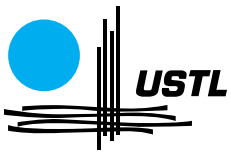


CENTRE NATIONAL
DE LA RECHERCHE
SCIENTIFIQUE

Approche centrée interaction pour la simulation d'agents cognitifs situés

THÈSE

présentée et soutenue publiquement le 3 juillet 2007

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille

(spécialité informatique)

par

Damien Devigne

Composition du jury

Président :	Jean-Luc Dekeyser, Professeur	LIFL, Université de Lille I
Rapporteurs :	Pierre Chevaillier, HDR	CERV, ENIB, Brest
	Zahia Guessoum, HDR	LERI, Reims
	Catherine Tessier, HDR	ONERA, SUPAERO, Toulouse
Directeurs :	Philippe Mathieu, Professeur	LIFL, Université de Lille I
	Jean-Christophe Routier, HDR	LIFL, Université de Lille I

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
Laboratoire d'Informatique Fondamentale de Lille — UMR USTL/CNRS 8022
U.F.R. d'I.E.E.A. – Bât. M3 – 59655 VILLENEUVE D'ASCQ CEDEX
Tél. : +33 (0)3 28 77 85 41 – Télécopie : +33 (0)3 28 77 85 37 – email : direction@lifl.fr

Ce travail est cofinancé par le CPER TAC de la région Nord-Pas de Calais, les fonds européens du FEDER, et une allocation de recherche MESR.

Table des matières

« Au début, il n'y avait rien. Enfin, ni plus ni moins de rien qu'ailleurs. »
Les Shadoks

Remerciements

Avant d'ouvrir ce mémoire, je tiens à remercier chaleureusement les autres acteurs de cette thèse, qui ont tous eu une importance à un moment ou un autre de l'avancée de mon travail :

– **mes encadrants :** Philippe, qui a un réel don pour remotiver les troupes quand on pense que rien ne va plus, Jean-Christophe pour son dynamisme, ses idées et sa compagnie lors des déplacements à l'étranger
– **les membres du jury,** et en particulier les rapporteurs pour leur relecture attentive du mémoire et leurs nombreux conseils qui ont permis de l'améliorer,
– **les autres smaciens,** pour la bonne ambiance de travail qui règne dans *le bureau 14*, les échanges d'opinion plus ou moins houleux lors de nos traditionnels repas du jeudi
– **mes anciens professeurs** qui m'ont donné le bagage nécessaire pour en arriver là, ainsi que la curiosité scientifique indispensable à la longue entreprise que constitue la thèse
– **mes parents,** dont les sacrifices m'ont permis de poursuivre de looongues études
– **mes amis** qui m'ont apporté quelques bouffées d'oxygène divertissantes permettant de sortir de temps en temps la tête de l'eau
– **ma moitié** qui m'accompagne patiemment en supportant la tension générée par certaines échéances de mon travail
– **les oubliés,** il y en a toujours... merci et pardon !
– **les autres,** à eux de voir s'ils préfèrent se considérer comme oubliés...

Résumé

Les simulations par agents offrent des possibilités que l'on ne retrouve pas dans les simulations numériques, comme l'analyse des interactions qui se produisent entre les entités et du rôle de chacune de ces entités dans le résultat de la simulation : le modèle centré agent permet une analyse des phénomènes au niveau micro. Toutefois, trop souvent, un système multi-agent est créé spécifiquement pour une simulation et il n'est pas possible de réutiliser facilement les composants d'une simulation à une autre. Le modèle centré interaction que nous proposons a l'avantage de séparer les aspects déclaratifs et les aspects procéduraux en positionnant comme notion centrale les interactions que les agents peuvent effectuer ou subir. Ceci permet de récupérer et réutiliser facilement les composants dont le code n'est pas éparpillé dans la simulation comme il peut l'être avec un modèle centré agent. Ce modèle permet entre autres de réaliser des simulations d'entités autonomes dans un environnement géographique. Pour modéliser de telles simulations et notamment la planification des déplacements, je propose une représentation des connaissances et un moteur de comportements capable de prendre en compte les contraintes propres aux aspects situés des simulations. Je propose également une gestion d'équipe d'agents qui utilise le même moteur de comportements. Une attention particulière a été apportée à l'autonomie de décision des agents au sein de l'équipe afin que ces derniers prennent part activement à la planification grâce à leur propre moteur de comportement et ne soient pas utilisés comme de simples outils.

Avant-propos

« The question of whether a computer can think is no more interesting than the question of whether a submarine can swim. »
E. W. Dijkstra

Depuis ses débuts, l'intelligence artificielle a donné lieu à toutes sortes de fantasmes, incarnés virtuellement par le biais de la science-fiction : de HAL dans « 2001, l'Odyssée de l'Espace », R2D2 dans « Star Wars » à « A.I. Artificial Intelligence » de Steven Spielberg en passant par la voiture parlante dans la série « K2000 », il a toujours semblé que nous nous rapprochions de plus en plus du jour où la réalité rejoindrait la fiction. Pourtant, à l'heure actuelle, force est de constater que les voitures ne discutent pas et que les ordinateurs ne font pas encore preuve d'une intelligence débordante.

L'intelligence artificielle (I.A.) a pourtant fait des avancées évidentes dans diverses applications : reconnaissance de formes, apprentissage (pour le langage, la classification), jeux (échecs par exemple), planification, agents conversationnels, etc. Pour résoudre ces différents problèmes, des algorithmes très efficaces ont été mis en place. Ces algorithmes sont spécialisés et s'adressent chacun à un problème ou à une classe de problèmes. A l'heure actuelle, nous ne connaissons pas encore « d'I.A. globale » qui serait capable d'un raisonnement de haut niveau s'appliquant à une grande variété de problèmes, à l'instar de l'intelligence humaine.

C'est avec ces quelques réflexions à l'esprit qu'il faut considérer ce travail : il élargit humblement le champ d'action de l'I.A. en s'attaquant à l'une de ses multiples branches. Tant qu'on n'aura pas découvert une I.A. globale et générique, ce n'est qu'ainsi que l'on pourra progresser.

Introduction Générale

Depuis ses débuts il y a une cinquantaine d'années, l'informatique a connu un essor fantastique, jusqu'à représenter aujourd'hui un passage obligé pour la plupart des activités humaines. L'informatique est maintenant présente partout, de manière explicite (internet, jeux, musique, vidéo...) ou plus ou moins visible (antivols de supermarché, automobile, téléphones...). Ce développement rapide de l'informatique a rapidement réveillé le rêve pluri-séculaire de la machine intelligente supposée assister ou remplacer l'Homme dans l'exécution de certaines tâches : pourquoi ne pas utiliser la puissance de calcul disponible dans un ordinateur pour faire prendre des décisions intelligentes automatiquement à celui-ci ? C'est ainsi qu'est née l'*intelligence artificielle*. Si à ses débuts il s'agissait de prouver automatiquement des théorèmes mathématiques ou de résoudre des problèmes d'algèbre, l'enjeu majeur de l'intelligence artificielle est aujourd'hui de réaliser une entité disposant d'une intelligence comparable à celle d'un être humain, afin de pouvoir converser, jouer avec celui-ci ou décider, travailler à la place de celui-ci. Dans le domaine des simulations, ceci est rendu possible entre autres grâce à la planification. Celle-ci consiste à trouver une succession d'actions dont l'exécution permet d'atteindre une configuration souhaitée de l'environnement. Les algorithmes de planification sont nombreux. Toutefois, s'ils sont très efficaces voire optimaux, les algorithmes de planification sont assez peu adaptés aux problèmes situés. Pourtant, nous verrons qu'une bonne gestion des déplacements participe beaucoup à la rationalité du comportement observé. Or dans le domaine de la simulation de comportements, la rationalité des comportements observés prime parfois sur l'optimalité. De plus, les solutions de planification actuelles ne permettent pas de gérer plusieurs agents de manière centralisée.

De plus en plus, les programmes disposant d'une intelligence artificielle cohabitent avec d'autres programmes ayant les mêmes capacités. Lorsque ces entités communiquent afin de résoudre un problème ensemble, on parle d'intelligence artificielle distribuée, ou de systèmes multi-agents. Dans ces systèmes, la communication et la collaboration ont un rôle prépondérant dans la résolution des problèmes. Les systèmes multi-agents offrent en outre la possibilité d'analyser les interactions qui se produisent entre plusieurs entités, ce qui permet l'étude de comportements sociaux au sein d'un groupe.

Il existe de nombreuses plates-formes d'agents. Celles-ci se proposent d'aider le concepteur d'un système multi-agent dans la réalisation de celui-ci, en fournissant des outils, des bibliothèques ou des interfaces de visualisation. Ces plates-formes sont assez peu adaptées à la réalisation de simulations géographiquement situées d'agents cognitifs. Elles sont fondées sur un modèle centré agent qui nuit à la réutilisabilité des comportements en ne séparant pas le code de l'agent et le code propre à une simulation. Je propose un modèle centré interaction qui, en séparant les aspects déclaratifs et les aspects procéduraux d'une simulation, permet

de réutiliser séparément les agents, les environnements et les interactions d'une simulation à une autre. Ce modèle est fondé sur la dualité peut-effectuer/peut-subir qui permet de mettre en relation un agent acteur et un agent cible pour exécuter une interaction. Je propose une représentation de connaissance basée sur ce modèle et un moteur de comportement générique capable de manipuler cette connaissance afin de réaliser des simulations de comportements dans lesquelles une importance particulière a été donnée à la rationalité : je montre qu'il est souhaitable de gérer les déplacements au plus tôt, c'est à dire lors de la planification et non à l'exécution. Je propose une solution pour intégrer les déplacements dans la représentation de la connaissance et les gérer dans le moteur de comportement.

La plupart des simulations font appel à plusieurs agents, que ce soit pour des simulations à visée pédagogique comme dans les simulateurs d'entraînement d'équipes d'intervention, à visée scientifique pour l'étude des comportements de foule ou à visée ludique dans le cas des jeux vidéos. Il m'est donc apparu important d'étendre le moteur de comportements aux équipes en gardant les mêmes lignes directrices, à savoir la généricité du moteur, la réutilisabilité et la rationalité des comportements. Je propose un modèle d'équipe hétérogène semi-centralisé dans lequel chaque agent jouit d'une autonomie de décision qui le différencie d'un simple outil qui serait piloté de manière centralisée. Je propose conjointement à ce modèle un algorithme de répartition des tâches fondé sur l'algorithme de Ford-Fulkerson qui s'adapte aux changements de composition de l'équipe afin de profiter optimalement des différentes capacités proposées par les membres de l'équipe, même si la composition de celle-ci varie.

Les différentes contributions de cette thèse (modèle centré interaction, représentation de la connaissance, l'intégration des déplacements dans la planification, la replanification partielle, le modèle d'équipe hétérogène semi-centralisé et la gestion dynamique des capacités de l'équipe) ont débouché sur la création d'une plate-forme de simulation permettant d'éprouver les concepts mis en œuvre et de réaliser des simulations qui exploitent les différentes contributions. Cette plate-forme peut assister le game designer en lui offrant la possibilité de faire des expériences pour tester des comportements et de réutiliser les comportements créés dans d'autres simulations.

Dans une première partie, je commencerai par recenser les problèmes qui se posaient au début de ce travail et qui en sont à l'origine, puis j'en étudierai les domaines connexes et sous-jacents. Nous verrons tout d'abord l'arrivée des systèmes multi-agents dans l'histoire de l'intelligence artificielle. Je commencerai par définir la notion d'agent, de système multi-agent et je présenterai les possibilités offertes par ce modèle. Nous étudierons les différents modes d'interaction que l'on peut rencontrer au sein de ce modèle : la concurrence, la compétition, la collaboration. Nous verrons que ces trois modes d'interaction très différents posent chacun des problèmes qui leur sont propres. J'étudierai ensuite la planification, dont je détaillerai quelques algorithmes. Je m'intéresserai particulièrement à un algorithme de planification qui permet la recherche de chemin dans un environnement géographique comportant des obstacles. Enfin, je présenterai quelques formes de simulations par agents constituées par les plates-formes d'agents, et nous j'approfondirai en particulier le cas du jeu vidéo.

