 82(9):609–623. http://dx.doi.org/10.1177/0037549706073695.

[Ram et al., 1996] Janaki D. Ram, T. H. Sreenivas, et Ganapathy K. Subramaniam (1996). Parallel simulated annealing algorithms. *J. Parallel Distrib. Comput.*, 37(2):207–212. http://www.sciencedirect.com/science/article/B6WKJ-45MG45T-1M/2/de7c71de0ef3474a7db5561ca330a281. http://dx.doi.org/10.1006/jpdc.1996.0121.

[Rechenberg, 1973] Ingo Rechenberg (1973). *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog-Verlag.

[Reynolds, 1987] Craig W. Reynolds (1987). Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, volume 21, pages 25–34. ACM Press. http://portal.acm.org/citation.cfm?id=37406. http://dx.doi.org/10.1145/37401.37406.

[Rogers et von Tessin, 2004] Alex Rogers et Peter von Tessin (2004). Multi-objective calibration for agent-based models. In *5th Workshop on Agent-Based Simulation*. http://eprints.ecs.soton.ac.uk/11547/.

[Sallans et al., 2003] Brian Sallans, Alexander Pfister, Alexandros Karatzoglou, et Georg Dorffner (2003). Simulation and validation of an integrated markets model. *J. Artificial Societies and Social Simulation*, 6(4). http://jasss.soc.surrey.ac.uk/6/4/2.html.

[Schwefel, 1979] Hans-Paul Schwefel (1979). Direct search for optimal parameters within simulation models. In *ANSS '79: Proceedings of the 12th annual symposium on Simulation*, pages 91–102, Piscataway, NJ, USA. IEEE Press. http://portal.acm.org/citation.cfm?id=800246.807345.

[Schwefel et Rudolph, 1995] Hans-Paul Schwefel et Günter Rudolph (1995). Contemporary evolution strategies. In *Proceedings of the Third European Conference on Advances in Artificial Life*, pages 893–907, London, UK. Springer-Verlag. http://portal.acm.org/citation.cfm?id=648349.

[Schwefel, 1993] Hans-Paul P. Schwefel (1993). *Evolution and Optimum Seeking: The Sixth Generation*. John Wiley & Sons, Inc., New York, NY, USA. http://portal.acm.org/citation.cfm?id=529401.

[Servat, 2000] David Servat (2000). *Modélisation de dynamiques de flux par agents*. Thèse, Université de Paris6.

[Shnerb et al., 2000] Nadav M. Shnerb, Yoram Louzoun, Eldad Bettelheim, et Sorin Solomon (2000). The importance of being discrete: Life always wins on the surface. *PNAS*, 97(19):10322–10324.

http://view.ncbi.nlm.nih.gov/pubmed/10962027. http://dx.doi.org/10.1073/pnas.180263697.

[Shoham et al., 2003] Yoav Shoham, Rob Powers, et Trond Grenager (2003). Multi-agent reinforcement learning: a critical survey. Technical report, Stanford University.

[Sinha et Goldberg, 2003] Abhishek Sinha et David E. Goldberg (2003). A survey of hybrid genetic and evolutionary algorithms. Technical report, Illinois Genetic Algorithms Laboratory. http://www.illigal.uiuc.edu/web/technical-reports/2003/04/04/a-survey-of-hybrid-genetic-and-

[Socha et Dorigo, 2006] Krzysztof Socha et Marco Dorigo (2006). Ant colony optimization for continuous domains. In *EURO XXI in Iceland*. https://www.euro-online.org/euro21/display.php?page=treate_abstract&frompage=edit_session_cluster&paperid=2096.

[Stockholm et al., 2007] Daniel Stockholm, Rachid Benchaouir, Julien Julien Picot, Philippe Philippe Rameau, Thi My Anh Neildez, Gabriel Landini, Corinne Corinne Laplace-Builhé, et Andras Paldi (2007). The origin of phenotypic heterogeneity in a clonal cell population in vitro. *PLoS ONE*, 4(2):e394. http://www.plosone.org/article/fetchArticle.action?articleURI=info:doi/10.1371/journal.pone.0000394. http://dx.doi.org/10.1371/journal.pone.0000394.

[Stone et Veloso, 2000] Peter Stone et Manuela Veloso (2000). Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383. http://dx.doi.org/10.1023/A:1008942012299.

[Sutton, 1988] Richard S. Sutton (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44. http://portal.acm.org/citation.cfm?id=637937. http://dx.doi.org/10.1023/A:1022633531479.

[Takadama et al., 2003] Keiki Takadama, Yutaka L. Suematsu, Norikazu Sugimoto, Norberto Eiji Nawa, et Katsunori Shimohara (2003). Cross-element validation in multiagent-based simulation: Switching learning mechanisms in agents. *J. Artificial Societies and Social Simulation*, 6(4).

[Technologies, 2007] XJ Technologies (2007). Anylogic 6. http://www.xjtek.com/anylogic/.

[Terán et Edmonds, 2002] Oswaldo Terán et Bruce Edmonds (2002). Computational complexity of a constraint model-based proof of the envelope of tendencies in a mas-based simulation model. In *2nd International Workshop on Complexity in Automated Deduction (CiAD)*.

[Terán et Edmonds, 2004] Oswaldo Terán et Bruce Edmonds (2004). Constraint model-based exploration of simulation trajectories in a mabs model. In *18th Workshop on (Constraint) Logic Programming*. http://cfpm.org/cpmrep161.html.

[Thellier et al., a] Michel Thellier, Vic Norris, Patrick Amar, Christophe Baron, et Camille Ripoll. Dynamics of intracelullar functioning-dependent structures.

[Thellier et al., b] Michel Thellier, Vic Norris, Christophe Baron, et Camille Ripoll. Introduction to the concept of functioning-dependent structures.

[Tobias et Hofmann, 2004] Robert Tobias et Carole Hofmann (2004). Evaluation of free java-libraries for social-scientific agent based simulation. *Journal of Artificial Societies and Social Simulation*, 7(1). http://jasss.soc.surrey.ac.uk/7/1/6.html.

[Troitzsch, 2004] Klaus G. Troitzsch (2004). Validating simulation models. In Graham Horton, editor, *18th European Simulation Multiconference. Networked Simulations and Simulation Networks*, pages 265–270. The Society for Modeling and Simulation International, SCS Publishing House. http://www.uni-koblenz.de/~kgt/EPOS/RevisedAbstracts/Troitzsch.pdf.

[Universität Würzburg, 2002] Informatik VI Universität Würzburg (2002). Sesam : 'shell for simulated agent systems'. www.simsesam.de/.

[Vanier et Bower, 1999] Michael C. Vanier et James M. Bower (1999). A comparative survey of automated parameter-search methods for compartmental neural models. *Journal of Computational Neuroscience*, 7(2):149–171. http://dx.doi.org/10.1023/A:1008972005316.

[Varenne, 2001] Franck Varenne (2001). What does a computer simulation prove? In Society for Computer Simulation (SCS), editor, *Simulation in Industry - Proc. of the 13th European Simulation Symposium*, pages 549–554. 2001, http://hal.archives-ouvertes.fr/hal-00004125/en/.

[Volterra, 1926] Vito Volterra (1926). Fluctuations in the abundance of a species considered mathematically. *Nature*, 188:558–560.

[Von Neumann, 1966] John Von Neumann (1966). *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA. http://portal.acm.org/citation.cfm?id=1102024.

[Watkins et Dayan, 1992] Christopher J. C. H. Watkins et Peter Dayan (1992). Technical note: Q-learning. *Machine Learning*, V8(3):279–292. http://dx.doi.org/10.1023/A:1022676722315.

[Whitley, 1994] Darrell Whitley (1994). A genetic algorithm tutorial. *Statistics and Computing*, 4(2):65–85. http://dx.doi.org/10.1007/BF00175354.

[Whitley et Kauth, 1988] D. Whitley et J. Kauth (1988). GENITOR: A different genetic algorithm. In *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, pages 118–130. Denver, CO.

[Whitley, 1989] Darrell L. Whitley (1989). The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 116–123, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. http://portal.acm.org/citation.cfm?id=657257.

[Wikipedia, 2007] Wikipedia (2007). Systèmes complexes. http://fr.wikipedia.org/wiki/Systèmes_complexes.

[Wilensky, 1998a] Uri Wilensky (1998a). Netlogo ants model. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL, http://ccl.northwestern.edu/netlogo/models/Ants.

[Wilensky, 1998b] Uri Wilensky (1998b). Netlogo cooperation model. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL., http://ccl.northwestern.edu/netlogo/models/Cooperation.

[Wilensky, 1998c] Uri Wilensky (1998c). Netlogo divide the cake model. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL, http://ccl.northwestern.edu/netlogo/models/DivideTheCake.

[Wilensky, 1998d] Uri Wilensky (1998d). Netlogo flocking model. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL., http://ccl.northwestern.edu/netlogo/models/Flocking.

[Wilensky, 1998e] Uri Wilensky (1998e). Netlogo shepherds model. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL, http://ccl.northwestern.edu/netlogo/models/Shepherds.

[Wilensky, 1998f] Uri Wilensky (1998f). Netlogo traffic 2 lanes model. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL., http://ccl.northwestern.edu/netlogo/models/Traffic2Lanes.

[Wilensky, 1998g] Uri Wilensky (1998g). Netlogo wolf sheep predation model. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL., http://ccl.northwestern.edu/netlogo/models/WolfSheepPredation.

[Wilensky, 1999] Uri Wilensky (1999). Netlogo. Center for Connected Learning and Computer-Based Modeling. Northwestern University, Evanston, IL, http://ccl.northwestern.edu/netlogo.

[Wilensky et Reisman, 2006] Uri Wilensky et Kenneth Reisman (2006). Thinking like a wolf, a sheep, or a firefly: Learning biology through constructing and testing. *Cognition and Instruction*, 24(2):171–209.

[Wilson, 1998] William G. Wilson (1998). Resolving discrepancies between deterministic population models and individual-based simulations. *The American Naturalist*, 151(2):116–134. http://www.journals.uchicago.edu/cgi-bin/resolve?id=doi:10.1086/286106.

[Windrum et al., 2007] Paul Windrum, Giorgio Fagiolo, et Alessio Moneta (2007). Empirical validation of agent-based models: Alternatives and prospects. *Journal of Artificial Societies and Social Simulation*, 10(2):8. http://jasss.soc.surrey.ac.uk/10/2/8.html.

[Wolpert et Macready, 1995] David H. Wolpert et William G. Macready (1995). No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute, Santa Fe, NM. http://citeseer.ist.psu.edu/75066.html.

[Wolpert et Macready, 1997] D. H. Wolpert et W. G. Macready (1997). No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1):67–82. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=585893.

[Wooldridge et Jennings, 1995a] Michael Wooldridge et Nicholas Jennings (1995a). Agent theories, architectures, and languages: A survey. In *Intelligent Agents*, pages 1–39. Springer. http://dx.doi.org/10.1007/3-540-58855-8_1.

[Wooldridge et Jennings, 1995b] Michael Wooldridge et Nicholas R. Jennings (1995b). Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152. citeseer.ist.psu.edu/article/wooldridge95intelligent.html.

[Xu et al., 2003] Jin Xu, Yongqin Gao, Jeffrey Goett, et Gregory Madey (2003). A multi-model docking experiment of dynamic social network simulations. In *Agent 2003*. http://www.nd.edu/~oss/Papers/papers.html.

Annexes

A Plates-formes multi-agents

Nous présentons dans cette annexe un petit tour d'horizon de différentes plates-formes de simulation multi-agent. Le but est de montrer quelques exemples de plates-formes existantes avec leurs principales caractéristiques, et d'expliquer notre choix de plate-forme de test. Pour plus de détails sur les plates-formes multi-agents existantes, il faut mieux se reporter à des articles de synthèse [Gilbert et Bankes, 2002, Tobias et Hofmann, 2004, Castle et Crooks, 2006, Railsback et al., 2006a, Parisey et al., 2007, Railsback et al., 2006b], et plus particulièrement au dernier article cité : Railsback et ses co-auteurs ont développé un modèle multi-agent « *StupidModel* »[1] afin de comparer les différentes plates-formes en exhibant les différents fonctionnalités. Ce modèle est une référence pour les plates-formes de simulation à base d'agents.

Nous nous focaliserons sur le modèle « *Boids* » de Reynolds [Reynolds, 1987] pour l'étude de quelques plates-formes : ce modèle est en effet très couramment utilisé, et généralement il fait parti des modèles exemples fournis par les simulateurs.

Tout d'abord, la première question est de se demander pourquoi utiliser une plate-forme de simulation, et non des outils informatiques traditionnels. Par exemple, le modèle « *Boids* » a été implanté sous `Mathematica 6.0` [Project, 2007]. La figure A.1 montre l'interface graphique du modèle, et le listing A.1 montre le début du code. Cependant, ces outils, comme `Mathematica`, n'ont pas été conçus pour la conception de modèles à base d'agents : il faut mieux privilégier les plates-formes dédiées à la simulation multi-agent déjà disponibles.