Dans une seconde partie, je détaillerai les différentes contributions. J'étudierai le modèle centré interaction et ses avantages par rapport au modèle classique centré agent, fondé sur la dualité peut-subir/peut-effectuer qui permet de décrire la dynamique d'une simulation indépendamment d'un moteur de comportement. Je présenterai une adaptation de la méthodologie IODA qui aide à la création d'une simulation centrée interaction pour résoudre un problème.

Je m'intéresserai ensuite à la représentation des connaissances de nos agents permettant de décrire les agents eux-mêmes, les interactions qu'il peuvent utiliser, et l'environnement dans lequel ils évoluent. Cela m'amènera à me pencher sur les notions de distances et de déplacement, les problèmes spécifiques qu'ils posent et la représentation choisie permettant d'appréhender des problèmes. Je présenterai ensuite le moteur de comportement des agents, décliné dans trois modes d'interaction : celui d'un agent seul dans l'environnement (contexte mono-agent), agents en concurrence (contexte de concurrence) et agents en équipe (contexte de coalition et collaboration). J'insisterai particulièrement sur la résolution de problèmes d'équipes, et sur la gestion dynamique des capacités des agents de l'équipe. Enfin, je présenterai la plate-forme réalisée afin de valider ce travail, du point de vue de l'implémentation des solutions ainsi que de son utilisation.

Première partie

Contexte

Chapitre 1

Les problèmes à résoudre

« Il est bien connu que l'un des ingrédients essentiels du succès est d'ignorer que ce que l'on entreprend est impossible. »
T. Pratchett, Les annales du disque monde, vol 3

La simulation de comportements dans un environnement géographique comporte de multiples challenges, que ce soit au niveau de la conception ou de la réalisation. En effet plusieurs points imposent une vigilance particulière car ils constituent des *problèmes de recherche durs* : souvent il n'existe pas de solution triviale pour ces problèmes, ou alors si une solution triviale existe, elle ne résout que partiellement le problème. De plus il peut exister plusieurs solutions pour résoudre un de ces problèmes, chacune de ces solutions pouvant être bonne sur certains critères et plus mauvaise sur d'autres, imposant des choix parfois difficiles. Enfin, la combinaison de ces problèmes durs et leur résolution simultanée constituent elles-mêmes un nouveau problème...

1.1 La réutilisabilité des comportements

Bien souvent, les comportements des entités présentes dans les simulations sont créés sans réutiliser des comportements existants et sans le souci de pouvoir les réutiliser ultérieurement. Le code du comportement d'un agent est fondu dans celui-ci et l'expression des interactions qui peuvent se produire entre les agents n'apparaît pas séparément, ce qui ne permet pas de réutiliser ce code facilement. Ce fut la motivation initiale de ce travail : permettre la réutilisabilité des comportements. Il est donc apparu indispensable de séparer les aspects déclaratifs et les aspects procéduraux de la simulation, qui sont respectivement le code des interactions et celui des agents. Cette séparation devrait bien sûr se faire au niveau du modèle, ce que ne propose pas le modèle classique centré agent. C'est donc à la définition même du modèle que nous nous sommes attaqué dans un premier temps : nous avons débouché sur un modèle centré interaction.

1.2 La représentation des connaissances

La réutilisabilité des comportements implique la réutilisabilité des briques de base d'une simulation : il est nécessaire de pouvoir réutiliser les environnements mais surtout des agents et des interactions. Le code des agents doit donc être très abstrait par rapport aux simulations, aux environnements et aux interactions afin qu'un agent puisse être réutilisé sans modification dans une autre simulation. D'autre part, l'expression des interactions se doit d'être abstraite par rapport aux agents et aux environnements : une interaction nommée « ouvrir » doit pouvoir être utilisée pour ouvrir tout ce qui est « ouvrable ». C'est ainsi qu'est apparu le modèle « peut-subir/peut-effectuer » qui permet de caractériser les interactions que les agents peuvent subir ou exécuter dans la simulation. Les interactions obtenues doivent bien sûr être manipulables par l'agent afin qu'il puisse faire un raisonnement à l'aide de celles-ci.

1.3 La crédibilité des comportements et la rationalité

Que ce soit dans l'univers grandissant du jeu vidéo ou dans celui de la simulation de comportements à des fins d'études, les comportements doivent être crédibles aux yeux du joueur ou de l'utilisateur. Dans le premier cas, c'est l'intérêt porté au jeu qui risque de souffrir de la non crédibilité du comportement, tandis que dans le second cas c'est la simulation elle-même et ses résultats qui hériteront de la non crédibilité d'un comportement. La crédibilité d'un comportement dépend étroitement de la rationalité de celui-ci. Il est difficile de juger de la rationalité d'un comportement de manière automatique : la rationalité n'est pas l'optimalité ! De plus, elle est parfois subjective. Nous avons donc choisi de caractériser la rationalité d'un comportement par le fait qu'un être humain confronté à la même situation aurait agi d'une manière sensiblement identique à celle de l'agent. Ainsi la rationalité se définit plutôt par contradiction : un comportement est rationnel lorsqu'il n'apparaît pas clairement comme irrationnel. J'ai tenté de respecter cette contrainte tout au long de ce travail, en particulier en ce qui concerne les déplacements.

1.4 Les contraintes géographiques et la prise en compte des déplacements dans la planification

Habituellement, les simulations situées sont réalisées grâce à une alternance de phases de planification et de phases d'exécution. Les déplacements ne sont considérés que lors de la seconde phase, lorsque que l'agent exécute le plan qu'il vient de calculer. Comme nous allons le voir, cette manière de procéder engendre parfois des déplacements inutiles, qui nuisent à la crédibilité du comportement : c'est par exemple le cas lorsqu'un agent essaie de passer par une porte dont il sait a priori qu'elle est fermée. Ses connaissances lui permettaient d'aller chercher d'abord la clef permettant d'ouvrir cette porte. Pour aboutir à ce genre de raisonnement, il est nécessaire d'intégrer le calcul des déplacements dans la planification. La difficulté principale concerne les obstacles, et les conditions de passage qu'ils imposent. Il faut trouver une manière de représenter ces conditions qui soit compatible avec la représentation de connaissances de l'agent, et qui les rende manipulables afin d'être intégrées dans le calcul du plan.

1.5 La dynamicité

Comme nous l'avons vu en 2.3, l'intervention d'autres agents dans la simulation introduit la non-monotonie de l'environnement. Une information utilisée pour établir un plan peut-être caduque au moment de l'utilisation du plan à cause de la présence des autres agents dans l'environnement. Un agent peut donc rencontrer des échecs à l'exécution : il faut gérer ces échecs (ou les éviter), et permettre à l'agent d'adapter son plan en fonction des évènements qu'il rencontre. La difficulté qui se pose est de savoir quelles sont les informations pertinentes dans le flux d'informations de l'agent, quelles sont les parties du plan qu'il est nécessaire de modifier et quelles sont celles qui ne sont pas affectées.

1.6 Les équipes et l'autonomie de décision

L'intégration des équipes dans le modèle a apporté son lot de problèmes. Assez rapidement, le choix d'une gestion centralisée de l'équipe s'est imposée. Celle-ci possède de nombreux avantages, mais possède l'inconvénient de se montrer trop « directive » : les agents possèdent un moteur de comportement qui leur permet d'établir des raisonnements, il serait dommage de ne les considérer que comme des simples exécutants sous les ordres du chef. Une attention particulière a donc été apportée à l'autonomie des agents. La difficulté consiste à trouver le juste milieu entre la centralisation complète avec des agents exécutants et la décentralisation avec des agents chargés de gérer des problèmes importants, ou face auxquels ils ne sont pas compétents.

1.7 L'utilisation optimale des aptitudes des agents

Dans ce modèle centralisé respectant l'autonomie de décision des agents, il faut pouvoir utiliser au mieux les différentes capacités des agents lorsque l'équipe n'est pas homogène. Même si un agent est complètement polyvalent, il n'est pas optimal de lui confier la réalisation de toutes les tâches de l'équipe sans utiliser les autres agents. Il faut affecter les tâches à réaliser aux agents de manière à effectuer le plus de tâches simultanément. De plus, la composition des équipes est éventuellement dynamique : la liste des membres peut être modifiée pendant l'exécution des tâches par l'équipe. Il est nécessaire de prendre en compte efficacement les aptitudes des nouveaux membres, ou de trouver une manière de réorganiser l'équipe afin de gérer la disparition d'un membre ayant une capacité particulière.

Les solutions proposées

Je présente dans cette partie des solutions que j'ai proposées pour résoudre la majorité de ces problèmes. Dans la même lignée que STRIPS, nous approfondissons la séparation entre le déclaratif (code des interactions) et le procédural (code des agents) dans un modèle centré interaction afin d'augmenter la réutilisabilité des comportements. La question de la rationalité du comportement est résolue en ce qui concerne l'intégration des déplacements dans la planification afin d'éviter des déplacements inutiles. En revanche, certains points comme

l'opportunisme restent à approfondir. Je propose un mécanisme de replanification partielle permettant de gérer la non-monotonie de l'environnement de l'agent. La solution que je propose pour intégrer les équipes d'agents dans le modèle accorde une grande part à l'autonomie de décision des agents, ainsi qu'à l'affectation dynamique des agents afin d'utiliser optimalement leurs capacités, grâce à une propriété de l'algorithme de Ford-Fulkerson, utilisé habituellement pour résoudre des problèmes de flux. J'ai en revanche écarté certains aspects des simulations comme la gestion explicite du temps, la gestion des émotions ou des interactions sociales pour me concentrer sur les questions citées ci-dessus. Certains de ces problèmes qui restent en suspens et les améliorations éventuelles des solutions proposées font d'ores et déjà partie des perspectives qui permettront de prolonger mon travail, que je vais maintenant vous présenter dans le détail.

Chapitre 2

Etat de l'art

« Vous êtes susceptibles d'être critiqués par trois sortes de gens :
- ceux qui font le contraire,
- ceux qui ne font rien,
- et ceux qui font la même chose. »
Louis PAUWELS

2.1 L'histoire de l'Intelligence Artificielle et ses domaines

2.1.1 Un peu d'histoire...

Les premières tentatives de faire exécuter par une machine des travaux qui requièrent de « l'intelligence » ne sont pas récentes. Les automates de calcul du XVII^{e} siècle en sont un bon exemple. Lors de l'invention de la machine à calculer de Pascal en 1642, le fait que l'arithmétique soit reproductible par une machine souleva des interrogations sur la nature de la pensée. Tandis que Pascal et Descartes en déduisaient que l'arithmétique échappait au domaine du raisonnement, Hobbes en concluait que le raisonnement n'était qu'un calcul mécanique. Evidemment, les problèmes liés à la mécanique ne permirent pas de construire autre chose que des automates, et c'est l'arrivée de l'informatique au milieu du XX^{e} siècle qui permit à l'intelligence artificielle de voir le jour.

En 1928, John Von Neumann mit au point l'algorithme MinMax qui s'applique aux jeux à deux joueurs à somme nulle. Cet algorithme est à l'origine d'évolutions (alphabeta, negascout) aujourd'hui utilisées par les joueurs d'échecs artificiels, bien que complétées par d'autres méthodes (bibliothèques d'ouvertures et de fermetures, ...). Il est intéressant de noter que cet algorithme a été inventé en dehors de tout contexte informatique !

L'un des premiers programmes d'intelligence artificielle fut *Logic Theorist* en 1956 par Allan Newell, Herbert Simon et Cliff Shaw. Ce programme de démonstration automatique de théorèmes partait d'une hypothèse de base (axiome) en y appliquant des règles (théorèmes) afin de démontrer des théorèmes nouveaux, ne faisant pas partie de ses connaissances initiales.

L'application systématique des théorèmes produisait un arbre dont la taille devenait rapidement ingérable, à cause de l'explosion combinatoire propre aux algorithmes de chaînage avant. Des heuristiques furent mises en place de manière empirique afin de guider la recherche dans l'arbre des théorèmes en sélectionnant les branches susceptibles d'aboutir au résultat souhaité. Ce programme fut un succès car il se montra capable de démontrer 38 des 52 théorèmes du second chapitre des « Principia mathematica » de Bertrand Russell et Alfred North Whitehead. Pour l'anecdote, l'un des théorèmes fut démontré d'une manière jugée plus élégante par le programme que la démonstration dans l'ouvrage de Russel et Whitehead.

L'apparition des langages *IPL* en 1956 puis *LISP* en 1958 aidèrent au développement de l'IA par leur traitement de listes. L'importance de ce langage prit toute sa mesure en 1970 lorsqu'il fut possible de le compiler plutôt que de l'interpréter.

En 1958, un premier modèle de réseau de neurones voit le jour : c'est le perceptron, créé par Franck Rosenblatt. Il est inspiré du système visuel, et il fonctionne grâce à des corrections d'erreurs des poids de ses connexions synaptiques. Il est composé de trois couches : la couche sensorielle, ou rétine, sur laquelle sont envoyés les stimuli, la couche d'associations, et la couche de décision qui fournit la réponse du perceptron. En 1969, les recherches sur ce modèle prometteur furent stoppées suite à l'ouvrage *Perceptrons* de Marvin Lee Minsky et Seymour Papert, dans lequel les limites de ce modèle étaient démontrées. Pourtant aujourd'hui, une évolution de ce modèle (les réseaux de neurones multicouches) est très utilisée pour la reconnaissance d'écriture, de la voix ou des visages.

Plusieurs tentatives de créer une IA calquée sur le raisonnement humain apparurent : *GPS* (General Problem Solver) en 1957 s'inspirait d'expériences de psychologie pour créer un système de résolution global grâce à un ensemble de règles heuristiques génériques. *GTP* (Geometry Theorem Prover) en 1959 partait d'un théorème à démontrer pour remonter jusqu'aux axiomes ou à des théorèmes connus. Le programme élaguait l'arbre de recherche en ne considérant que les propriétés qui étaient vérifiées sur la figure géométrique sur laquelle il raisonnait, comme le font intuitivement les humains. *Analogy* en 1963 tenta de faire un programme de résolution essayant de trouver des analogies entre les problèmes, en se fondant sur l'hypothèse que les humains fonctionnent de cette manière, et non à l'aide d'un raisonnement logique.

En 1969, l'algorithme *Planner* ouvre la voie des planificateurs : il sera suivi entre autres par *Graphplan* [BF95], *Satplan* [KS92] et d'autres. Je ne m'étends pas ici sur les planificateurs qui seront étudiés plus loin dans le document.

Le langage *Prolog* (PROgrammation LOGique) inventé en 1972 par A. Colmerauer et P. Roussel était conçu à l'origine pour l'interprétation du langage naturel, mais il est aujourd'hui utilisé pour résoudre divers problèmes d'IA. C'est un langage déclaratif basé sur la logique du premier ordre dans lequel on décrit des faits et des règles. Le programme peut répondre à une question posée en appliquant les règles sur les faits en utilisant les concepts d'unification, de récursivité et de retour sur trace (backtracking).

Enfin, parmi tous les algorithmes et programmes d'IA, il est important de citer l'$A*$ de Dijkstra, initialement prévu pour trouver le plus court chemin dans un graphe, mais qui trouve de nombreuses applications en IA : il permet de résoudre des problèmes comme le taquin, mais surtout, il est très utilisé pour calculer les déplacements des PNJ (Personnages Non Joueurs)

dans les jeux vidéo. Je l'utilise d'ailleurs dans mon travail, il fera donc l'objet d'une étude particulière.

On ne saurait bien sûr parler d'intelligence artificielle sans le célèbre test de Turing. En 1950 Turing élabora un test permettant de savoir si une machine était capable de penser [Tur50]. Son test était basé sur un jeu d'imitation, dans lequel un invité devait converser avec un homme et une femme sans les voir, par le biais de messages écrits. Le but de l'invité était de trouver qui était l'homme et qui était la femme, en leur posant des questions. L'homme devait se faire passer pour la femme, tandis que la femme devait convaincre l'invité que c'était bien elle la femme. Turing avait décrété que l'on pourrait conclure qu'une machine serait intelligente lorsqu'elle pourrait prendre la place de l'homme et qu'il ne soit pas possible à l'invité de l'identifier dans plus de 70% des cas après 5 minutes de conversation. Plus généralement aujourd'hui, le test de Turing se résume à la conversation avec une machine seule qui doit se faire passer pour un humain via un terminal. Turing avait prédit qu'avant la fin du siècle dernier une machine aurait passé son test avec succès... on sait aujourd'hui que sa prévision était fort optimiste. Le CAPTCHA[1] (Completely Automated Public Turing test to Tell Computers and Humans Apart), dérivé du test de Turing, au moins au niveau du concept, est d'ailleurs souvent utilisé sur internet : en demandant à l'utilisateur de saisir le texte apparaissant sur une image déformée, il permet d'interdire l'accès à des programmes malveillants.

Au fil des avancées, la frontière de l'intelligence artificielle semble inexorablement reculer : certains rangent a posteriori les problèmes résolus dans la catégorie des problèmes qui ne requièrent pas d'intelligence, comme s'ils refusaient de voir l'intelligence humaine comme un processus calculatoire imitable par une machine.

2.1.2 Les domaines d'application de l'IA

Les domaines d'application de l'intelligence artificielle sont très variés :

- **l'aide à la décision** grâce par exemple aux systèmes experts qui imitent le raisonnement d'un expert dans un domaine particulier, ils sont efficaces pour établir des diagnostics. L'aide à la décision utilise aussi des techniques d'apprentissage variées (classifieurs, réseaux de neurones...)
- **l'apprentissage** permet à un programme de collecter lui-même ses données de travail, ou de tirer profit de ses expériences.
- **le calcul formel** permet de traiter des expressions symboliques (résolution de systèmes d'équations, calcul algébrique, équations différentielles...)
- **la simulation du raisonnement humain** permet de reproduire certains raisonnements humains. Des logiques permettent de formaliser ces modes de raisonnement (logique floue, logique modale, logique temporelle...)
- **le traitement du langage naturel** permet de dégager le sens d'un texte afin de le traduire dans une autre langue, de converser...
- **la résolution de problème** est un domaine assez vaste qui regroupe les jeux (backgammon, échecs, go), des problèmes d'emploi du temps, les problèmes de planification

[1] http ://www.captcha.net/

- **la reconnaissance de formes** utilisée pour analyser l'écriture ou la parole, reconnaître des visages, utilise souvent des réseaux de neurones permettant une reconnaissance "floue"
- **la robotique** est très utilisée dans l'industrie pour réaliser des tâches répétitives ou dangereuses. Il existe des robots simples effectuant des tâches programmées, et des robots plus autonomes qui prennent des décisions en fonction de leur environnement (exemple : le projet Pathfinder pour l'exploration de la planète Mars).
- **les systèmes complexes adaptatifs** qui étudient l'évolution de populations qui s'organisent à partir de règles simples.
- **les jeux vidéo** ne doivent pas être oubliés : ils sont actuellement très demandeurs de comportements intelligents afin de parfaire le réalisme
- **le cinéma** utilise depuis peu les techniques multi-agents pour réaliser certaines scènes, comme la scène du *gouffre de Helm* dans le film *Le Seigneur des Anneaux* réalisé grâce au logiciel Massive.

C'est à la suite de cette longue évolution qu'est apparue la notion d'agent, lorsque les capacités de décision conférées à un programme le rendirent de plus en plus autonome : on est passé du concept de l'intelligence artificielle distribuée à celui du système multi-agent.

2.2 Les agents et les SMA

La notion d'agent permet de qualifier une entité logicielle ayant ses propres capacités de raisonnement. D'après Ferber [Fer95a], "on appelle agent intelligent une entité réelle ou abstraite qui est capable d'agir sur elle-même et sur son environnement, qui dispose d'une représentation partielle de cet environnement, qui, dans un univers multi-agent, peut communiquer avec d'autres agents et dont le comportement est la conséquence de ses observations, de sa connaissance et des interactions avec les autres agents."

Plus précisément, Ferber leur attribue un ensemble de propriétés [Fer95b]. Un agent est une entité :

1. qui est capable d'agir dans un environnement ;
2. qui peut communiquer directement ou non avec les autres agents ;
3. qui peut être menée par un ensemble de tendances (sous la forme d'objectifs individuels ou de fonction de satisfaction/survie qu'elle essaie d'optimiser) ;
4. qui possède des ressources propres ;
5. qui est capable de percevoir son environnement (mais de manière limitée) ;
6. qui n'a qu'une représentation partielle de son environnement (et sans doute n'en n'a-t-elle aucune) ;
7. qui possède des compétences et peut offrir des ressources ;
8. qui peut être capable de se reproduire ;
9. et dont le comportement tend à satisfaire ses objectifs, en tenant compte des ressources et des compétences qu'elle possède tout en dépendant de ses perceptions, de ses représentations et des communications qu'elle reçoit.

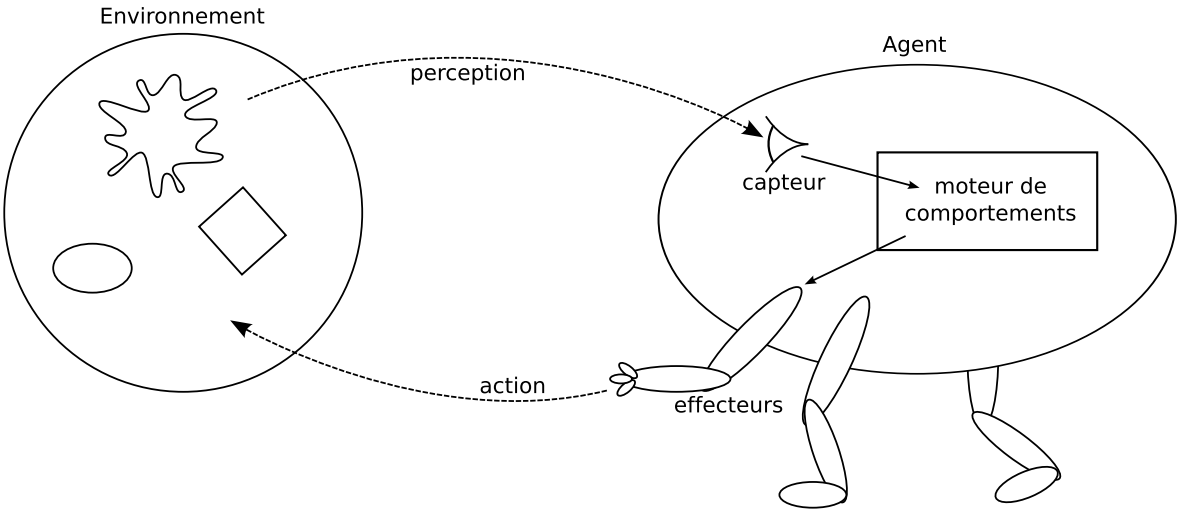


FIG. 2.1 – Modèle général d'un agent : il possède des capteurs pour prendre des informations dans l'environnement, et des effecteurs pour agir dans l'environnement. C'est grâce à son moteur de comportement que l'agent effectue des actions en fonction de ses perceptions. (figure originale reproduite de [RN02])

Cette définition est discutée car toutes ces propriétés ne sont pas nécessaires pour réaliser un agent. Par exemple, la capacité de reproduction d'un agent, si elle a un sens lorsqu'il s'agit d'un agent logiciel, s'envisage plus difficilement si l'on considère un agent embarqué dans un robot : la réplication physique du robot pose problème !

Un agent est capable de raisonner sur son environnement, mais aussi d'agir sur celui-ci (figure 2.1). Ceci a pour effet de modifier l'environnement et donc d'influencer ses futures décisions ainsi que celles des autre agents éventuellement présents dans l'environnement. Un agent est autonome : la totalité de son comportement n'est pas codée explicitement : on ne code dans l'agent que les outils qui lui permettent de manipuler la connaissance afin de résoudre un but. La majorité des actions entreprises par un agent est le fruit de sa « réflexion ».