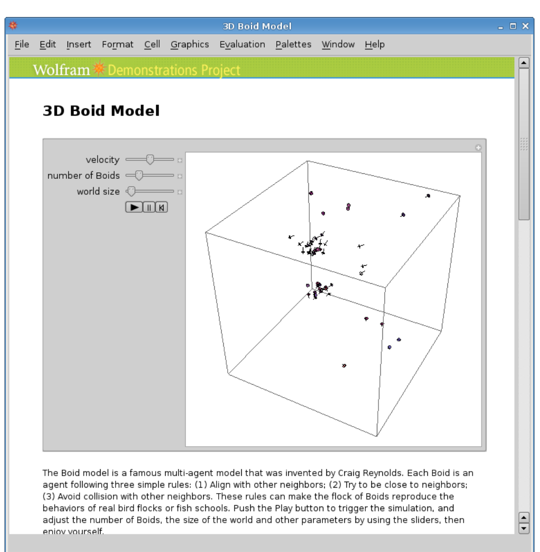


FIG. A.1 – *Le modèle « Boids » sous* `Mathematica`

Voici une liste des principales plates-formes multi-agents (examinées sous un environnement `linux` avec la distribution *Ubuntu Edgy 6.10* avec une installation standard (avec `java` en version 1.6)) :

– `Mason` [Luke et al., 2005] (testé en version 12) : `Mason` (« *Multi-Agent Simulator Of Neighborhoods... or Networks... or something...* ») est une plate-forme de simulation codée en java. Elle est incompatible avec `Java 1.6` (elle est conçu pour du `Java 1.3`.). Elle intègre un modèle « *Flockers* » de démonstration. Une différence par rapport aux autres plates-formes est que l'utilisateur doit implanter ces modèles en java en héritant de classes pré-définies. La figure A.2 présente l'interface du logiciel, et le listing A.2 montre un bout du code du modèle « *Flockers* ».

Listing A.1 *Début du code source du modèle « Boids » sous* `Mathematica`

```
3D Boid Model
Initialize[] := Module[{},
   vR = 2;(*The speed of each boid*)
   Size = 100;(*The size of the world*)
   Init =
    Table[{{RandomReal[{0, Size}], RandomReal[{0, Size}],
       RandomReal[{0, Size}]},
      vR {Cos[#1] Cos[#2], Cos[#1] Sin[#2],
          Sin[#2]} & @@ {RandomReal[{0, 2*Pi}],
        RandomReal[{0, 2*Pi}]}}, {i, 1, 50}];
   (*Initialize all positions and velocities of boids*)
   rNeighbor = 10;(*Neighbor distance*)
   rCollision = 5;(*Collision distance*)
   MySteps = 0;(*Simulation steps*)
   States =
    Init;(*Simulation states of positions and velocities of each boid \
in every step*)
   ];
OneStep[x_List] := Module[{Neighbors, Collisions, vNew, zero, state},
   (*The main loop of the simulation*)
   zero = Table[0, {3}];
   Neighbors =
    Table[Join[{x[[i]]},
      Select[x,
       EuclideanDistance[First[#], First[x[[i]]]] < rNeighbor &&
         EuclideanDistance[First[#], First[x[[i]]]] >
          rCollision &]], {i, 1, Length[x]}];
   Collisions =
    Table[Join[{x[[i]]},
      Select[x,
       EuclideanDistance[First[#], First[x[[i]]]] < rCollision &&
         First[#] != First[x[[i]]] &]], {i, 1, Length[x]}];
   vNew = (0.3 If[Length[#] > 1,
           Normalize[Total[Last /@ Rest[#]]/(Length[#] - 1)], zero] +
         0.2 If[Length[#] > 1,
           Normalize[(Total[First /@ Rest[#]]/(Length[#] - 1)) -
             First[First[#]]], zero]) & /@ Neighbors +
     0.2 Last /@
       x + (0.3 If[Length[#] > 1,
          Normalize[
           First[First[#]] - Total[First /@ Rest[#]]/(Length[#] - 1)],
           zero] & /@ Collisions);
   vNew = vR Normalize /@ vNew;
   state =
    Transpose[{Mod[# + Size, Size] & /@ ((First[#] + Last[#]) & /@ x),
       vNew}];
   States = state;
   ];
DrawIt[x_List] := Module[{pos, vv},
   (*Drawing the boids*)
```

FIG. A.2 – *Le modèle « Flockers » sous* `Mason`

Listing A.2 *Début du code source du modèle « Flockers » sous* `Mason`

```
package sim.app.flockers;
import sim.engine.*;
import sim.field.continuous.*;
import sim.util.*;
import ec.util.*;

public class Flocker implements Steppable, sim.portrayal.Oriented2D
    {
    public Double2D loc = new Double2D(0,0);
    public Double2D lastd = new Double2D(0,0);
    public Continuous2D flockers;
    public Flockers theFlock;

    public Bag getNeighbors()
        {
        return flockers.getObjectsWithinDistance(loc, theFlock.neighborhood, true);
        }

    public double getOrientation() { return orientation2D(); }
    public boolean isDead() { return dead; }
    public void setDead(boolean val) { dead = val; }

    public double orientation2D()
        {
        if (lastd.x == 0 && lastd.y == 0) return 0;
        return Math.atan2(lastd.y, lastd.x);
        }

    public Double2D momentum()
        {
        return lastd;
        }

    public Double2D consistency(Bag b, Continuous2D flockers)
        {
        if (b==null || b.numObjs == 0) return new Double2D(0,0);

        double x = 0;
        double y= 0;
        int i =0;
        for(i=0;i<b.numObjs;i++)
            {
            Flocker other = (Flocker)(b.objs[i]);
            if (!other.dead)
                {
                double dx = flockers.tdx(loc.x,other.loc.x);
                double dy = flockers.tdy(loc.y,other.loc.y);
                double lensquared = dx*dx+dy*dy;
                if (lensquared <= theFlock.neighborhood * theFlock.neighborhood)
                    {
                    Double2D m = ((Flocker)b.objs[i]).momentum();
```

– `Repast` [North et al., 2005] (testé en version 3.1) : c'est une plate-forme de simulation disponible en plusieurs langages (`Java`, `Python`, `.Net`). Elle est simple à installer, mais, elle n'est fourni qu'avec peu d'exemples de modèles, dont aucun ne ressemble au modèle « *Boids* ». La figure A.3 montre l'interface du logiciel.

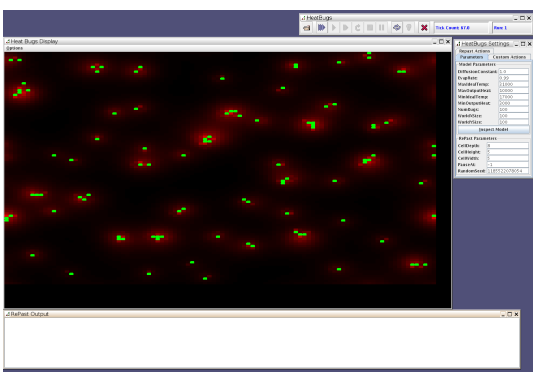


FIG. A.3 – *Le modèle « HeatBugs » sous* `Repast`

– `Breve` [Klein, 2002] (testé en version 2.5.1) : est une plate-forme de simulation 3D, elle est simple à installer. Elle intègre de base un modèle « *Swarm* » inspiré du modèle « *Boids* ». La figure A.4 présente l'interface du logiciel, et le listing A.3 montre un bout du code du modèle « *Swarm* ».

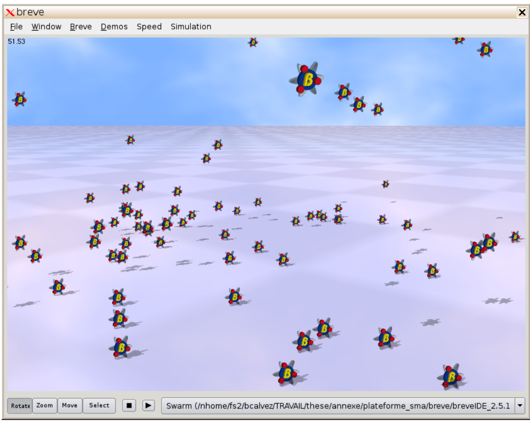


FIG. A.4 – *Le modèle « Swarm » sous* `Breve`

– `Starlogo` [MIT, 2000] (testé en version 2.22) : il existe plusieurs versions de cette plate-forme de simulation codée en java, dont une version libre `OpenStarLogo` (normalement multi-plate-forme, mais il manque des scripts shell pour pouvoir la compiler sous Linux) et une version 3D `StarlogoNG` disponible seulement sous *Windows* et *Mac*. La figure A.5 présente l'interface du logiciel, et le listing A.4 montre un bout du code du modèle « *Boids* ».
– `SeSAm` [Universität Würzburg, 2002] (testé en version 2.2) : `SeSAm` (« *Shell for Simulated Agent Systems* ») est une plate-forme de simulation codée en java. Malgré la sortie récente de cette version (avril 2007), l'installation échoue avec l'utilisation de `Java 1.6`. Le test de ce logiciel a été effectué sur le modèle `Flocking.xml`, qui est un modèle de démonstration du logiciel. La figure A.6 présente l'interface du logiciel, et le listing A.5 montre un bout du code du modèle « *Flocking.xml* ».
– `Netlogo` [Wilensky, 1999] (testé en version 4 beta3) : `Netlogo` est une plate-forme de simulation codée en java. L'utilisateur doit implanter ses modèles en utilisant le langage `logo`. L'installation est très simple, avec des mises-à-jour fréquentes. Le test de ce simulateur a été

1. `http://condor.depaul.edu/~slytinen/abm/StupidModel/`

Listing A.3 *Début du code source du modèle « Swarm » sous* `Breve`

```
@include "Stationary.tz"
@include "Control.tz"
@include "Mobile.tz"

@define SWARM_SIZE        80.

Controller Swarm.

Control : Swarm {
        + variables:
                birds (list).
                item (object).
                breveTexture, cloudTexture (object).
                selection (object).

                # the menu items corresponding to different flocking behaviors

                wackyMenu (object).
                obedientMenu (object).
                normalMenu (object).

        + to click on item (object):
                if selection: selection hide-neighbor-lines.
                if item: item show-neighbor-lines.

                selection = item.

                super click on item.

        + to init:
                floor (object).

                # Set up menus to modify the type of movement allowed.

                self add-menu named "Smoosh The Birdies" for-method "squish".
                self add-menu-separator.
                obedientMenu = (self add-menu named "Flock Obediently" for-method "flock-obediently").
                normalMenu = (self add-menu named "Flock Normally" for-method "flock-normally").
                wackyMenu = (self add-menu named "Flock Wackily" for-method "flock-wackily").

                self enable-lighting.
                self enable-smooth-drawing.
                self move-light to (0, 20, 20).

                breveTexture = (new Image load from "images/breve.png").
                cloudTexture = (new Image load from "images/clouds.png").

                # Add a huge floor.

                floor = new Floor.
                floor catch-shadows.
```

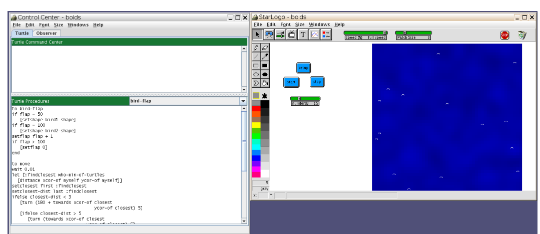


FIG. A.5 – *Le modèle « Boids » sous* `StarLogo`

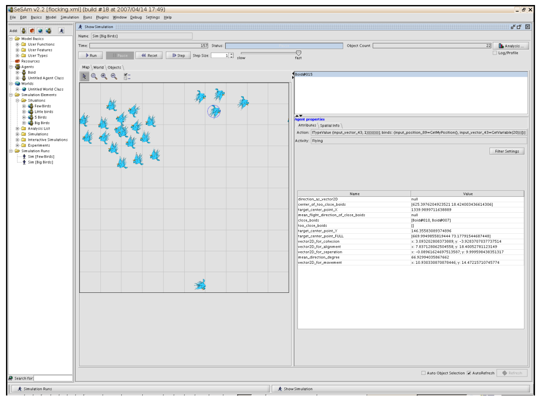


FIG. A.6 – *Le modèle « Flocking.xml » sous* `SeSAm`

Listing A.4 *Début du code source du modèle « Boids » sous* `StarLogo`

```
Java StarLogo
2.1
English
‘turtle‘
to bird—flap
if flap = 50
   [setshape bird1—shape]
if flap = 100
   [setshape bird2—shape]
setflap flap + 1
if flap > 100
   [setflap 0]
end

to move
wait 0.01
let [:findclosest who—min—of—turtles
  [distance xcor—of myself ycor—of myself]]
setclosest first :findclosest
setclosest—dist last :findclosest
ifelse closest—dist < 3
   [turn (180 + towards xcor—of closest
                             ycor—of closest) 5]
   [ifelse closest—dist > 5
      [turn (towards xcor—of closest
                             ycor—of closest) 5]
      [turn (heading—of closest) 5]]
fd 0.3
rt 1.0 * (—1 + random 3)
end

to turn :other—heading :degrees
ifelse ((:other—heading >= heading) and
        ((:other—heading — heading) <= :degrees)) or
       ((:other—heading <= heading) and
        ((heading — :other—heading) <= :degrees))
   [seth :other—heading]
   [ifelse ((:other—heading — heading) mod 360) < 180
      [rt :degrees]
      [lt :degrees]]
end


‘observer‘
turtles—own [closest closest—dist flap]
patches—own [air]

to setup
ca
ask—patches [setair random 10]
```