Selon Jennings [Jen00], il est probable que les agents constituent durant les années à venir un nouveau paradigme de programmation, a l'instar des objets. On pourrait donc voir apparaître une *programmation orientée agents*. Les agents et les objets, s'ils sont similaires sur quelques points, sont pourtant d'une nature assez différente. Il est vrai que les agents, tout comme les objets, possèdent un état interne qui évoluent tout au long du programme, et qu'ils représentent une entité qui a un sens dans le programme. Toutefois, les méthodes d'un objet doivent être invoquées par un objet extérieur, alors que celles d'un agent sont invoquées si l'agent le juge nécessaire : le contrôle est effectué par l'agent lui même, car celui-ci possède un comportement, à la différence d'un objet. Une autre différence fondamentale réside dans la décentralisation du contrôle : chacun des agents est lui-même une source de contrôle dans le programme, alors que les objets sont contrôlés par une source unique.

Les agents sont actuellement très en vogue. De plus en plus, on en voit apparaître dans les applications ou sur internet : ces agents peuvent être très simplistes en prenant juste la forme

d'un avatar animé comme le chien qui aide à la recherche de documents dans l'explorateur de fichiers de Windows, un peu plus sophistiqués comme le trombone d'une suite logicielle connue, ou encore plus élaborés comme les agents conversationnels que l'on rencontre sur les sites commerciaux. Le terme *agent* regroupe aussi certains logiciels résidents permettant de faire des recherches rapides, ou d'effectuer des fonctions en tâche de fond. C'est le cas du *Winamp Agent*, des agents anti-virus ou firewall, etc.

Les agents peuvent être classés dans deux catégories en fonction de la façon dont ils essayent d'atteindre leurs buts. D'une manière générale, il y a deux possibilités pour atteindre un but : la première consiste à effectuer des changements dans le monde jusqu'à l'obtention de l'état désiré, la seconde consiste à calculer, à partir de l'état souhaité, les opérations à effectuer. C'est ce qui différencie les agents réactifs des agents cognitifs.

2.2.1 Les agents réactifs

On dit qu'un agent est réactif lorsqu'il agit uniquement en fonction de stimuli (internes ou externes) sans avoir besoin d'une représentation symbolique ou historique. On parle dans ce cas de comportement *behavioriste*.

Une possibilité pour réaliser un agent réactif est l'utilisation d'une architecture de subsomption [Bro86] dont un exemple[2] est présenté en figure 2.2. Cette architecture est composée de plusieurs modules ayant chacun une tâche à réaliser. Ces modules sont organisés hiérarchiquement en fonction de l'importance de la tâche qu'ils réalisent. Chaque module choisit une action à réaliser en fonction de ce qui est perçu dans l'environnement. Plusieurs modules peuvent avoir choisi une action différente, et pour décider de l'action qui sera effectuée, on se réfère à la priorité du module. Le comportement des agents réactifs est donc essentiellement induit par les modifications de l'environnement. On peut résumer l'architecture de subsomption pour cet agent à un ensemble de règles ordonnées avec le code de la figure 2.3.

Ces agents ne possèdent généralement pas de modèle de leur environnement. Ceci les oblige donc à utiliser des informations locales afin de guider le choix de la prochaine action qu'ils effectueront. Le comportement d'un agent réactif n'est pas codé explicitement, mais découle du déclenchement de règles simples en fonction des modifications de l'environnement, des autres agents ou de l'agent lui-même. Ceci le rend particulièrement difficile à configurer pour lui faire effectuer une tâche spécifique.

Toutefois les agents réactifs sont souvent utilisés en groupe, au sein duquel on peut observer le phénomène d'émergence [CGGG02]. Marvin Minsky nous en donne la définition suivante dans [Min86] : *Apparition inattendue, à partir d'un système complexe, d'un phénomène qui n'avait pas semblé inhérent aux différentes parties de ce système. Ces phénomènes émergents ou collectifs montrent qu'un tout peut-être supérieur à la somme de ses parties.* Parmi les réalisations les plus spectaculaires avec des agents réactifs, on peut citer les travaux de Craig Reynolds sur le flocking [Rey87]. Les agents réactifs utilisés dans ces travaux pour simuler les animaux sont appelés *boids*[3]. En donnant à des agents réactifs quelques règles simples, il

[2] extrait de `http://www.limsi.fr/~jps/enseignement/examsma/2005/4.apprentissage/MaryFelkin/MaryFelkin.html`

[3] Ce terme est la contraction de birdoid que l'on pourrait traduire par oisoïde.

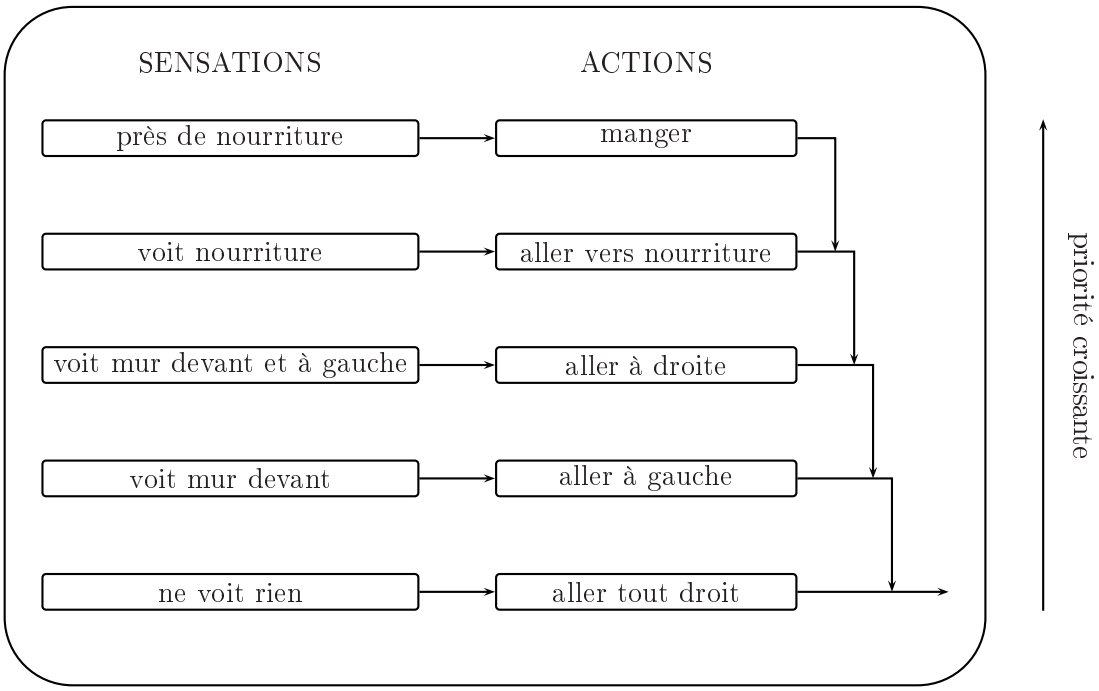


FIG. 2.2 – Architecture de subsomption : schéma d'un agent forageur dont le comportement se résume à récolter de la nourriture.

est possible de modéliser et de simuler des déplacements de groupes d'animaux, tels certains oiseaux ou poissons. Voici les trois règles qui permettent d'effectuer ce genre de simulation :

séparation : tourner de manière à s'éloigner des congénères trop proches afin d'éviter les collisions

alignement : adopter la direction moyenne des individus proches

cohésion : se rapprocher de la position moyenne des individus proches

Les agents qui se trouvent au delà d'un certain rayon d'influence ne sont pas pris en compte dans les calculs. Le comportement obtenu est réaliste, et remplit une des caractéristiques fondamentales du vivant : il est imprévisible. En effet, même s'il est possible de prévoir à très court terme la direction que va prendre un agent, il est en revanche impossible de savoir à plus long terme quelle sera sa position dans le groupe d'agents.

L'interaction de plusieurs agents réactifs permet l'émergence d'une intelligence collective. Ce principe est utilisé dans la théorie des AMAS [GGG03] : dans ce système, les agents sont programmés pour chercher à être les plus coopératifs possibles avec les autres agents, ce qui a pour effet de faire émerger la fonction du système sans qu'aucun agent n'ait explicitement connaissance de celle-ci. On doit simplement fournir à chaque agent une fonction partielle à réaliser dans le système. Toute la problématique de l'émergence réside dans l'écriture de cette fonction : d'une certaine manière, on peut dire que la fonction globale du système est cachée dans la fonction individuelle des agents. Bien que le concepteur de l'agent se défende de coder implicitement la fonction globale (ou une partie de celle-ci) dans les agents, il oriente toutefois le comportement du groupe en créant le comportement des agents. En outre, il est très difficile

```
SI (près de nourriture) ALORS manger
SINON
{
    SI (voit nourriture) ALORS aller vers nourriture
    SINON
    {
        SI (voit un mur devant et à gauche) ALORS tourner à droite
        SINON
        {
            SI (voit un mur devant) ALORS tourner à gauche
            SINON je vais tout droit
        }
    }
}
```

FIG. 2.3 – Ce code de l'agent forageur est un exemple d'implémentation de l'architecture de subsomption présentée en figure 2.2

de construire une fonction partielle permettant de voir émerger la fonction globale souhaitée au niveau du groupe d'agents.

2.2.2 Les agents cognitifs

Le schéma global d'un agent cognitif est semblable à celui d'un agent réactif : il est constitué d'un côté de capteurs qui lui permettent de percevoir l'environnement et d'en retirer des informations, et de l'autre d'un ensemble d'effecteurs dont le rôle est de déclencher des actions afin de modifier l'environnement de manière à avancer dans la résolution de ses buts. La différence majeure réside dans le moteur de comportement : l'agent cognitif a un comportement orienté par les buts : il est capable de trouver une action qui l'amène à un état désiré. Cet état peut être celui qui constitue le but de l'agent, ou un état intermédiaire qui s'approche de l'état souhaité. L'agent cognitif utilise à cette fin un solveur de problèmes qui trouve une séquence d'actions dont l'exécution approche successivement l'agent de l'état souhaité. Le solveur de problèmes est capable de raisonner sur l'environnement et en particulier sur les conséquences d'une action ou d'une suite d'actions. Ce raisonnement s'appuie sur une représentation symbolique de l'environnement et des actions que l'agent peut déclencher dans l'environnement. Classiquement, les actions prennent la forme d'un triplet *précondition, action, effet* très adaptées pour un solveur de problèmes de type chaînage arrière.

2.2.3 Comparaison réactif/cognitif

Malgré ces ressemblances structurelles, le fonctionnement des agents réactifs et celui des agents cognitifs sont finalement assez radicalement opposés : le moteur des premiers ne leur permet qu'un raisonnement à court terme (si l'on peut d'ailleurs parler de raisonnement dans ce cas). Les seconds ont un raisonnement et donc un comportement plus orienté long terme,

dans la mesure où ils déclenchent des actions dont l'effet permet soit de s'approcher du but soit de s'en approcher plus facilement ensuite.

D'une certaine manière, en opposition au chaînage arrière des agents cognitifs, on pourrait voir le moteur de comportement des agents réactifs comme un chaînage avant. Alors que les agents cognitifs établissent un raisonnement partant de l'état à obtenir pour aller vers l'état actuel, les agents réactifs partent de l'état actuel et appliquent des opérateurs lorsque cela est possible jusqu'à ce qu'ils atteignent l'état final. La résolution d'un but chez un agent cognitif est le fruit d'un comportement volontaire : toutes ses actions sont effectuées pour avancer dans la résolution du but. Chez l'agent réactif, le déclenchement d'actions ne fait pas forcément avancer dans la résolution du but.

Lorsque l'on tente de les comparer, il est souvent question de l'explosion combinatoire de la résolution de problème chez l'agent cognitif, alors qu'elle n'existe pas chez l'agent réactif, plus léger. La description du problème et le modèle utilisé permettent toutefois de contenir cette explosion. Les agents cognitifs possèdent l'avantage d'être plus polyvalents. En effet, un agent réactif est souvent « câblé » pour exécuter une tâche déterminée, alors que le moteur de comportement d'un agent cognitif doit lui permettre de résoudre une plus grande variété de problèmes. De plus, l'agent cognitif est capable de raisonner sur les états antérieurs (historique) et sur une représentation interne de l'environnement, ce qui lui confère des comportements plus évolués et plus variés.

2.2.4 Les systèmes multi-agents

Un système multi-agents (SMA) est un système distribué composé d'entités logicielles qui communiquent appelées agents. Il s'agit d'un système car l'ensemble des agents forme un tout cohérent autour d'un point commun. Ce point peut être un but commun, un langage ou encore un environnement. Un système multi-agents ne comporte aucun contrôle global : chaque agent du système a une vision partielle ou globale du problème à traiter, et des capacités de résolution limitées. Les agents sont dotés de capacités de coopération et de coordination qui leur permettent d'interagir pour résoudre un problème ensemble en formant des coalitions. Les données du problème à traiter sont distribuées parmi les agents, à la différence d'un système centralisé. Les SMA sont confrontés à plusieurs problèmes :

- description du problème ou transformation de celui-ci afin de l'adapter
- récolte et interprétation du résultat fourni par le SMA
- communication entre les agents : quelles informations, comment ?
- organisation des agents afin qu'ils agissent de manière cohérente
- représentation des informations (état du monde, plan...)
- résolution des conflits entre les agents

La communication des systèmes multi-agents peut être de plusieurs types : les systèmes multi-agents utilisant des agents réactifs optent souvent pour la communication par signaux, généralement déposés dans l'environnement, tandis que les systèmes multi-agents cognitifs ont plus volontiers recours à un langage (par exemple ACL de la FIPA...) puisque leurs capacités cognitives leur en permettent l'analyse. Les systèmes multi-agents sont souvent plongés au sein d'un environnement explicite ou implicite. Celui-ci peut se trouver sous plusieurs formes :

environnement géographique (explicite), environnement social (implicite), marché financier (explicite ou implicite) [DBBM07]...

Le système multi-agent peut-être étudié à différents niveaux (figure 2.4). Au niveau (1), au peut étudier le comportement global du système, ou la fonction émergente du système. Au niveau (2) on peut s'intéresser aux interactions qui se produisent entre les agents, à leur comportement social. Le niveau (3) permet de s'attacher au comportement individuel de l'agent au sein du système afin de comprendre finement son rôle dans le système.

2.3 Le Multi-Agent : les modes d'interaction

L'intelligence artificielle permet de doter une entité artificielle d'un comportement relevant de l'intelligence. Lorsque l'on veut réaliser une simulation dans laquelle interviennent plusieurs entités, on ne peut se permettre de le faire en multipliant simplement le nombre d'entités sans reconsidérer leur comportement : une simulation multi-agent ne se résume pas à plusieurs simulations mono-agent simultanées. En effet, une entité ne peut faire abstraction de la présence de ses congénères : ceux-ci sont susceptibles d'entrer en interaction avec elle, que ce soit le fait d'une simple interférence dans le déroulement de ses actions, ou que ce soit le fait d'une interaction délibérée.

Les sports collectifs constituent un exemple intéressant de système multi-agent car ils en cumulent plusieurs modes de fonctionnement. Dans ces sports, deux équipes s'affrontent lors d'un match et l'équipe qui marque le plus de points remporte le match. Au sein d'une équipe, les joueurs suivent une stratégie collective décidée par le capitaine et l'adaptent en fonction de la situation de jeu qu'ils rencontrent et selon leur degré d'autonomie. Leurs performances individuelles lors d'un match leurs permettent en outre d'être repérés par le sélectionneur pour le match ou le tournoi suivant. Nous avons cité ici trois modes d'interaction entre entités dans un système multi-agent : la compétition (entre les deux équipes), la coopération (au sein de l'équipe) et la concurrence (entre les joueurs). Nous allons détailler ces différents modes.

2.3.1 La concurrence et la compétition

La présence simultanée de deux agents dans un environnement implique presque toujours des problèmes de concurrence ou de compétition. Bien que les définitions de ces deux termes soient assez proches dans le langage courant, ils impliquent pourtant deux notions qui se distinguent par l'intentionnalité. La concurrence est l'état de fait dans lequel se trouvent les deux agents : ils partagent le même espace et les mêmes ressources (nourriture, objets, espace physique, temps processeur ou encore bande passante). Une ressource ne peut généralement être utilisée que par un seul agent à la fois, et le fait que l'un des agents utilise une ressource pénalise *involontairement* l'autre agent puisqu'il ne peut bénéficier de cette ressource pendant ce temps, voire jamais si c'est une ressource qui est consommée lorsqu'on l'utilise. Si l'on résume le but d'un agent à la maximisation d'une fonction d'utilité, il est donc possible que la maximisation de la fonction d'un agent se fasse au détriment des autres agents qui ne peuvent pas maximiser la leur, bien que ce ne soit pas intentionnel de la part du premier.

Parfois le but des agents est de maximiser l'écart entre leur fonction d'utilité et celle des autres agents, on parle de compétition. Dans ce cas, l'action d'un agent n'est pas obligatoire-

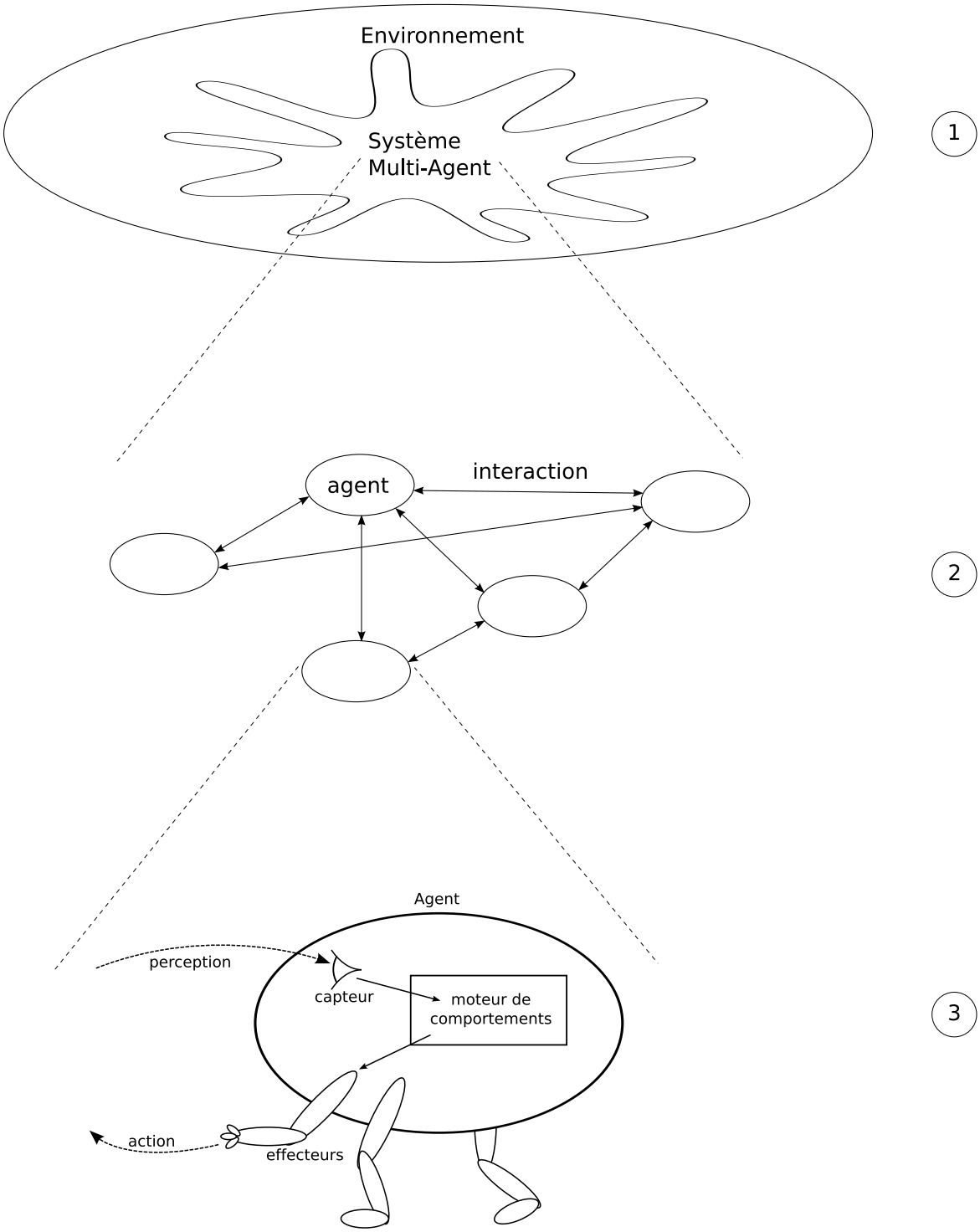


FIG. 2.4 – Les niveaux d'observation du système multi-agent

ment destinée à maximiser sa propre fonction d'utilité, mais peut avoir pour but de diminuer (ou d'empêcher d'augmenter) *volontairement* celle d'un ou de plusieurs autres agents.

Les tableaux 2.1 et 2.2 illustrent les différences d'attitude par rapport à une ressource R d'un agent en concurrence ou en compétition. Qu'il soit dans un contexte de concurrence ou de compétition, un agent utilise une ressource quand il en a besoin. Dans un contexte de concurrence, il libère la ressource le plus rapidement possible si un autre agent en a besoin, alors qu'il la conserve dans un contexte de compétition afin de pénaliser l'autre agent, s'il a conscience que ce dernier en a besoin. Lorsqu'un agent n'a pas besoin d'une ressource, il n'a aucune raison de la prendre dans un contexte de concurrence. S'il possède la ressource et qu'un autre agent en a besoin, il la libère. En revanche, dans un contexte de compétition, un agent qui n'a pas besoin d'une ressource peut monopoliser celle-ci même s'il n'en a pas besoin dans le seul but de pénaliser un ou plusieurs autres agents en les empêchant d'en bénéficier.

Agent \ Concurrent	a besoin de R	n'a pas besoin de R
a besoin de R	appropriation → utilisation → libération	appropriation → utilisation
n'a pas besoin de R	libération	-

TAB. 2.1 – Attitude d'un agent par rapport à une ressource dans le cas de la concurrence

Agent \ Adversaire	a besoin de R	n'a pas besoin de R
a besoin de R	appropriation → utilisation → rétention	appropriation → utilisation
n'a pas besoin de R	appropriation → rétention	-

TAB. 2.2 – Attitude d'un agent par rapport à une ressource dans le cas de la compétition

Dès qu'il est dans un contexte de concurrence et a fortiori de compétition, un agent doit tenir compte de cet état de fait. Notamment, il ne peut tenir pour vraie une information qu'il a relevée dans l'environnement et stockée dans sa mémoire. En effet, un autre agent est susceptible d'avoir modifié cette information à tout moment. Les agents doivent donc sans cesse remettre en cause les informations contenues dans la mémoire, et réviser leur plan en conséquence afin de les adapter aux modifications qui ont été apportées à l'environnement, afin d'éviter un éventuel échec à l'exécution lorsque leur plan n'est plus compatible avec l'état de l'environnement.

La compétition peut se présenter entre deux ou plusieurs agents, mais elle peut aussi se présenter entre des équipes d'agents, par exemple dans les sports collectifs où une équipe doit marquer plus de points que l'équipe adverse. On trouve de telles équipes d'agents dans la compétition robocup [KTS+97] [Rob] qui consiste à créer une équipe de robots qui jouent au football. La notion d'équipe introduit un nouvelle forme d'interaction entre les agents : la collaboration.