Listing A.5 *Début du code source du modèle « Flocking.xml » sous* `SeSAM`

```
<?xml version="1.0" encoding="UTF-8"?>
<XMLSeSAmModel version="0.2" globalAssert="true"
    compileDebugFunction="true" runDebugFunction="true" optimize="true"
    inline="true" closures="false" xmlns="http://www.simsesam.de/schemas/model">
    <userType name="vector2D" id="UserType_0" type="composed">
        <father systemID="RootType"/>
        <attribute id="ComposedTypeAttribute_0" name="x">
            <doubleType/>
        </attribute>
        <attribute id="ComposedTypeAttribute_1" name="y">
            <doubleType/>
        </attribute>
    </userType>
    <userFunction name="rotate_counterclockwise" id="UserFunction_0">
        <functionCall>
            <call functionName="CreateComposedTypeValue">
                <userType id="UserType_0"/>
                <call functionName="-">
                    <call functionName="*">
                        <call functionName="Cos">
                            <parameterID id="FunctionArgument_1"/>
                        </call>
                        <call functionName="GetComposedTypeValue">
                            <parameterID id="FunctionArgument_0"/>
                            <composedAttribute
                                composedTypeID="UserType_0" index="0"/>
                        </call>
                    </call>
                    <call functionName="*">
                        <call functionName="Sin">
                            <parameterID id="FunctionArgument_1"/>
                        </call>
                        <call functionName="GetComposedTypeValue">
                            <parameterID id="FunctionArgument_0"/>
                            <composedAttribute
                                composedTypeID="UserType_0" index="1"/>
                        </call>
                    </call>
                </call>
                <call functionName="+">
                    <call functionName="*">
                        <call functionName="Cos">
                            <parameterID id="FunctionArgument_1"/>
                        </call>
                        <call functionName="GetComposedTypeValue">
                            <parameterID id="FunctionArgument_0"/>
                            <composedAttribute
                                composedTypeID="UserType_0" index="1"/>
                        </call>
                    </call>
```

réalisé sur le modèle `Flocking` [Wilensky, 1998d]. La figure A.7 présente l'interface du logiciel, et le listing A.6 montre un bout du code du modèle « *Flocking* ».

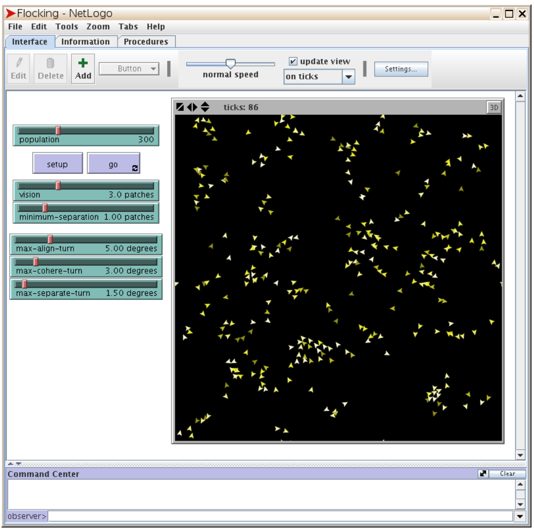


FIG. A.7 – *Le modèle « Flocking » sous* `Netlogo`

- `Swarm` [Minar et al., 1996] (testé en version 2.2) : c'est une plate-forme codée en `Objectif-C`. L'installation est complexe : `configure` où il manque des dépendances, version stable où il faut modifier des lignes dans le code source pour que ça compile, par exemple.

Cette énumération expose les plates-formes de simulations existantes. Si un utilisateur veut utiliser une plate-forme, il doit regarder en détail certains nombres de points :

- la facilité d'installation : un logiciel conçu en `Java` est normalement plus simple à installer qu'un logiciel conçu en `Objectif-C` avec des librairie comme `TCL-TK` ;
- la fréquence des mises à jours ;
- la compatibilité avec des technologies récentes ;
- le fait d'être multi-plate-forme ;
- la licence du logiciel ;
- la documentation ;
- le nombre d'exemples ;
- la performance : généralement un logiciel en `Java` est moins performant qu'un logiciel en `C` ;
- l'expressivité de la plate-forme ;
- la communauté derrière la plate-forme ;
- la simplicité de programmation.

Pour tous nos tests, nous avons décidé d'utiliser la plate-forme `NetLogo`, car son intérêt est tout d'abord sa simplicité[Railsback et al., 2006b]. Cet avantage est cependant au détriment de l'expressivité du langage et de la rapidité. Un autre autre intérêt de `NetLogo` est qu'il est fourni avec un grand nombre de modèles qui nous ont servi à tester les différentes méthode d'exploration de l'espace des paramètres.

Si l'utilisateur désire implanter un modèle particulier, d'abord il peut créer un prototype en une plate-forme relativement simple comme `NetLogo`. Puis, il peut se tourner vers des plates-formes plus complexes. Par contre, si ses besoins sont assez précis, comme l'a montré le groupe de travail *SMABio* [2], il est préférable de construire soi-même son propre simulateur.

2. `http://www.phys-mito.u-bordeaux2.fr/mitowiki/index.php?title=GT_SMA_%26_Simulation_Cellulaire`
`http://indico.in2p3.fr/categoryDisplay.py?categId=84`

Listing A.6 *Début du code source du modèle « Flocking » sous* `Netlogo`

```
turtles-own [
  flockmates          ;; agentset of nearby turtles
  nearest-neighbor    ;; closest one of our flockmates
]

to setup
  clear-all
  crt population
    [ set color yellow - 2 + random 7  ;; random shades look nice
      set size 1.5  ;; easier to see
      setxy random-xcor random-ycor ]
end

to go
  ask turtles [ flock ]
  ;; the following line is used to make the turtles
  ;; animate more smoothly.
  repeat 5 [ ask turtles [ fd 0.2 ] display ]
  ;; for greater efficiency, at the expense of smooth
  ;; animation, substitute the following line instead:
  ;;   ask turtles [ fd 1 ]
  tick
end

to flock  ;; turtle procedure
  find-flockmates
  if any? flockmates
    [ find-nearest-neighbor
      ifelse distance nearest-neighbor < minimum-separation
        [ separate ]
        [ align
          cohere ] ]
end

to find-flockmates  ;; turtle procedure
  set flockmates other turtles in-radius vision
end

to find-nearest-neighbor ;; turtle procedure
  set nearest-neighbor min-one-of flockmates [distance myself]
end

;;; SEPARATE

to separate  ;; turtle procedure
  turn-away ([heading] of nearest-neighbor) max-separate-turn
end

;;; ALIGN
```

B Exemples de simulations multi-agents

À travers les différentes méthodes proposées dans ce manuscrit, de nombreux exemples de simulation multi-agent ont été proposés. Dans cette annexe sont présentés les différents modèles à base d'agents utilisés qui ont comme point commun d'être issus de la bibliothèque de modèles de `NetLogo` [Wilensky, 1999], comme indiqué dans l'annexe A page 119. Ces modèles ont été modifiés pour appliquer les méthodes de parcours automatique de l'espace, expliqué dans l'annexe C page 133, et pour les complexifier (par exemple ajout de paramètres pré-existants, mais non utilisés...).

B.1 Modèle « *Cooperation* » de `NetLogo`

Le modèle « *Cooperation* » [Wilensky, 1998b] est un modèle provenant de la bibliothèque de modèle de `NetLogo`. Ce modèle représente un phénomène biologique évolutionnaire : des agents (ici des vaches ou « *cows* » sont en compétition sur une ressource naturelle (de l'herbe ou « *grass* »)). Plus les agents broutent l'herbe, plus ils pourront se reproduire, et donc plus seront les gagnants du points de vue évolutionnaire. Dans le modèle original, deux types d'agents sont distingués : les agents égoïstes et les agents coopératifs. Les agents égoïstes broutent l'herbe, quelque soit la hauteur de l'herbe. Les agents coopératives ne broutent l'herbe que si l'herbe est suffisamment haute. LA hauteur de l'herbe est importante : l'herbe pousse plus lentement en dessous de cette hauteur (et donc pousse plus rapidement au dessus). Le fait de brouter augmente l'énergie de l'agent alors que chaque action de l'agent consomme cette énergie. Quand l'agent a suffisamment d'énergie, il se reproduit en créant un clone de lui-même. Les agents coopératives laissent de la nourriture pour toute la population alors que les agents égoïstes broutent toute la nourriture sans se soucier de la population. La figure B.1 montre le modèle dans la plate-forme `NetLogo`.

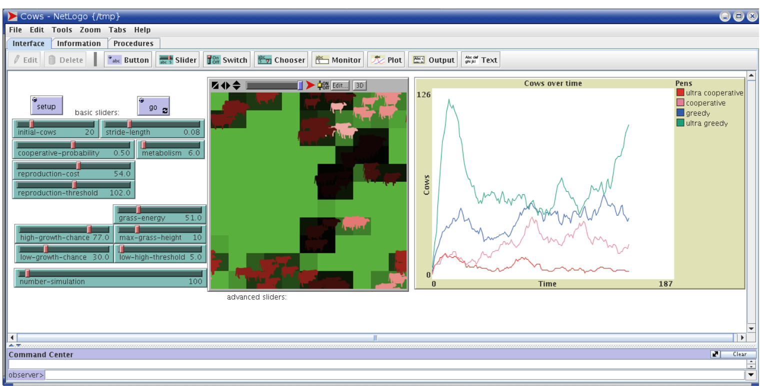


FIG. B.1 – *Image du modèle « Cooperation »*

Ce modèle a été adapté pour l'utilisation des différentes stratégies de parcours de l'espace des paramètres. De plus, ce modèle a été complexifié : chaque agent est caractérisé par une variable `appetite` caractérisant l'appétit de l'agent. Un agent broute de l'herbe si la hauteur de l'herbe est supérieur à la variable `appetite`. Il est possible de caractériser différentes stratégies : coopérative (un agent laisse beaucoup d'herbe sur l'environnement) ou égoïste (l'agent laisse aucune herbe sur l'environnement). Un des objectifs de calibrage du modèle serait de trouver la valeur optimale de `appetite` pour maximiser le nombre d'agents sur l'environnement.

B.2 Modèle « *Traffic 2 Lanes* » de NetLogo

Le modèle « *Traffic 2 Lanes* » [Wilensky, 1998f] est un modèle provenant de la bibliothèque de modèle de NetLogo. Ce modèle représente une autoroute à deux voies où des voitures (les agents de la simulation) peuvent accélérer, freiner ou changer de voie. Les agents sont caractérisés par différentes variables locales ou globales, dont les principales sont :

- lanes : cette variable local indique la voie où l'agent se situe actuellement ;
- patience : cette variable local représente le niveau de patience actuel de la voiture ;
- speed : cette variable local indique la vitesse instantanée de la voiture ;
- speed-limit : ce paramètre local indique la vitesse maximal à laquelle l'agent peut rouler ;
- max-patience : ce paramètre local indique la patience limite d'une voiture ;
- look-ahead : ce paramètre global représente le nombre de cases en avant qu'un agent perçoit ;
- speed-up : ce paramètre global représente l'accélération en milli-case par pas de simulation ;
- slown-down : ce paramètre global représente le freinage en milli-case par pas de simulation ;

L'agent est modélisé de la façon suivante :

- L'agent perçoit look-ahead cases devant lui ;
- S'il ne perçoit aucun autre agent, alors il accélère de speed-up sans dépasser la vitesse limite de speed-limit ;
- S'il perçoit d'autres agents devant lui, il prend la vitesse de l'un d'eux, puis il freine de slow-down ;
- De plus, s'il perçoit d'autres agents devant lui, si l'un de ces agents a une vitesse inférieure à la sienne, alors il décide de le doubler au prochain pas de simulation :
  - Pour changer de voie, à chaque pas de simulation, il essaie d'aller sur l'autre voie : il y a trois changements de cases verticales avant atteindre l'autre voie, donc il faut au minimum trois pas de simulation pour changer de voie. Il ne peut changer de case verticale que si la case est libre.
  - Durant cette période, la patience de l'agent est décrémentée à chaque pas de simulation. Si elle devient nulle ou négative, alors l'agent ne veut plus changer de voie, et essaie de regagner sa voie d'origine en appliquant la même méthode de changement de voie (la patience est réinitialisée, et sera décrémentée à chaque pas de simulation).

La figure B.2 montre le modèle dans la plate-forme NetLogo.