2.3.2 La collaboration

Il existe principalement deux raisons d'effectuer un travail en commun. La première raison, la plus évidente, est le fait que la charge de travail est divisée par le nombre de protagonistes. La seconde raison est qu'un agent peut avoir besoin d'un confrère parce que ce dernier a des compétences qu'il n'a pas. Ainsi, travailler à plusieurs peut se faire par la parallélisation d'un travail si les agents sont polyvalents, en travaillant à la chaîne si les agents sont très spécialisés, ou par une autre organisation répartissant les tâches au mieux entre les agents en fonction de leurs aptitudes. D'autres raisons permettent d'expliquer la nécessité de former un groupe : dans le règne animal, l'adage « l'union fait la force » exprime la raison pour laquelle les loups chassent en meute, ou pourquoi les poissons se déplacent en banc : dans le premier cas il s'agit d'acculer la proie ou de l'encercler (mécanisme d'attaque), dans le second cas il s'agit de paraître plus imposant aux yeux du prédateur (mécanisme de défense). Enfin, le fait d'agir en groupe peut faire réaliser des économies à chacun des membres du groupe. Ceci explique le vol en formation « en V » de certains oiseaux migrateurs : la dépression créée par l'aile de l'oiseau précédent permet d'avancer en dépensant moins d'énergie. Certains coûts fixes peuvent aussi justifier la motivation à se regrouper : c'est le cas lorsqu'on effectue une commande en groupe pour partager les frais de port. Enfin, la collaboration peut être nécessaire pour faire face aux problèmes de concurrence en essayant de gérer les ressources équitablement : c'est l'un des fondements de la société.

Effectuer une tâche en groupe peut donc se révéler bénéfique à chacun des individus du groupe. Toutefois, certains problèmes nécessitent la coordination des agents ou leur synchronisation. La coordination des agents est nécessaire pour que les actions de chacun soient cohérentes par rapport au but du groupe. La synchronisation est un cas particulier de coordination qui exige une cohérence des agents dans le temps. Par exemple, s'il s'agit de déplacer un objet trop lourd pour être déplacé par un agent seul, les agents doivent non seulement soulever l'objet simultanément (synchronisation) mais aussi aller dans la même direction (coordination). Les problèmes de synchronisation ou de coordination impliquent généralement le recours à une forme de communication. Ils peuvent être résolus de manière centralisée à l'aide d'un chef qui joue le rôle de coordonnateur, ou de manière décentralisée. Pour cela, on peut par exemple utiliser des réseaux de Petri [EH96] : les plans des agents sont représentés par des réseaux de Petri dont certaines places sont partagées entre deux agents. On peut ainsi s'assurer qu'une action ne sera entreprise que par un seul agent (si l'un d'eux consomme le jeton les autres ne peuvent plus déclencher la transition) ou réaliser un point de synchronisation en attendant que chaque agent ait ajouté son jeton dans une place partagée.

2.3.3 L'émergence

Parfois des entités œuvrent dans un but commun sans avoir connaissance de celui-ci. C'est le cas par exemple de la formation d'un tas de bois chez les termites dont on peut voir une excellente simulation dans la plate-forme NetLogo (cf 2.8.2). Le comportement individuel de chaque termite est très simple : il consiste à ramasser un morceau de bois puis à errer dans l'environnement et à déposer le morceau de bois à côté du prochain morceau de bois rencontré. Dans l'expression de ce comportement, rien n'indique le but final de l'agent. Pourtant, lorsque l'on fait fonctionner une telle simulation, elle aboutit toujours à la formation d'un (parfois deux) tas de bois. La collaboration des agents est involontaire puisqu'elle n'est pas exprimée

dans leur comportement. De plus, les agents n'ont aucune connaissance de la fonction globale du système. On parle dans ce cas d'émergence de la fonction collective. Dans le cas d'une simulation en environnement situé, l'émergence est généralement fondée sur la stigmergie : il s'agit d'une communication indirecte basée sur de petites modifications de l'environnement comme le changement de position des morceaux de bois dans le cas des termites ou le dépôt de phéromones dans le cas des fourmis. On peut aussi citer le cas des des boids dont il est question dans la section 2.2.1 : il est possible de programmer le déplacement d'une nuée d'oiseaux de manière totalement décentralisée et sans communication explicite.

L'émergence est une forme de travail en groupe ne faisant pas intervenir la notion d'équipe. Pourtant l'équipe en tant que structure explicite peut être précieuse pour permettre de gérer efficacement le groupe d'agents, ses buts individuels et le but collectif du groupe.

2.3.4 Les équipes

Les agents qui travaillent dans un but commun (qu'il soit connu d'eux ou non) peuvent se regrouper pour former une équipe. Les regroupements d'agents peuvent prendre plusieurs formes en fonction de leurs besoins individuels et collectifs, nous présentons les différentes solutions en 2.4.

Cette gestion de l'équipe, de ses membres et de ses buts peut être centralisée ou non. Dans le cas d'une gestion totalement centralisée, un chef coordonne les actions de l'équipe, en recrute éventuellement les membres et répartit les tâches entre les agents. Dans le cas contraire, l'équipe peut fonctionner grâce à une auto-organisation de ses membres, sans que l'un des agents ne joue le rôle de coordonnateur. Si la centralisation paraît plus simple à mettre en œuvre (il est plus intuitif de penser ou de programmer "centralisé" que "réparti"), elle pose pourtant des problèmes : la communication vers le chef peut aboutir à un goulot d'étranglement car il faut centraliser les informations, le chef doit avoir une grande capacité de calcul puisqu'il établit les plans individuels et gère la stratégie de l'équipe et enfin le chef constitue un point névralgique important : s'il disparaît pour une quelconque raison, l'équipe ne fonctionne plus. La décentralisation complète, si elle n'est pas affectée par ces points, pose en revanche d'autres problèmes : transmission d'un message à tous les agents en évitant les répétitions [NTCS99], répartition des tâches, coordination des plans individuels et synchronisation... Entre ces deux modes de fonctionnement, on peut avoir différents degrés de commandement : par exemple le coordonnateur peut intervenir partiellement dans la gestion de l'équipe, le reste étant auto-organisé. Les différentes formes de coopération sont présentées en 2.5.

Le fonctionnement d'une équipe peut varier considérablement en fonction de la répartition des aptitudes au sein de l'équipe, du degré d'implication des membres ainsi que de leur capacités de communication. Nous allons détailler ces quelques points.

Répartition des aptitudes

Une équipe peut être homogène ou hétérogène. On parle d'équipe homogène lorsque tous les agents qui la composent ont les mêmes aptitudes, et d'équipe hétérogène si les aptitudes sont distribuées entre les agents. Le cas le plus simple à gérer est celui de l'équipe homogène :

n'importe quel agent de l'équipe peut remplacer n'importe quel autre agent pour réaliser une tâche. La gestion des inscriptions dans l'équipe ou des défections en est très simple. Dans le cas d'une équipe hétérogène, la gestion des membres est beaucoup plus complexe : dans un souci d'optimalité, on essaiera d'exploiter au mieux les capacités des agents afin de réaliser un maximum de tâches simultanément et de laisser le moins d'agents inactifs possible.

Degré d'implication des membres de l'équipe

Une autre particularité peut singulièrement compliquer la gestion des buts dans une équipe : il s'agit du degré d'implication des membres dans l'équipe. En effet, les agents peuvent avoir un degré d'implication variable en fonction de ce qu'ils sont prêts à sacrifier pour l'équipe. Si l'on considère le cas des insectes sociaux, le survie de l'équipe est bien souvent prioritaire par rapport à la survie de l'individu, et il est courant que l'un d'entre eux se sacrifie pour le groupe (abeille qui meurt après avoir piqué, fourmis qui forment un « pont »pour permettre au groupe de traverser un ruisseau...). A l'inverse, certains agents ne s'impliquent dans la résolution d'un but de l'équipe que si cela ne les empêche pas d'atteindre leur objectif personnel. Ce cas de figure complique évidemment la gestion des tâches dans l'équipe car on n'a jamais la garantie qu'une tâche confiée à un agent sera exécutée par celui-ci.

La communication au sein de l'équipe

Au sein d'une équipe, les agents sont souvent amenés à communiquer, que ce soit entre eux ou avec leur chef dans le cas échéant. Ils peuvent le faire de différentes manières. La communication par signaux est répandue dans le règne animal, en particulier via les phéromones (stigmergie chez les fourmis par exemple). Elle peut aussi se faire grâce à un langage évolué dans lequel l'information est plus dense et qui demandera une plus grande capacité de raisonnement, que ce soit pour l'émetteur ou le récepteur du message. Enfin, il est possible de communiquer via un « tableau noir »sur lequel les agents viennent écrire et consulter les messages à l'intention de tous les autres agents. L'utilisation d'une mémoire partagée est très similaire à un tableau noir. La communication permet d'organiser les tâches dans l'équipe de manière efficace en s'assurant que deux agents ne vont pas entreprendre une même tâche si ce n'est pas nécessaire ou effectuer des tâches antinomiques. La communication apporte son lot de contraintes, qu'il s'agisse d'utiliser certains moyens de communication ou de se trouver à portée de réception du destinataire du message dans le cas d'un message sonore : la nécessité de communiquer peut entraîner des déplacements, et donc un plan potentiellement complexe !

2.3.5 L'inférence de comportement

Ces différents modes d'interaction entre les agents peuvent les amener à faire de l'inférence de comportement afin de « raisonner sur le raisonnement de l'autre » : dans le cas de la collaboration, il s'agit d'anticiper les besoins de l'adversaire afin de l'aider au mieux, ou tout au moins de ne pas le gêner, tandis que dans le cas de la concurrence et a fortiori de la compétition, il s'agit d'anticiper ses actions afin de pouvoir contrecarrer ses actions. C'est le cas par exemple du bot de John Laird pour le jeu Quake présenté dans l'article *It Knows What You're Going To Do : Adding Anticipation to a Quakebot* [Lai01]. Le bot observe son ennemi

dans une situation donnée (figure 2.5 image (1)). Dans cet exemple, l'ennemi se dirige vers une salle sans issue contenant une ressource. Le bot se projette dans la situation afin de deviner ce que son ennemi est en train de faire (figure 2.5 image (2)), en l'occurrence, il a l'intention de prendre la ressource. Le bot prévoit alors que son ennemi va ensuite ressortir de la salle (figure 2.5 image (3)). Il se positionne alors dans un angle mort sur le chemin du retour de son ennemi afin de lui tendre une embuscade (figure 2.5 image (4)). L'inférence de comportement est aussi à la base de l'élaboration des stratégies dans le dilemme itéré du prisonnier [BDM98] : à partir de l'historique des coups précédents, l'agent essaie de déterminer le comportement de l'adversaire pour déduire les futurs coups qu'il va jouer. Il peut ainsi adapter sa stratégie pour se garantir des coups qui maximiseront ses propres gains.

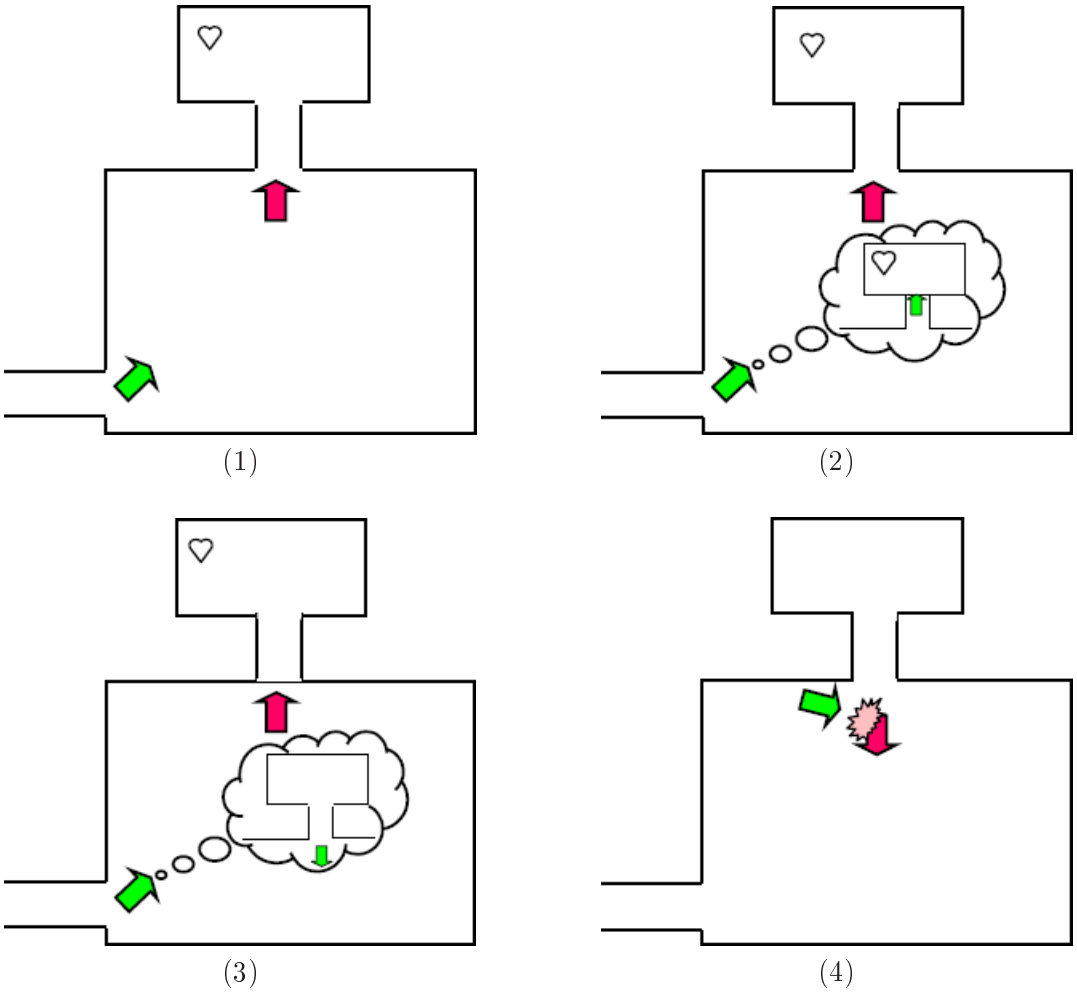


FIG. 2.5 – Exemple d'inférence de comportement tiré de [Lai01]

2.4 Les organisations

La réunion de plusieurs agents dans l'intention de résoudre un but commun requiert une organisation de la part des agents. Cette organisation conditionne la manière dont les agents interagissent entre eux : en fonction de l'organisation, les relations d'autorité, le flux d'informations, les allocations de tâches ou de ressources ne seront pas les mêmes. Ces différentes caractéristiques de la structure organisationnelle des agents influent sur la forme, la taille et le comportement global du système. Il a été démontré par ailleurs que l'organisation avait des répercussions importantes sur les performances du système [HML04] et qu'il n'existait pas d'organisation qui soit la mieux adaptée quelle que soit la situation rencontrée. Aussi, en fonction des caractéristiques des agents, des buts et de l'environnement, il est nécessaire de choisir la structure organisationnelle la mieux adaptée parmi les hiérarchies, les holarchies, les coalitions, les équipes, les congrégations, les sociétés, les fédérations, les marchés ou les organisations matricielles que nous allons définir maintenant. Notons que ces définitions ne sont pas mutuellement exclusives. De plus amples descriptions de ces structures pourront être trouvées dans [HL05] dont ces définitions sont inspirées.

Les hiérarchies. L'organisation des agents est une structure d'arbre : les agents se trouvant en haut de l'arbre ont une vision plus globale que ceux qui sont en dessous. Les communications entre les agents ne se produisent qu'entre deux entités connectées, c'est-à-dire entre un père et un fils dans l'arbre. Les messages entre deux agents non connectés directement empruntent obligatoirement la voie hiérarchique : les messages ne sont transmis par les supérieurs que si cela est jugé nécessaire.

Les holarchies. Le terme de Holon revient à Arthur Koestler. Il s'agit d'une organisation fractale constituée de hiérarchies imbriquées au sein desquelles chaque entité du système est à la fois un tout autonome et une partie subordonnée d'un système plus vaste. La holarchie est un holon au même titre que chacune de ses parties.

Les coalitions. Cette organisation est un regroupement temporaire d'agents créé dans un but précis dont la résolution entraînera la dissolution de la structure. La structure interne d'une coalition est généralement plate, bien qu'un agent particulier puisse éventuellement se distinguer des autres par son rôle de représentant ou d'intermédiaire pour le groupe, qui est vu de l'extérieur comme une seule entité.

Les équipes. Les agents constituant une équipe ont consenti à travailler ensemble en vue de réaliser un but commun : leurs actions ont pour but d'aider à la satisfaction des buts de l'équipe. L'organisation au sein de l'équipe est libre, toutefois, en général les agents jouent un ou plusieurs des rôles requis pour exécuter les sous-tâches nécessaires pour atteindre le but de l'équipe. Ces rôles peuvent évoluer dans le temps pour s'adapter à la situation, bien que le but de l'équipe ne varie pas.

Les congrégations. A l'instar des équipes et des coalitions, les congrégations sont un regroupement d'agents dont la structure organisationnelle est plate. La différence se situe principalement au niveau de la durée de vie de la structure qui n'est pas limitée à un seul but. Les agents d'une congrégation n'ont pas obligatoirement un but fixé, mais ont un ensemble de capacités et d'exigences à satisfaire, ce qui justifie le fait de constituer une congrégation. La constitution d'une congrégation n'est pas stable et peut évoluer dans le temps.

Les sociétés. Les sociétés sont des structures ouvertes dans lesquelles les agents peuvent entrer et sortir à souhait. Les agents ont différents buts, ont des capacités éventuellement hétérogènes et peuvent agir et communiquer dans le cadre que leur confère la société. Celle-ci est persistante malgré l'arrivée et le départ des agents. Au sein d'une société, des sous-organisations peuvent regrouper certains agents, ou ils peuvent être complètement indépendants. L'entrée dans la société implique le respect d'un certain nombre de conventions ou normes sociales afin de facilité la coexistence.

Les fédérations. Une fédération est un regroupement d'agents qui ont cédé une partie de leur autonomie à un membre qui est chargé de les représenter. Les membres de la fédération ne communiquent qu'avec le délégué, qui agit comme intermédiaire avec l'extérieur de la fédération grâce aux connaissances qu'il possède sur les membres du groupe.

Les marchés. Deux principaux types d'agents interviennent dans les marchés. Les agents acheteurs, d'une part, qui placent des enchères en vue d'acquérir les articles proposés par les agents vendeurs, d'autre part. Ces derniers déterminent le gagnant d'une vente en fonction des enchères. Cette organisation est proche des fédérations dans la mesure où un individu (le vendeur) est responsable de la coordination de quelques autres agents (les acheteurs), mais elle en diffère par l'aspect compétitif qui règne entre les agents, contrairement à une fédération. Il est à noter qu'un marché peut être utilisé pour des opérations non financières comme la négociation de temps processeur sur un système.

Les organisations matricielles. Ces organisations permettent de représenter des systèmes hiérarchiques non stricts, dans la mesure où ils s'affranchissent de la règle « un agent, un chef » que l'on trouve dans les systèmes hiérarchiques présentés précédemment. Cette organisation se rapproche plus fidèlement des organisations humaines dans lesquelles un individu reçoit des ordres ou tout au moins des pressions de la part de plusieurs autres individus. Le terme organisation matricielle vient du fait qu'on peut représenter cette structure par une matrice dans laquelle les supérieurs sont en ligne, les subordonnés en colonne, et à l'intérieur de laquelle on place une croix pour désigner une relation d'autorité.

Dans la partie 4.4 de la contribution, les agents agents coopèrent au sein d'un groupe. Ce groupe est une *équipe* structurée *fédération* : un groupe d'agents constitué lors de la conception de la simulation coopère en vue d'atteindre un but commun. L'un des agents, grâce à ses connaissances sur les agents du groupe, joue le rôle de chef : il assure une fonction d'intermédiaire en recevant le but de l'équipe et en répartissant les tâches entre les agents.

2.5 Les formes de coopération

Les agents d'un système multi-agent ont un comportement qui tend à satisfaire leur(s) but(s), que celui-ci soit explicite ou implicite. Lorsqu'un but est commun à plusieurs agents ou lorsqu'un agent réalise le but d'un autre agent, on parle de *coopération*. La coopération est l'un des concepts clefs des systèmes multi-agent, qui les différencie d'autres paradigmes comme la programmation objet ou le calcul distribué. On peut trouver la coopération sous différentes formes en fonction du contexte, d'après [DFJN97] dont est issue la figure 2.6 qui donne une typologie des différentes formes de coopération.

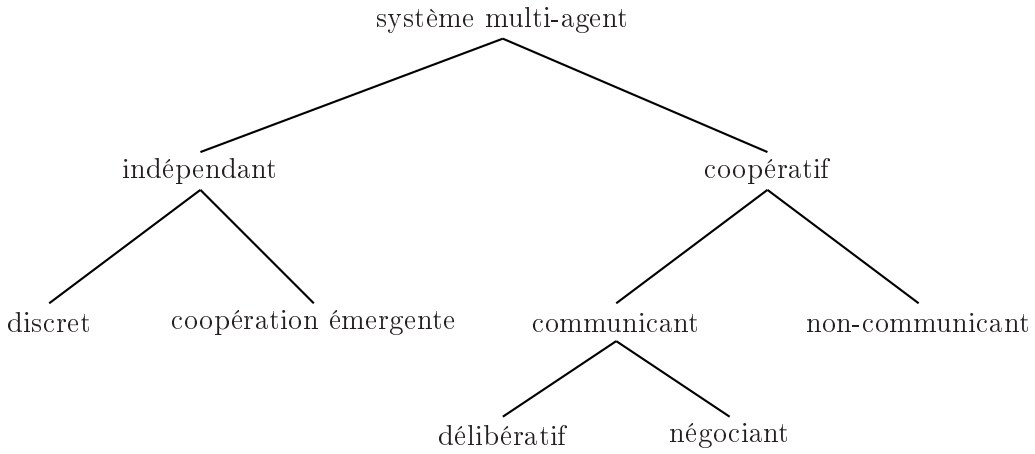


FIG. 2.6 – Les différentes formes de coopération dans les systèmes multi-agent (figure issue de [DFJN97])

L'indépendance des agents exprime le fait que les agents poursuivent leur propre cheminement indépendamment des autres agents. On dit que les agents indépendants sont discrets lorsque leurs comportements n'ont rien de commun. Dans le cas contraire les agents sont en situation de coopération dans la mesure où ils peuvent coopérer sans que cela soit intentionnel. Lorsque les agents ne sont pas indépendants, certaines de leurs actions sont coopératives, c'est-à-dire intentionnellement destinées à aider un autre agent dans la résolution de ses buts. Cette coopération peut se faire sans communication explicite, en réagissant au comportement d'un congénère, ce qui est une forme de communication par l'environnement. La communication explicite se fait par échange de messages. Dans un système délibératif, l'échange de message permet la construction concertée du plan de manière à coopérer avec les autres agents, par le biais ou non de la coordination. Cette construction concertée du plan existe dans les systèmes négociants, mais un degré de compétition entre les agents limite la coopération lorsqu'elle met en péril le but individuel d'un agent.

Nous considèrerons dans les aspects *équipe* de ce travail un système multi-agent coopératif délibératif : le but commun des agents (même s'il n'est pas connu d'eux) les place dans une situation coopérative explicite, et leurs actions sont entièrement tournées vers la résolution du but commun, quels que soient leurs buts individuels.

2.6 La planification

2.6.1 Introduction à la planification

L'une des formidables capacités du cerveau humain est d'être capable de résoudre un problème en prévoyant un ensemble de tâches dont l'exécution vise à se rapprocher du but souhaité, puis à l'atteindre : c'est ce que l'on appelle la planification. Ses applications sont nombreuses et variées : opérations militaires, exploration spatiale autonome, tâches mécaniques pour la construction... De nombreux travaux ont été effectués pour doter la machine de

cette capacité, afin de pouvoir remplacer l'Homme ou de permettre de résoudre rapidement des problèmes trop compliqués pour un cerveau humain.