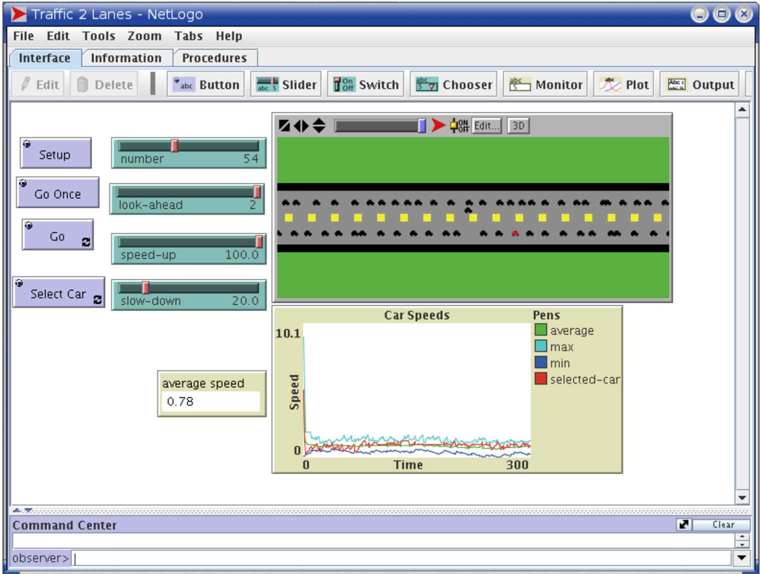


FIG. B.2 – *Image du modèle « Traffic 2 Lanes »*

Ce modèle a été adapté pour l'utilisation des différentes stratégies de parcours de l'espace des paramètres. De plus, ce modèle a été complexifié : les paramètres globaux max-patience, look-a-

`head`, `speed-up`, `slown-down` sont devenus locaux. Un des objectifs de calibration est de trouver le jeu de paramètres qui permet de minimiser les embouteillages sur les deux voies.

B.3 Modèle « *Flocking* » de `NetLogo`

Le modèle « *Flocking* » [Wilensky, 1998d] est un modèle provenant de la bibliothèque de modèle de `NetLogo`. Il est inspiré du modèle « *boids* » de Reynolds [Reynolds, 1987]. Ce modèle représente le vol d'une nuée d'oiseaux. Les agents de ce modèles sont des oiseaux. La nuée qui apparaît dans ce modèle est ni créée, ni menée d'aucune façon par des agents chefs. En fait, chaque agent suit exactement les mêmes règles, qui s'organisent la nuée d'oiseaux. Les agents oiseaux suivent ces trois règles suivantes :

- « *alignment* » : l'alignement signifie qu'un oiseau a tendance à tourner pour que son déplacement soit dans la même direction que le déplacement des oiseaux voisins.
- « *separation* » : la séparation signifie qu'un oiseau a tendance à tourner pour éviter un autre oiseau qui est trop près.
- « *cohesion* » : la cohésion signifie qu'un oiseau a tendance à se déplacer vers les autres oiseaux proches (à moins que les autres oiseaux soient trop proches)

Tant que deux oiseaux sont proches (c'est-à-dire la distance les séparant est inférieure à `minimum-separation`), alors seulement la règle de « *separation* » est appliquée. La figure B.3 montre le modèle dans la plate-forme `NetLogo`.

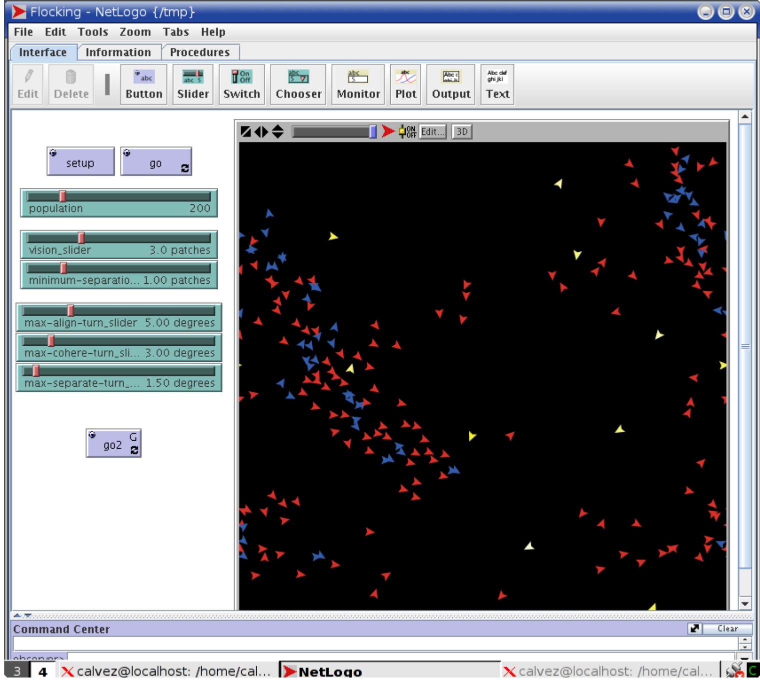


FIG. B.3 – *Image du modèle « Flocking »*

Ce modèle a été adapté pour l'utilisation des différentes stratégies de parcours de l'espace des paramètres. De plus, ce modèle a été complexifié en rendant locaux des des paramètres globaux :

- `vision` : ce nombre de cases caractérise la zone de l'environnement de perception de l'agent. Cette zone correspond à l'ensemble des cases se trouvant à une distance inférieure à `vision` cases.
- `minimum-separation` : c'est la distance maximale pour qualifier que deux agents soient trop proches.
- `max-align-turn` : c'est l'angle maximal de changement de cap de l'agent appliqué par la règle « *alignment* ».

– `max-cohere-turn` : c'est l'angle maximal de changement de cap de l'agent appliqué par la règle « *cohere* ».
– `max-separation-turn` : c'est l'angle maximal de changement de cap de l'agent appliqué par la règle « *separation* ».

Un des objectifs de calibration pourrait être de maximiser la taille des nuées.

B.4 Modèle « *Divide The Cake* » de NetLogo

Le modèle « *Divide The Cake* » [Wilensky, 1998c] est un modèle appartenant à la bibliothèque de modèles de la plate-forme `NetLogo`. C'est un modèle de jeu évolutionnaire. Trois types d'agents se partage une ressource commune (de l'herbe dans le modèle implanté dans `NetLogo`). Le type des agents est différencié par leur appétit représenté par la variable `Appetite` :

– les agents égoïstes ou « *Greedy* » sont les agents dont l'appétit est égale à $\frac{2}{3} \times max_appetite$;
– les agents équitables ou « *Fair* » sont les agents dont l'appétit est égale à $\frac{1}{2} \times max_appetite$;
– les agents modestes ou « *Modest* » sont les agents dont l'appétit est égale à $\frac{1}{3} \times max_appetite$.

Durant le pas de simulation, les agents se déplacent (`move`) puis essaient de manger de l'herbe (`eat`) : cette dernière phase se déroule de la façon suivante pour un agent :

– S'il a déjà participé à cette phase (`turn` à vrai) durant ce pas de simulation, il ne fait rien.
– Sinon, il regarde la somme des appétits des autres agents présents sur la même case que lui en ajoutant son appétit. Si cette somme est inférieur à `max_appetite`, alors tous les agents présents sur la case peuvent se reproduire (plus leurs appétits est forts, plus ils ont des chances de se reproduire), et mettent leur variable `turn` à faux. Sinon, tous les agents présents sur la case meurent.

La figure B.4 montre le modèle dans la plate-forme `NetLogo`.

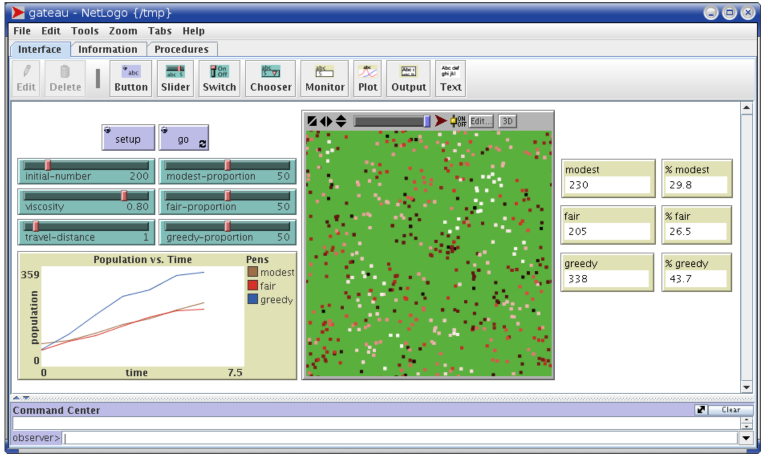


FIG. B.4 – *Image du modèle « Divide The Cake »*

Ce modèle a été adapté pour l'utilisation des différentes stratégies de parcours de l'espace des paramètres. De plus, ce modèle a été complexifié : la variables `Appetite` prend des valeurs entre 0 et $max_appetite$. Il est toujours possible de distinguer trois types de population d'agents :

– les agents égoïstes ou « *Greedy* » sont les agents dont l'appétit est supérieur à $\frac{2}{3} \times max_appetite$;
– les agents équitables ou « *Fair* » sont les agents dont l'appétit est compris entre $\frac{1}{3} \times max_appetite$ et $\frac{2}{3} \times max_appetite$;
– les agents modestes ou « *Modest* » sont les agents dont l'appétit est inférieur à $\frac{1}{3} \times max_appetite$.

Un objectif d'exploration de l'espace des paramètre serait de trouver la valeur du paramètre `Appetite` qui maximise le nombre d'individu de la population d'agents.

B.5 Modèle « *Ants* » de NetLogo

Le modèle « *Ants* » [Wilensky, 1998a] est modèle fourni par la bibliothèque de modèle de la plate-forme NetLogo. Ce modèle représente le fourragement par colonie de fourmis. Dans ce modèle très simple, la fourmilière se situe au centre de l'environnement, et trois sources de nourriture à la périphérie. Initialement, les fourmis partent de la fourmilière à la recherche de nourriture. Quand une fourmi trouve de la nourriture, elle revient à la fourmilière en déposant des phéromones. Quand une fourmi est en phase de recherche de nourriture, elle essaie de percevoir dans son environnement des phéromones : si elle en perçoit, elle se dirige vers la zone où elle perçoit le plus de phéromones. Sinon, elle fait une recherche aléatoire. Plus de fourmis transportent de nourriture vers la fourmilière, plus le chemin chimique se renforce. Ainsi, dans le modèle simulé, l'utilisateur observe des files de fourmis comme dans la réalité. Les deux principaux paramètres intéressants dans cette simulation sont ;

– diffusion-rate : ce paramètre caractérise la diffusion des phéromones sur l'environnement.
– evaporation-rate : ce paramètre caractérise l'évaporation des phéromones sur l'environnement.

La figure B.5 montre le modèle dans la plate-forme NetLogo.

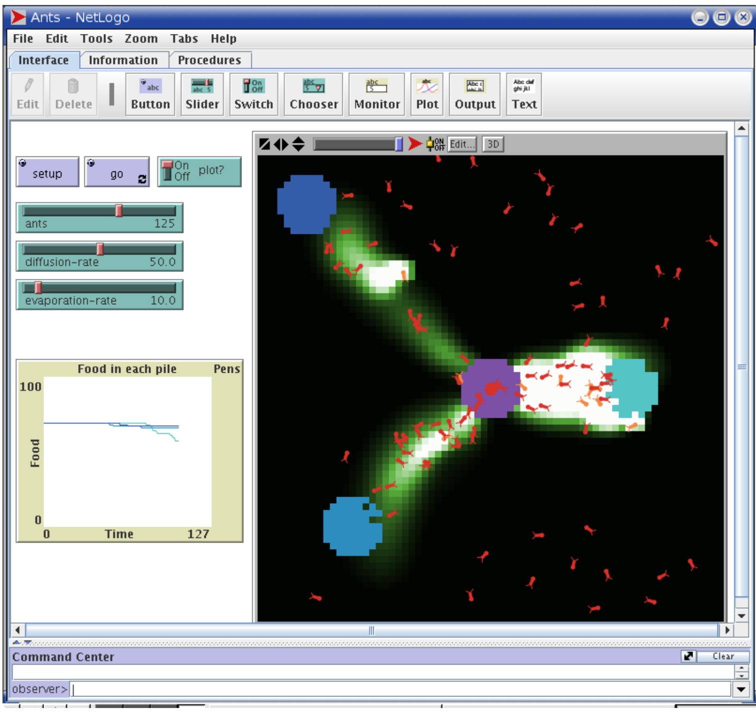


FIG. B.5 – *Image du modèle « Ants »*

Ce modèle a été adapté pour l'utilisation des différentes stratégies de parcours de l'espace de paramètres. De plus, ce modèle a été complexifié : des paramètres locaux, pré-existants dans le modèle et non-modifiable, ont été ajoutés :

– speed : ce paramètre caractérise la vitesse de l'agent ;
– patch_ahead : ce paramètre caractérise le nombre de cases devant l'agent lui permettant de percevoir des phéromones ;
– angle_vision : ce paramètre caractérise l'angle de vision de l'agent lors de la perception des phéromones ;
– drop_size : ce paramètre caractérise la quantité initiale de phéromones que l'agent dispose lors de son retour à la fourmilière suite à la découverte de nourriture.

Un objectif de calibrage de ce modèle pourrait être d'optimiser la fonction de retour de la nourriture, et donc une fonction de *fitness* pourrait être la quantité de nourriture ramené avant t pas de simulation.

B.6 Modèle « *Wolf Sheep Predation* » de `NetLogo`

Le modèle « *Wolf Sheep Predation* » [Wilensky, 1998g] est modèle fournit par la bibliothèque de modèle de la plate-forme `NetLogo`. Ce modèle représente un écosystème où cohabitent deux populations d'agents : des moutons (les proies) et des loups (les prédateurs). Il se compose de deux sous-modèles. Dans le premier sous-modèle, les loups et les moutons se déplacent aléatoirement dans l'environnement. Chaque déplacement coûte une unité d'énergie (variable `energy`) aux loups, et aucune énergie aux moutons. Quand son énergie devient nulle ou négative, le loup meurt. À chaque pas de simulation, les loups essaient d'attraper un mouton : il attrape un mouton seulement s'ils sont sur la même case ou *patches*, et dans ce cas, son énergie remonte de `wolf-gain-from-food`. De plus, à chaque pas de simulation, les agents ont une possibilité de se reproduire en fonction d'une probabilité : à chaque fois qu'ils se reproduisent, leur énergie est divisée par deux. Pour l'autre sous-modèle, la seule différence se trouve au niveau du comportement du mouton. Dans ce modèle, de l'herbe est ajouté à l'environnement. Elle pousse selon un taux `grass-regrowth-time`. Quand un mouton broute l'herbe, son énergie augmente de `sheep-gain-from-food`. Et comme pour le premier sous-modèle avec les loups, chaque déplacement coûte une unité d'énergie aux moutons. Les principaux paramètres intéressants de ce modèle sont :

- `grass-regrowth-time` : ce paramètre correspond au nombre de pas de simulation nécessaire à la repousse de l'herbe.
- `sheep-gain-from-food` : ce paramètre correspond au gain d'énergie quand un mouton broute de l'herbe.
- `wolf-gain-from-food` : ce paramètre correspond au gain d'énergie quand un loup mange un mouton.
- `sheep-reproduce` : ce paramètre correspond à la probabilité de reproduction d'un mouton à chaque pas de simulation.
- `wolf-reproduce` : ce paramètre correspond à la probabilité de reproduction d'un loup à chaque pas de simulation.

La figure B.6 montre le modèle dans la plate-forme `NetLogo`.

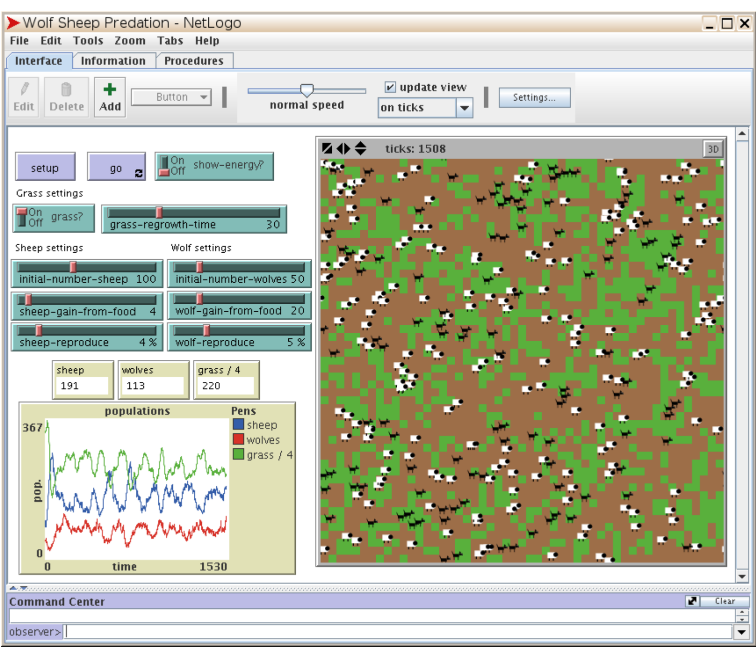


FIG. B.6 – *Image du modèle « Wolf Sheep Predation »*

C Implantation des différentes méthodes

Les différentes méthodes présentées dans ce manuscrit ont été implémentées principalement en `java` et plus particulièrement `java` en version 1.6. En revanche, toute la partie concernant étude des résultats a été faite en `perl`, `bash` avec l'utilisation d'outils comme `gnuplot`, `R`.

L'idée de l'implantation est d'être plus générique possible : plus générique au niveau des modèles à calibrer, plus générique dans les stratégie pour explorer l'espace des paramètres. Tout les simulations employées par ce programme ont utilisé `NetLogo`, mais il est très facile de changer de moteur de simulation, il suffit de créer une classe qui implémente l'interface `Simulateur`. Au niveau de la généricité du programme, si un utilisateur veut réimplanter une population de chromosome pour un algorithme génétique, il suffit de créer une classe qui implémente l'interface

```
public interface Population
      < T extends Number, G extends Gene<T> , C extends Chromosome<T,G> >
      extends Copyable<Population <T,G,C> >.
```

Idem, pour un chromosome avec l'interface

```
public interface Chromosome
      < T extends Number, G extends Gene<T> >
      extends Copyable<Chromosome <T,G> >,
              Crossover<Chromosome<T,G> >.
```

Pour pouvoir utiliser un modèle pour `NetLogo`, il suffit d'implanter ces quelques fonctions :

- `getfitness` : cette fonction renvoie la *fitness* de la simulation.
- `calcul_fitness` : cette fonction calcule la *fitness* de la simulation. Attention, cette opération peut être longue.
- `addParaGlobal` *`arguments`* : cette fonction permet de régler la valeur des paramètres globaux à la simulation
- `addParaLocalGlobal` *`arguments`* : cette fonction permet de régler la valeur des paramètres globaux à la simulation et des paramètres locaux par défaut de la simulation.
- `setup-evolution` : cette fonction initialise le modèle (création de l'environnement...) sans créer des agents.
- `sim` : cette fonction exécute un pas de simulation.
- `setup-turtles-default` *`nombre_agents`* : cette fonction crée *nombre_agents* agents dans la simulation, initialisée par les paramètre par défaut.
- `addAgent` *`group arguments`* : cette fonction crée un agent appartenant au groupe *`group`* dont les paramètres sont initialisés par *`arguments`*.

Après, il faut initialiser un fichier de configuration en `xml` pour configurer le logiciel. Le listing C.1 montre un exemple.

Ensuite, il suffit de lancer le programme.

Le programme distribue les calculs entre plusieurs serveurs de calcul. La communication entre les différents processus `Java` se fait parla technologie `RMI`.

Voici un exemple de scénario simple de communication entre le client, le serveur central, et une unité de calcul (en réalité, le client réalise plusieurs simulation en même temps, et réalise toutes ces tâches en parallèle) :

1. Le client se connecte au serveur central.
2. Le client demande une unité de calcul au serveur central. Si une unité de calcul est disponible, le serveur central lui renvoie le nom de l'unité de calcul. Si aucune unité de calcul n'est disponible, le client s'endort avant de réitérer sa demande.
3. Le client se connecte à l'unité de calcul.
4. L'unité de calcul se connecte au serveur central, pour lui indiquer qu'elle n'est plus disponible.
5. Le client demande à l'unité de calcul de charger le modèle dans le simulateur, si nécessaire.

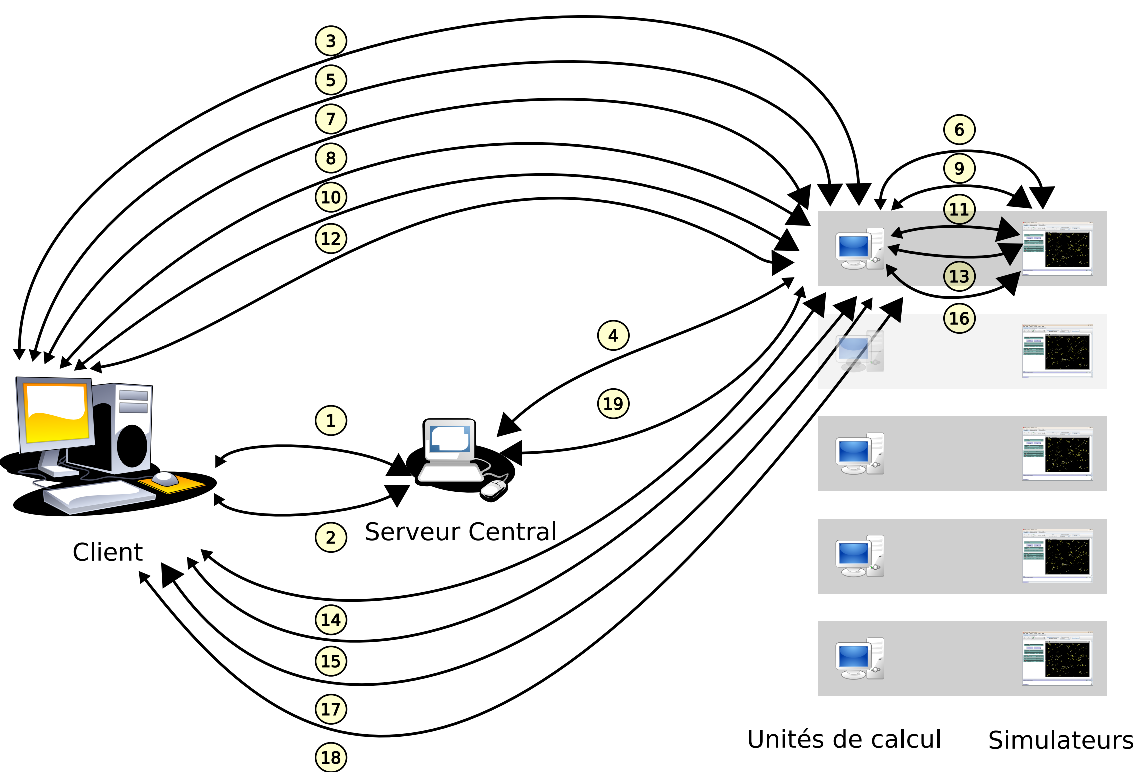
Listing C.1 *Exemple de fichier de configuration*

```
<?xml version='1.0' encoding='utf-8'?>
<conf>
    <cheminModele>/nhome/other/sydra/bcalvez/TRAVAIL/GRILLE/all/modele/Ants_v2.nlogo</cheminModele>
    <!--    <cheminModele>/home/calvez/TRAVAIL/boite_noire/all/modele/Ants_v2.nlogo</cheminModele> -->
    <nombreSimulation>10</nombreSimulation>
    <nombrePasDeSimulation>500</nombrePasDeSimulation>
    <nombreSelection>1000</nombreSelection>
    <nombreAgent>10</nombreAgent>
    <numberMaxInterval>10000</numberMaxInterval>
    <numberSimulationMax>10000</numberSimulationMax>
    <depth>9</depth>

    <typeAgent>
        <name>Ants</name>
        <numberAgent>10</numberAgent>
    </typeAgent>
    <para>
        <nom>speed</nom>
        <debut>0</debut>
        <fin>5</fin>
        <decoupage>10</decoupage>
        <precision>0.00001</precision>
        <global>false</global>
        <type>Ants</type>
    </para>
    <!-- <cheminPara>/nhome/other/sydra/bcalvez/TRAVAIL/GRILLE/Modele_2_niveaux/Flocking_exp9_TEST/para_0.csv</cheminPara> -->
    <para>
        <nom>patch_ahead</nom>
        <debut>0</debut>
        <fin>10</fin>
        <decoupage>10</decoupage>
        <precision>0.00001</precision>
        <global>false</global>
        <type>Ants</type>
    </para>
    <para>
        <nom>angle_vision</nom>
        <debut>0</debut>
        <fin>360</fin>
        <decoupage>10</decoupage>
        <precision>0.00001</precision>
        <global>false</global>
        <type>Ants</type>
    </para>
    <para>
        <nom>drop_size</nom>
        <debut>0</debut>
        <fin>200</fin>
        <decoupage>10</decoupage>
        <precision>0.00001</precision>
```

6. L'unité charge dans le simulateur le modèle si nécessaire.
7. Le client fixe le nombre de pas de simulation dans l'unité de calcul.
8. Le client demande à l'unité de calcul d'initialiser la simulation.
9. L'unité de calcul initialise la simulation (`setup-evolution`).
10. Le client demande à l'unité de calcul de régler le paramétrage et de créer des agents suivant les paramètres qu'il lui fournit pour le modèle.
11. L'unité de calcul règle les paramètres globaux de la simulation, et crée des agents suivant les paramètres fournit par le client (`addParaLocalGlobal addParaGlobal addAgent setup-turtles-default`
12. Le client demande la simulation du modèle à l'unité de calcul.
13. L'unité de calcul lance la simulation (`sim`) et le calcul de la *fitness* (`calcul_fitness`) sur le simulateur.
14. Le client attend que l'unité de calcul termine la simulation du modèle.
15. Le client demande à l'unité de calcul la *fitness* de la simulation.
16. L'unité de calcul récupère la *fitness* de la simulation (`getfitness`).
17. L'unité de calcul renvoie la *fitness* au client.
18. Le client signale à l'unité calcul qu'il n'a plus besoin d'elle.
19. L'unité de calcul se connecte au serveur central pour lui indiquer qu'elle est de nouveau disponible

La figure C.1 résume ce scénario de fonctionnement.



N.B. : la grosseur des flèches reflète le sens principale de communication.

FIG. C.1 – *Exemple de scénario de fonctionnement*

D Exemple détaillé de l'utilisation des méthodes

Nous exposons dans cette annexe en détail un exemple.

Nous allons prendre un modèle dans la bibliothèque de modèles de la plate-forme de simulation `NetLogo`. Le choix s'est porté sur le modèle « *Shepherds* » [Wilensky, 1998e].

Le modèle représente des bergers en train de regrouper leurs moutons. Les bergers suivent un jeu de règles simple : chaque berger se déplace aléatoirement. S'il trouve un mouton, il le ramasse, et continue sa marche aléatoire. Quand il rencontre un autre mouton, il cherche à proximité un espace vide afin de poser son mouton, puis, il recommence sa recherche de mouton. En ce qui concerne les moutons, ils déplacent seulement de façon aléatoire. La figure D.1 montre le modèle dans la plate-forme `NetLogo`.

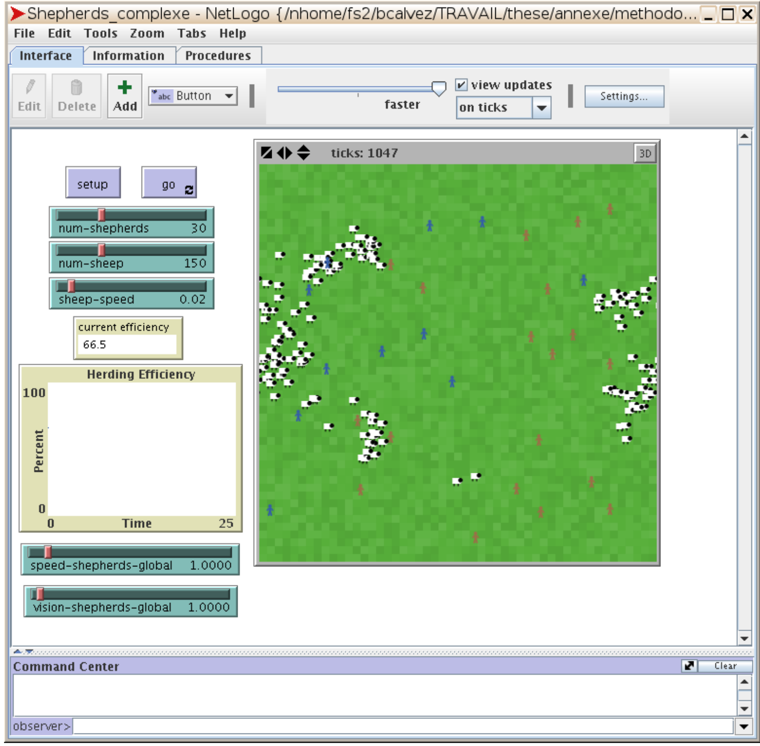


FIG. D.1 – *Image du modèle « Shepherds »*

Ce modèle est très simple. Pour le rendre plus intéressant, il a été complexifié : ajout d'une vitesse pour les bergers, et ajout de distance de perception des moutons pour les bergers lors de leur recherche. Le listing D.1 page suivante montre les différences entre les deux modèles.

Le problème est de trouver le jeu de paramètres qui maximise l'efficacité des bergers. Les deux paramètres ajoutés sont les seuls à varier, les autres paramètres sont fixées initialement :

- le nombre de berger (`num-shepherds`) est 30 ;
- le nombre de mouton (`num-sheep`) est 250 ;
- la vitesse des moutons (`sheep-speed`) est 0,1 case par pas de simulation.

Le modèle est simulé 10000 pas.

Puis, pour pouvoir utiliser les méthodes développées, il faut rajouter quelques fonctions dans le modèle, comme indiqué à l'annexe C page 133. Le listing D.2 page 139 montre les différences pour

Listing D.1 *Différence entre le modèle original « Shepherds » et le modèle complexifié*

```
16a17,18
>   speed-shepherds
>   vision-shepherds
34c36,38
<         setxy random-xcor random-ycor ]
---
>         setxy random-xcor random-ycor
>         set speed-shepherds speed-shepherds-global
>         set vision-shepherds vision-shepherds-global]
56c60
<       fd 1
---
>       fd speed-shepherds
77c81
<   set carried-sheep one-of sheep-here with [not hidden?]
---
>   set carried-sheep one-of sheep in-radius vision-shepherds with [not hidden?]
285a290,319
> SLIDER
> 9
> 425
> 234
> 458
> speed-shepherds-global
> speed-shepherds-global
> 0
> 10
> 1
> 0.0001
> 1
> NIL
> HORIZONTAL
>
> SLIDER
> 12
> 468
> 232
> 501
> vision-shepherds-global
> vision-shepherds-global
> 0
> 20
> 1
> 0.0001
> 1
> NIL
> HORIZONTAL
>
```

rajouter ces fonctions dans ce modèle. Il faut concevoir le fichier de configuration : le listing D.3 page ci-contre montre une possibilité de fichier de configuration.

Enfin, la méthode découpage est lancé durant 10000 simulations. La figure D.2 page 140 expose l'évolution de la *fitness* en fonction du nombre de simulations à travers différentes méthodes présentées dans ce manuscrit. Nous retrouvons les différentes conclusions énoncées dans ce manuscrit. Les méthodes à base de découpage de paramètres ont un démarrage plus rapide que la méthode avec algorithme génétique. Cependant, cette dernière a tendance à converger plus haut. La figure D.3 page 140 montre l'évolution du découpage du paramètre `speed-shepherds` : il y a convergence très nette du découpage vers environ 3,5. La figure D.4 page 141 montre l'évolution du découpage du paramètre `vision-shepherds` : il y a convergence vers environ 3,5. Ainsi, les bergers ont une vitesse relativement moyenne, mais une distance de perception des moutons proche de leur vitesse. Ce résultat s'explique très bien : en un pas de simulation, les bergers avance de `speed-shepherds` cases. Et ils perçoivent les moutons à `vision-shepherds` cases à la ronde. Donc, aussitôt qu'ils voient un mouton, en un pas de simulation, ils rejoignent le mouton. Il serait possible d'imaginer que le meilleur jeu de paramètres semblerait `speed-shepherds` et `vision-shepherds` égaux les plus grands possible. Mais, ce n'est pas le cas. En effet, tel qu'est programmé ce modèle, en un pas de simulation, les bergers avancent de `speed-shepherds` cases. Si un berger perçoit un mouton à un nombre de case très inférieure à `speed-shepherds`, lors de son déplacement, le berger dépassera le mouton : ils ne seront pas sur la même case, et donc le berger ne pourra pas le ramasser.

Nous pouvons aussi montrer un autre exemple sur le même modèle. Nous allons ajouter la possibilité de modifier les agents moutons : nous serons dans ce cas avec des agents hétérogènes. Les moutons seront caractérisés par une vitesse. Les conditions d'expérimentation sont identiques que précédemment. Le listing D.4 page 141 montre les différences entre le précédent modèle complexifié et le nouveau modèle où la vitesse des moutons est devenu un paramètre ajustable par la méthode de calibrage.

Les résultats sont similaires à la précédentes expériences pour les bergers. La figure D.5 page 142 montre l'évolution du découpage du paramètre `speed-sheep` : il y a convergence très nette du découpage vers environ 0,03. Les moutons avancent très lentement, ce qui laisse le temps aux bergers

Listing D.2 *Différence entre le modèle original « Shepherds » et le modèle complexifié*

```
110a111,170
> to-report getFitness
>     report herding-efficiency
> end
>
> to calcul-fitness
>     calculate-herding-efficiency
> end
>
> to addParaGlobal [ ]
> end
>
> to addParaLocalGlobal [ vitesse_berger vision_berger ]
>     set speed-shepherds-global vitesse_berger
>     set vision-shepherds-global vision_berger
> end
>
> to setup-evolution
>     ca
>     set num-shepherds 30
>     set num-sheep 250
>     set sheep-speed 0.1
>     set-default-shape sheep "sheep"
>     set-default-shape shepherds "person"
>     ask patches
>       [ set pcolor green + (random-float 0.8) - 0.4]   ;; varying the green just makes it look nicer
>     create-sheep num-sheep
>       [ set color white
>       set size 1.5  ;; easier to see
>       setxy random-xcor random-ycor ]
>     update-sheep-counts
> end
>
> to sim
>     go
> end
>
> to setup-turtles-default [nbAgent]
>   create-shepherds nbAgent
>     [ set color brown
>       set size 1.5  ;; easier to see
>       set carried-sheep nobody
>       set found-herd? false
>       setxy random-xcor random-ycor
>       set speed-shepherds speed-shepherds-global
>       set vision-shepherds vision-shepherds-global]
> end
>
> to addAgent [ groupe vitesse_berger vision_berger ]
> ifelse groupe = "Shepherds"
>     [
>      create-shepherds 1 [
>           set color brown
>           set size 1.5  ;; easier to see
>           set carried-sheep nobody
>           set found-herd? false
>           setxy random-xcor random-ycor
>           set speed-shepherds speed-shepherds-global
>           set vision-shepherds vision-shepherds-global]
>     ][ print "erreur dans addAgent : type non reconnu" ]
> end
```

Listing D.3 *Différence entre le modèle original « Shepherds » et le modèle complexifié*

```
<?xml version='1.0' encoding='utf-8'?>
<conf>
    <cheminModele>/nhome/fs2/bcalvez/TRAVAIL/these/annexe/methodologie/modele/Shepherds_methode.nlogo</cheminModele>
    <nombreSimulation>10</nombreSimulation>
    <nombrePasDeSimulation>10000</nombrePasDeSimulation>
    <nombreSelection>1000</nombreSelection>
    <nombreAgent>30</nombreAgent>
    <numberMaxInterval>50000</numberMaxInterval>
    <numberSimulationMax>10000</numberSimulationMax>
    <depth>9</depth>

    <typeAgent>
        <name>Shepherds</name>
        <numberAgent>30</numberAgent>
    </typeAgent>
    <para>
        <nom>speed-shepherds</nom>
        <debut>0</debut>
        <fin>10</fin>
        <decoupage>10</decoupage>
        <precision>0.00001</precision>
        <global>false</global>
        <type>Shepherds</type>
    </para>
    <para>
        <nom>vision-shepherds</nom>
        <debut>0</debut>
        <fin>20</fin>
        <decoupage>10</decoupage>
        <precision>0.00001</precision>
        <global>false</global>
        <type>Shepherds</type>
    </para>
</conf>
```

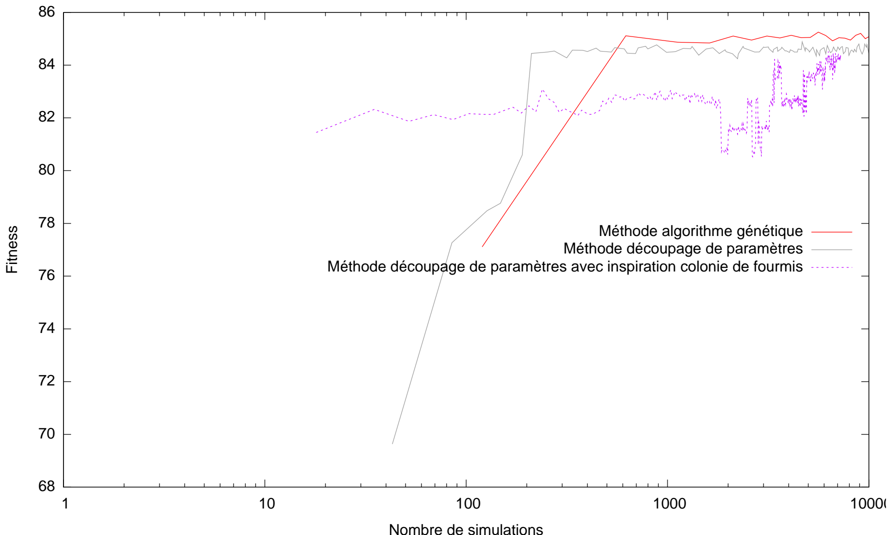


FIG. D.2 – *Évolution de la fitness en fonction du nombre de simulations sur le modèle « Shepherds »*

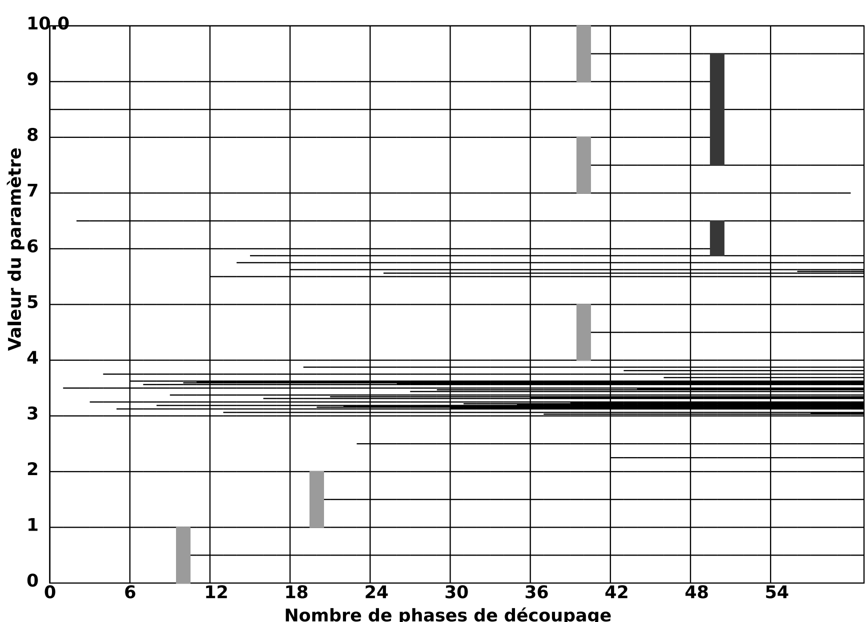


FIG. D.3 – *Évolution du découpage du paramètre* `speed-shepherds` *avec la méthode découpage de paramètres sur le modèle « Shepherds »*

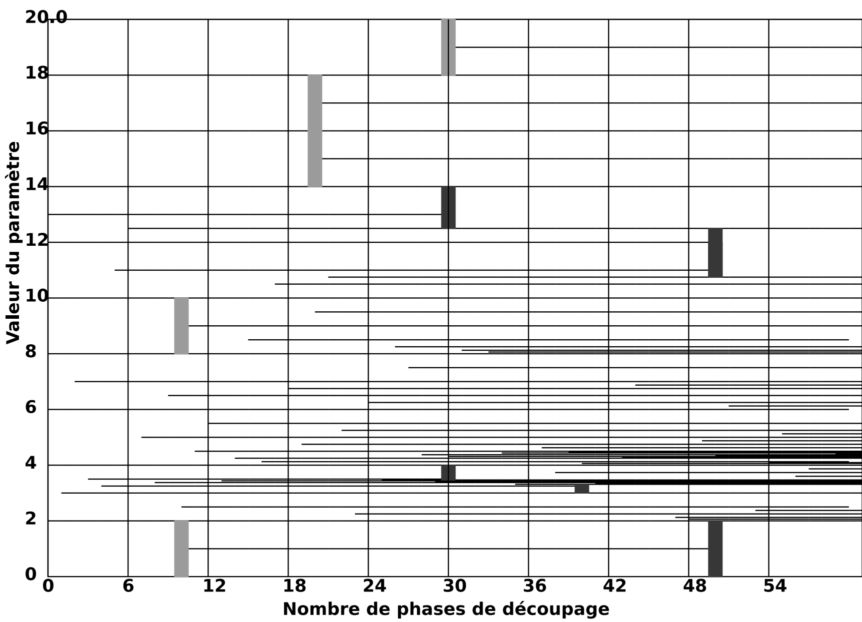


FIG. D.4 – *Évolution du découpage du paramètre* `vision-shepherds` *avec la méthode découpage de paramètres sur le modèle « Shepherds »*

Listing D.4 *Différence entre le modèle complexifié précédent et le modèle complexifié nouveau*

```
19a20,23
> sheep-own
> [
>     speed-sheep
> ]
29a34
>       set speed-sheep speed-sheep-global
66c71
<     fd sheep-speed ]
---
>     fd speed-sheep ]
122c127
< to addParaLocalGlobal [ vitesse_berger vision_berger ]
---
> to addParaLocalGlobal [ vitesse_berger vision_berger vitesse_mouton ]
124a130
>     set speed-sheep-global vitesse_mouton
155a162,167
>   create-sheep num-sheep
>     [ set color white
>       set size 1.5  ;; easier to see
>       set speed-sheep speed-sheep-global
>       setxy random-xcor random-ycor ]
>
169c181,192
<     ][ print "erreur dans addAgent : type non reconnu" ]
---
>     ]
>     [
>       ifelse groupe = "Sheep"
>       [
>         create-sheep 1
>           [ set color white
>             set size 1.5  ;; easier to see
>             set speed-sheep vitesse_berger
>             setxy random-xcor random-ycor ]
>       ]
>       [ print "erreur dans addAgent : type non reconnu" ]
>     ]
379a403,417
> SLIDER
> 254
> 473
> 449
> 506
> speed-sheep-global
> speed-sheep-global
> 0
> 10
> 1
> 0.0001
> 1
> NIL
> HORIZONTAL
>
```

de les ramener.

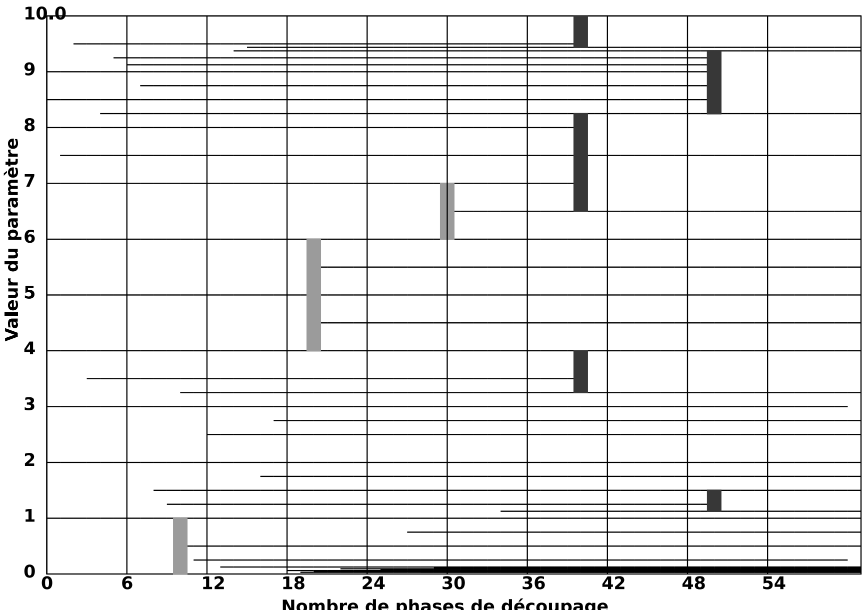


FIG. D.5 – *Évolution du découpage du paramètre* `speed-sheep` *avec la méthode découpage de paramètres sur le modèle « Shepherds »*

E Fonctions de référence

Lors du développement des différentes méthodes, nous avons dû tester plusieurs approches ce qui beaucoup de simulations, et donc énormément de temps. Pour éviter de s'investir sur des approches potentiellement intéressantes, mais qui sont en fait de compte totalement inintéressantes, nous avons développé une « sorte » de modèle à base d'agents à partir de fonctions mathématiques. L'idée est de pourvoir avoir un résultat de « pseudo-simulation » de façon instantanée , afin de réaliser un très grand nombre de tests (étude de stabilité, modification des paramètres des algorithmes...). Un autre avantage est d'avoir une connaissance complète a priori sur l'espace des paramètres, et de concevoir ses propres espaces de paramètres.

En pratique, nous avons un pseudo-modèle avec 100 agents. Les agents possèdent des paramètres variant entre 0 et 100. La *fitness* d'un modèle dépend d'une fonction mathématique dont ses paramètres sont ceux des agents.

Dans la suite de la section, nous allons exposer quelques résultats.

Cependant, il faut faire attention à la pertinence de ces résultats. Avec ces fonctions prédéfinies, nous réalisons un grand nombre de pseudo-simulations, alors qu'avec un modèle à base d'agents, ce n'est pas envisageable. Dans la simulation à base d'agents, les agents interagissent entre eux. Cependant introduire un « mauvais » agent peut mener avoir un résultat de simulation non satisfaisant alors que la simulation ne contient que de « bons » agents sauf un. Par ailleurs, l'environnement et le côté spatial des modèles à base d'agents sont complètement oubliés avec ces fonctions prédéfinies : regroupement d'agents, ressource rare... Finalement, les résultats obtenues ne sont que des indications sur le comportement des approches.

E.1 *Rastrigin*

Ce test repose sur la fonction de *Rastrigin*, fonction classique de test pour les algorithme génétique [Marco-Blaska et Désidéri, 1999]. Chaque agent possèdent cinq paramètres x_i avec $i \in [\![1,5]\!]$ ($x_i \in [0;100]$)). La *fitness* d'un agent est égale à :

$$50 + \sum_{i=1}^{5} \left(\left(\frac{x_i - 50}{25} \right)^2 - 10cos \left(2\Pi \left(\frac{x_i - 50}{25} \right) \right) \right).$$

La *fitness* globale est égale à la somme des *fitness* individuelles de chaque agent :

$$\sum_{a \in A} 50 + \sum_{i=1}^{5} \left(\left(\frac{x_{i_a} - 50}{25} \right)^2 - 10cos \left(2\Pi \left(\frac{x_{i_a} - 50}{25} \right) \right) \right).$$

La figure E.1 page suivante présente l'espace des paramètres d'un agent dans le cas où celui-ci a seulement deux paramètres. Le but de l'optimisation est de minimiser la *fitness*. Voici quelques points remarquables en dimension deux :

x	y	valeur
50	50	0

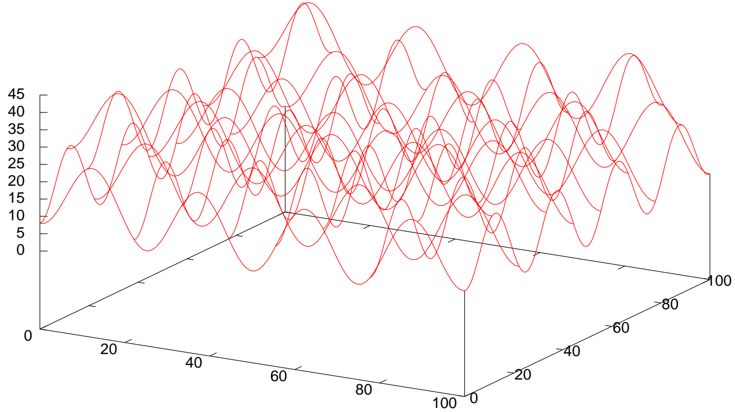


FIG. E.1 – *Visualisation de l'espace des paramètres d'un agent pour une fonction de Rastrigin en dimension 2.*

La figure E.2 montre les résultats de différentes méthodes face à ce problème. Les conclusions sont similaires aux conclusions de la sous-section II.4.1.1 page 79.

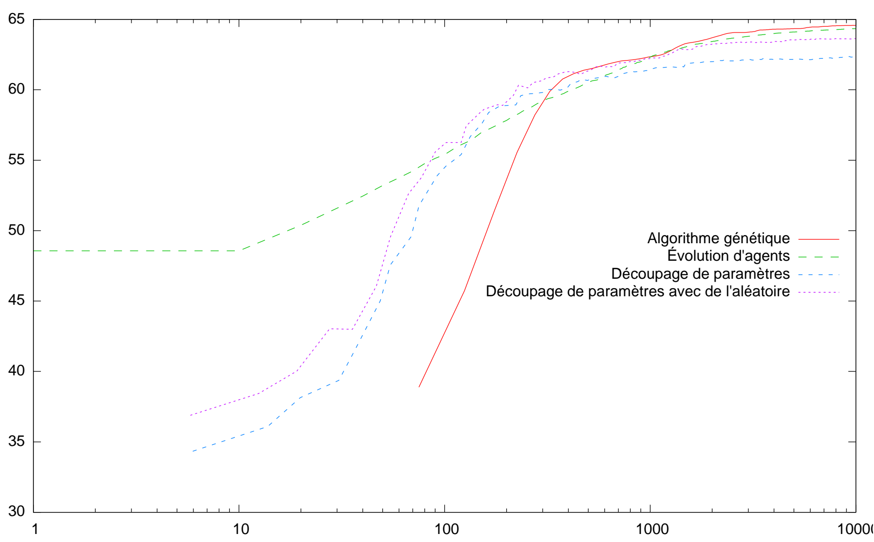


FIG. E.2 – *Visualisation de l'évolution de la fitness moyenne à travers les trois méthodes pour la fonction de test Rastrigin.*

E.2 Fonction « plate »

Cette fonction teste le problème où le paysage de la *fitness* est à peu près plat sauf pour l'optimal. Chaque agent possèdent cinq paramètres x_i avec $i \in [\![1,5]\!]$ ($x_i \in [0; 100]$). Soit la fonction g composée

telle que

$$g(x) = \begin{cases} 0{,}4x & x \in [0;25] \\ 10 & x \in]25;34] \\ x - 24 & x \in]34;35] \\ -x + 46 & x \in]35;36] \\ 10 & x \in]36;65] \\ -\frac{2}{7}x + \frac{200}{7} & x \in]65;100] \end{cases}$$

La *fitness* d'un agent est égale à la somme des valeurs de la fonction g appliquée à chaque paramètre multipliée par un poids tel que :

$$f(\vec{X}) = \frac{1}{6}g(x_1) + \frac{1}{3}g(x_2) + g(x_3) + \frac{1}{3}g(x_4) + \frac{1}{6}g(x_5)$$

La *fitness* globale est égale à la somme des *fitness* individuelles de chaque agent.

L'intérêt d'ajouter des poids sur les paramètres permet d'éviter d'avoir un espace isotrope quellesque soient les paramètres. La figure E.3 montre l'espace des paramètres d'un agent dans le cas de la dimension 2 sans poids sur les paramètres. Voici quelques points remarquables :

x_1	x_2	x_3	x_4	x_5	valeur
35	35	35	35	35	22

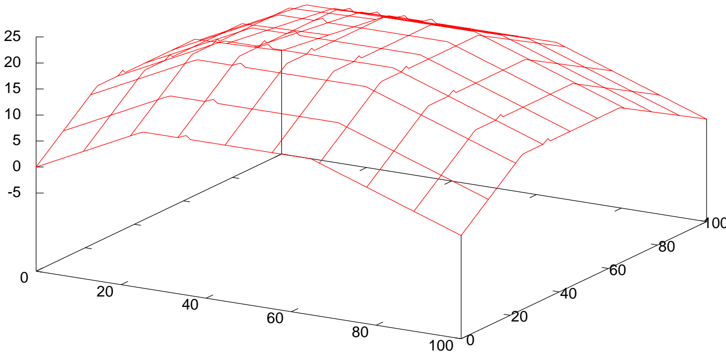


FIG. E.3 – *Visualisation de l'espace des paramètres d'un agent pour la fonction « plate » pour deux paramètres.*

La figure E.4 page suivante indique les résultats de différentes méthodes face à ce problème. Les conclusions sont similaires aux conclusions de la sous-section II.4.1.1 page 79. L'intérêt principale des ces « pseudo-simulations » est de pouvoir réaliser un très grand nombre de simulations afin d'analyser, par exemple, la stabilité des méthodes. Ainsi, la figure E.5 page suivante montre la stabilité des résultats pour la méthode algorithme génétique au niveau de la valeur du paramètre 2 sur les meilleurs modèles : au fil des différentes exécutions, les résultats sont stables. La figure E.6 page 147 montre la stabilité des résultats pour la méthode découpage de paramètres au niveau du découpage final du paramètre 2. Cette méthode est moins stable que la méthode à base d'algorithme génétique : il y a deux exécutions sur 26 où les résultats sont notablement différents.

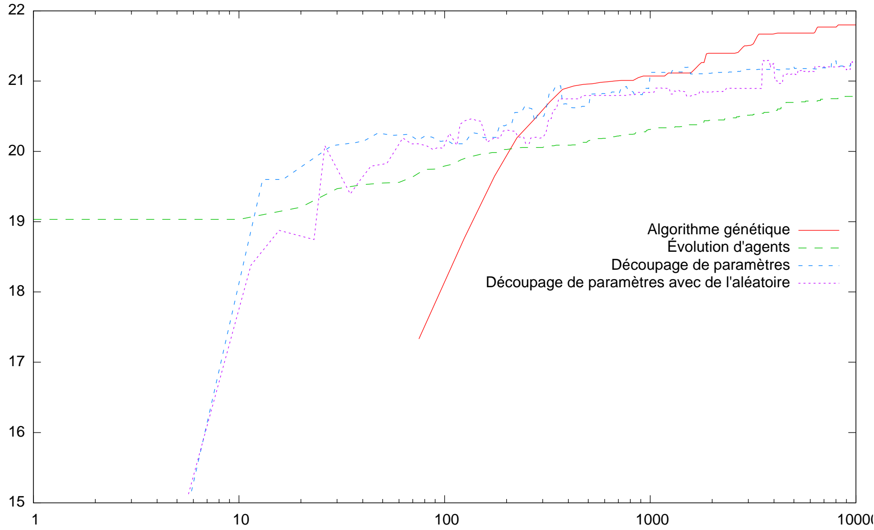


FIG. E.4 – *Visualisation de l'évolution de la fitness à travers les trois méthodes.*

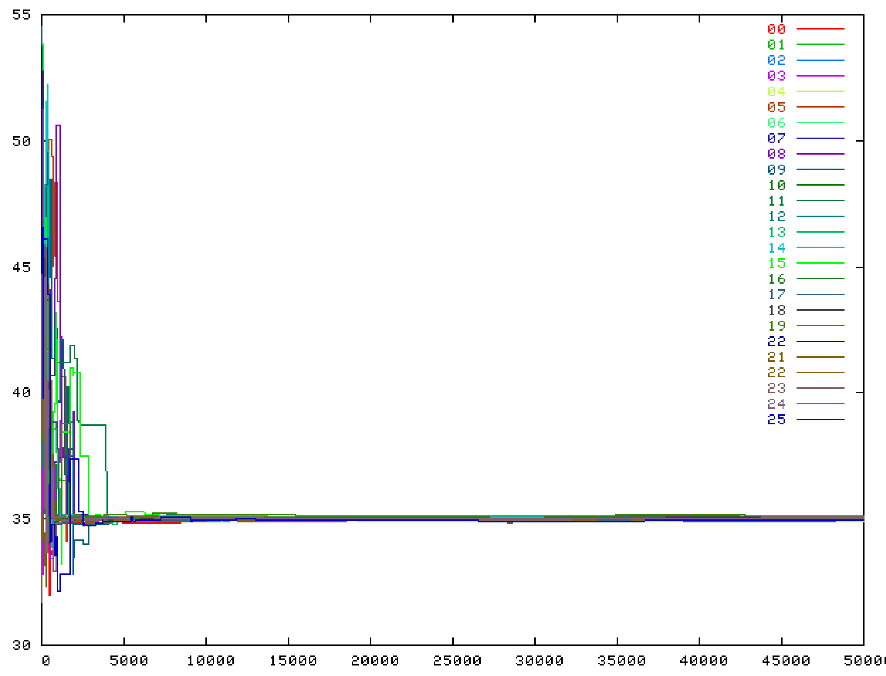


FIG. E.5 – *Visualisation de la moyenne du paramètre 2 des 5 meilleurs modèles de chaque génération.*

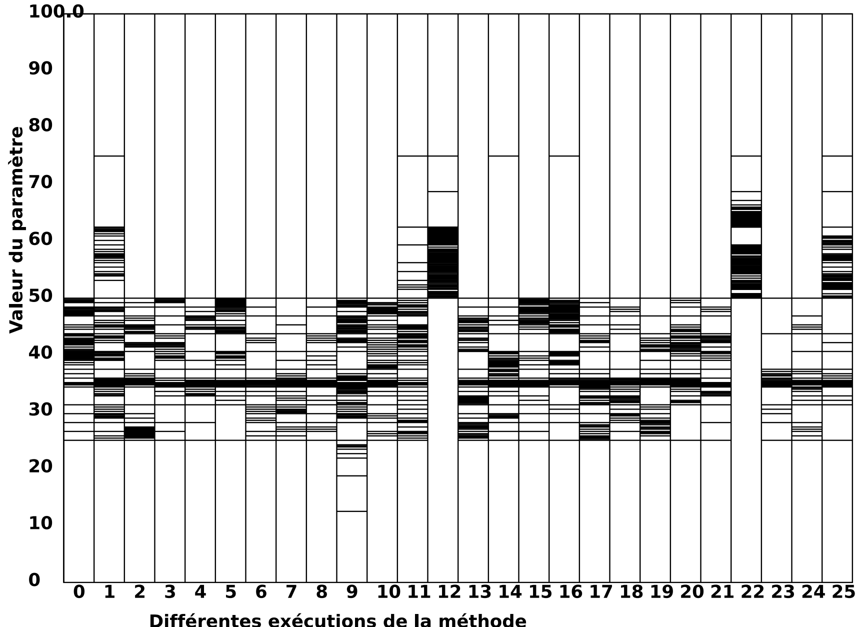


FIG. E.6 – *Visualisation du l'intervalle final du paramètre 2.*

E.3 Fonction « somme de cosinus »

Dans ce pseudo-modèle, nous avons 100 agents avec 10 paramètres variant de 0 à 100. La *fitness* d'un agent est égale à :

$$\begin{aligned} f(\vec{X}) = \quad & cos\left(\frac{x_1}{25}\right) + \tfrac{3}{4} \times cos\left(\frac{x_2}{25}\right) + \tfrac{1}{2} \times cos\left(\frac{x_3}{25}\right) + \tfrac{1}{4} \times cos\left(\frac{x_4}{25}\right) + \tfrac{1}{8} \times cos\left(\frac{x_5}{25}\right) \\ & -cos\left(\frac{x_6}{25}\right) - \tfrac{3}{4} \times cos\left(\frac{x_7}{25}\right) - \tfrac{1}{2} \times cos\left(\frac{x_8}{25}\right) - \tfrac{1}{4} \times cos\left(\frac{x_9}{25}\right) - \tfrac{1}{8} \times cos\left(\frac{x_{10}}{25}\right) \end{aligned}$$

La *fitness* globale est égale à la somme des *fitness* individuelles de chaque agent. La figure E.7 page suivante montre l'espace des paramètres d'un agent dans le cas de la dimension 2 sans poids sur les paramètres.

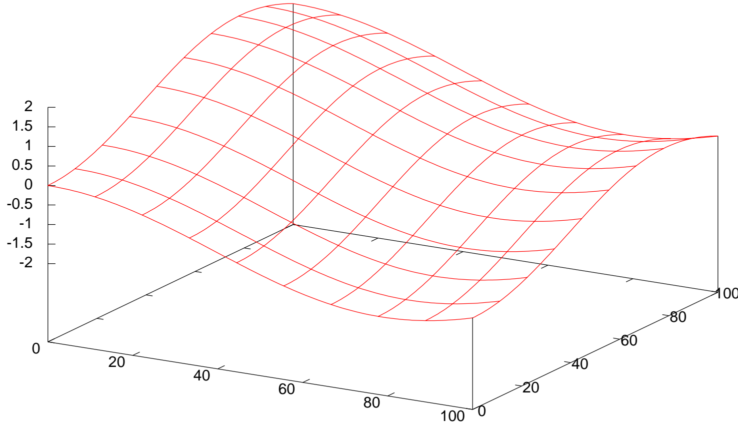


FIG. E.7 – *Visualisation de l'espace des paramètres d'un agent pour la fonction « somme de cosinus » pour deux paramètres*

E.4 Fonction « somme de sinus »

Dans ce pseudo-modèle, nous avons 100 agents. Chaque agent possèdent deux paramètres x et y ($(x,y) \in [0;100]^2$)). La *fitness* d'un agent est égale à $\frac{sin(\sqrt{\left(\frac{x-25}{4}\right)^2+\left(\frac{y-25}{4}\right)^2})}{\sqrt{\left(\frac{x-25}{4}\right)^2+\left(\frac{y-25}{4}\right)^2}}$. La *fitness* globale est égale à la somme des *fitness* individuelles de chaque agent: $\sum_{a\in A}\frac{sin(\sqrt{\left(\frac{x_a-25}{4}\right)^2+\left(\frac{y_a-25}{4}\right)^2})}{\sqrt{\left(\frac{x_a-25}{4}\right)^2+\left(\frac{y_a-25}{4}\right)^2}}$. La figure E.8 page suivante montre l'espace des paramètres d'un agent. Le but des méthodes est de maximiser la *fitness*.

Voici quelques points remarquables :

x	y	valeur
25	25	1

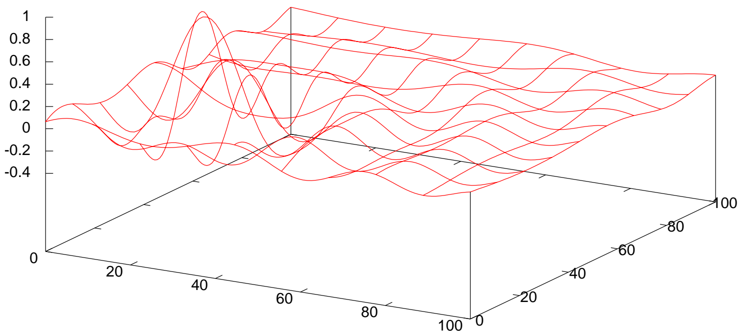


FIG. E.8 – *Visualisation de l'espace des paramètres d'un agent pour la fonction « somme de sinus »*

E.5 Fonction de *Ackley*

Dans ce pseudo-modèle, nous avons 100 agents avec 10 paramètres variant de 0 à 100. La *fitness* d'un agent repose sur la fonction de *Ackley* [Ackley, 1987] :

$$f(\vec{X}) = -20 \times e^{\left(-0{,}2\sqrt{\frac{1}{10}\sum_{i=1}^{10}\left(\frac{3}{5}(x_i-50)\right)^2}\right)} - e^{\left(\frac{1}{10}\sum_{i=1}^{10} cos(2\pi\left(\frac{3}{5}(x_i-50)\right))\right)} + 20 + e$$

La figure E.9 montre l'espace des paramètres d'un agent dans le cas de la dimension 2 sans poids sur les paramètres.

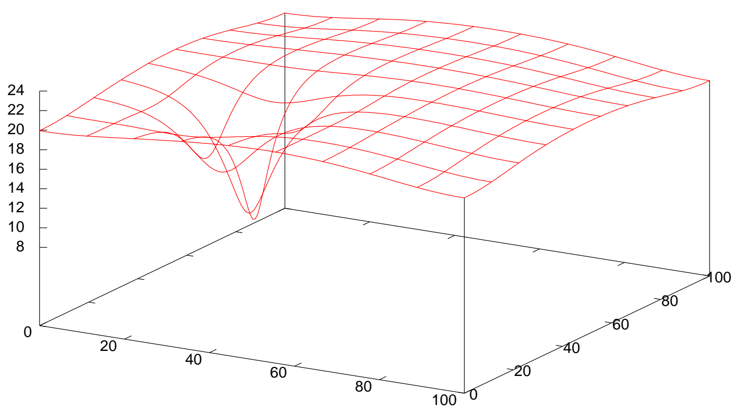
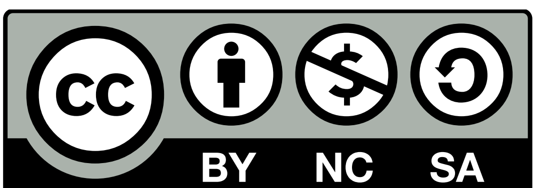


FIG. E.9 – *Visualisation de l'espace des paramètres d'un agent pour la fonction d'Ackley*


BY NC SA