Un problème de planification se représente par
- **un ensemble d'états** $\mathcal{E}$ représentant tous les états possibles du monde
- **un état initial** $i \in \mathcal{E}$ décrit dans un langage formel
- **un état final** $f \in \mathcal{E}$ décrit dans un langage formel, qui représente le but à atteindre
- **un ensemble d'opérateurs** O. Un opérateur $o()$ est une fonction de $\mathcal{E} \rightarrow \mathcal{E}$ qui décrit formellement une action permettant de passer d'un état $e_i \in \mathcal{E}$ à un état $e_j \in \mathcal{E}$

L'exemple de la figure 2.7 illustre la problématique de la planification. Dans ce jeu de domino, on doit placer trois dominos sur les emplacements grisés en respectant les règles du jeu de domino : lorsque deux dominos sont juxtaposés, les chiffres qui sont mis en correspondance du côté où les dominos se touchent doivent être identiques. Nous pouvons considérer qu'il s'agit d'un problème de planification dans lequel l'ensemble de états possibles $\mathcal{E}$ est $\{1, 2, 3, 4, 5, 6\}$, l'état initial i est 5 et l'état final f est 4. Les dominos disponibles sont les opérateurs : nous avons donc l'opérateur $5 \rightarrow 1$, $5 \rightarrow 6$, $3 \rightarrow 4$, $6 \rightarrow 3$ et $1 \rightarrow 2$. Nous ne considérons pas ici les opérateurs correspondant aux dominos "retournés" (par exemple l'opérateur $1 \rightarrow 5$ pour le domino a). Ici, chaque opérateur ne peut être utilisé qu'une seule fois puisqu'on le « consomme » lorsqu'on l'utilise.

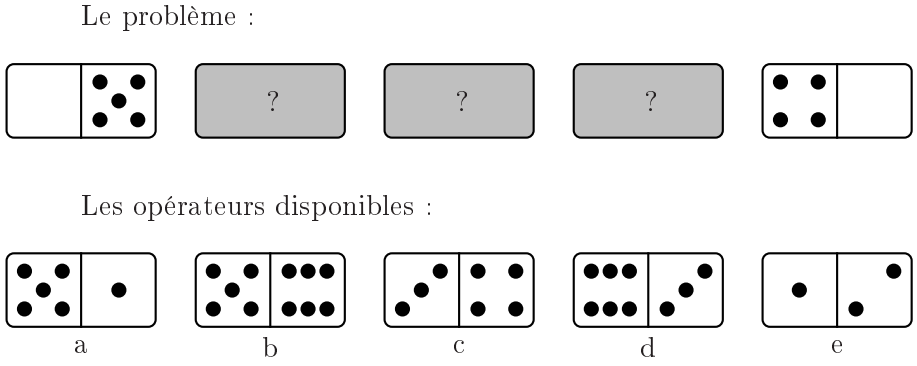


FIG. 2.7 – Exemple de problème de planification : le jeu de domino

Pour résoudre un problème de planification comme celui-ci, on peut utiliser des techniques de chaînage[4]. Le but du chaînage est de trouver un opérateur qui permet de passer d'un état à un autre. Lorsque l'on part de l'état initial pour atteindre l'état final, on parle de *chaînage avant*, lorsque l'on part de l'état final et que l'on tente de remonter vers l'état initial, on parle de *chaînage arrière*. Plus généralement, on regroupe ces deux techniques sous le terme de *moteurs d'inférence*. Un moteur d'inférence utilise une base de faits et de règles. La base de faits exprime un ensemble de réalités décrivant les observations de départ (exemple : il pleut), tandis que les règles expriment ce que l'on peut déduire à partir de certains faits (exemple : s'il pleut et qu'il y a du soleil, alors il y a un arc-en-ciel).

[4]Ce genre de problème est généralement résolu avec un algorithme de type Dijkstra, mais c'est le chaînage qui nous intéresse ici.

Le chaînage avant

A partir de la base de faits, le moteur de chaînage avant déduit tous les nouveaux faits déductibles grâce à l'application des règles. Le moteur applique non seulement les règles sur les faits de départ mais aussi sur les faits qu'il vient lui-même de déduire. Une règle se présente généralement sous la forme d'une expression du type *SI conditions ALORS conséquences*. Pour appliquer une règle, il faut que les faits requis par la partie conditions de la règle soient vrais. Lorsque l'on utilise un moteur de chaînage avant pour établir une preuve, de nombreux nouveaux faits sont produits jusqu'à ce que le fait attendu soit prouvé. Il est alors possible d'étudier la succession de règles qui permet d'aboutir à la solution : c'est ce que l'on appelle le *chemin de preuve*.

On peut aussi utiliser un moteur de chaînage avant pour faire de la planification : les faits expriment l'état du monde (exemple : la lumière est éteinte), et les règles expriment ce qu'il est possible de faire dans le monde (exemple : si j'appuie sur le bouton, la lumière s'allume) : ce sont ici des opérateurs. Le moteur de chaînage avant déduit donc tous les états du monde qu'il est possible d'atteindre en utilisant des combinaisons des opérateurs sur le monde tel qu'il est décrit dans l'état de départ. Si le moteur trouve une « preuve » (liste ordonnée d'opérateurs) qui permet de prouver l'état final à partir de l'état initial, alors cette preuve est un *plan* : à partir de l'état initial, si l'on applique la suite d'opérateurs décrits par le chemin de preuve, on atteindra l'état final. Le chaînage avant produit beaucoup de plans qui mènent à des états qui n'intéressent pas l'utilisateur.

Considérons l'exemple du jeu de dominos présenté en figure 2.7. Nous n'avons qu'un seul fait, qui est le nombre présent sur le domino à droite duquel on va poser le nôtre : *etat=5*. Les opérateurs sont les suivants :

a SI *etat=5* ALORS *etat=1*

b SI *etat=5* ALORS *etat=6*

c SI *etat=3* ALORS *etat=4*

d SI *etat=6* ALORS *etat=3*

e SI *etat=1* ALORS *etat=2*

L'état final à atteindre est *etat=4*. Pour pouvoir utiliser une règle (i.e. poser un domino), il faut que les faits présents dans la partie conditions de la règle soient vrais, en l'occurrence que *etat=5*. Seules deux règles répondent à cette exigence : les règles a et b. Il est nécessaire de faire un choix entre ces deux règles. Ce choix peut être arbitraire, ou parfois être guidé par des heuristiques. On décide d'appliquer la règle a, ce qui amène le monde dans l'état *etat=1*. Nous pouvons alors utiliser la règle e, qui nous amène dans l'état *etat=2*. Nous sommes alors bloqués puisqu'aucun des opérateurs n'est plus applicable (figure 2.8). Il faut alors remettre en cause le dernier choix que l'on a fait, s'il existe : c'est ce qu'on appelle le *backtracking*. On remet donc en cause le choix de l'opérateur a au départ, et on utilise le b. On peut alors appliquer l'opérateur d, puis l'opérateur c qui nous amène dans l'état souhaité *etat=4*. Notons que le backtracking effectué est indépendant du chaînage. Le chaînage est généralement réalisé sans but, en vue de déduire toutes les nouvelles règles qu'il est possible de déduire. L'utilisation du backtracking n'est là que pour guider la recherche de la solution lorsque le chaînage est utilisé avec un but.

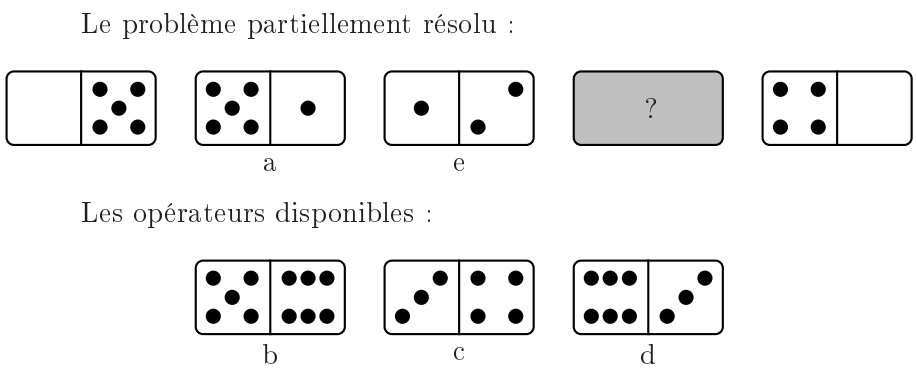


FIG. 2.8 – Chaînage avant pour le jeu de domino

Le chaînage arrière

A l'inverse du chaînage avant, le chaînage arrière permet de déduire les causes d'un fait (par exemple, s'il y a un arc-en-ciel, on déduit qu'il pleut et qu'il y a du soleil simultanément). A partir d'un fait, on utilise donc les règles « à l'envers » afin de trouver quelles ont pu être les causes de ce fait. On choisit donc une règle dont la partie *conséquence* est le fait considéré, ou un fait que l'on vient de déduire comme étant une cause possible.

De même, lorsque l'on utilise un chaînage arrière pour résoudre un problème de planification, on part de l'état final souhaité dans le monde, et l'on cherche un opérateur qui peut être appliqué pour atteindre cet état. Si les conditions de cet opérateur ne sont pas vérifiées, on cherche un autre opérateur qui permet de les obtenir à leur tour. On procède ainsi de suite jusqu'à ce qu'on arrive à des faits établis dans l'état de départ. La succession d'opérateurs trouvée constitue un plan pour atteindre l'état final considéré.

Dans l'exemple de notre jeu de domino (figure 2.7), on part de l'état final (*etat=4*) et on cherche un opérateur qui permet d'obtenir *etat=4* : l'opérateur c. On continue à appliquer ce procédé, en effectuant d'éventuelles remises en causes de choix (comme dans le cas du chaînage avant) et l'on trouve le même plan que celui trouvé par le chaînage avant : b, d, c.

Le cas du jeu de domino que nous venons d'étudier est très simple. Plus généralement, les opérateurs contiennent plusieurs conditions : *SI condition1 ET condition2 ALORS conséquence*. Lors du chaînage arrière, plusieurs conditions doivent donc être résolues, ce qui entraîne plusieurs chaînages en parallèle. Au lieu d'une liste d'opérateurs, le plan obtenu prend alors la forme d'un arbre ET/OU, dont la racine est le but à atteindre et les feuilles sont des faits vrais dans l'état de départ. Pour exécuter un plan en forme de liste, on applique séquentiellement les opérateurs qui le composent. De même pour un plan arborescent, on applique les opérateurs en respectant la précédence de l'arbre : un opérateur ne peut être appliqué que si tous les opérateurs constituant le sous-arbre ont été appliqués.

2.6.2 Le Graphplan

L'algorithme Graphplan [BF95] est un algorithme de planification résolvant les problèmes exprimables en STRIPS [FN71]. Cet algorithme alterne deux phases : la première consiste à

construire un graphe sur lequel le planificateur proprement dit vient s'appuyer dans la seconde phase pour établir son raisonnement. Le principal atout de Graphplan (et de ses nombreux dérivés) est la détection des conflits qui peuvent survenir lors de l'utilisation de deux opérateurs aux conséquences contraires.

Le langage STRIPS

Le langage STRIPS [FN71] est issu d'un planificateur basique du même nom fonctionnant sur le principe d'une pile de buts. Nous ne rentrerons pas dans les détails quant au fonctionnement de ce planificateur mais allons étudier le formalisme sur lequel il s'appuie, puisque c'est celui-là même qui est utilisé par l'algorithme Graphplan.

Un état du monde est représenté par un terme dans un langage du premier ordre. Les propriétés du monde qui changent sont représentées par des prédicats, que l'on appelle fluents dans l'algorithme Graphplan. L'algorithme fonctionne avec des prédicats d'ordre 0.

Un problème STRIPS est composé de :

- un état initial : c'est un ensemble de prédicats
- un but : c'est une conjonction de prédicats qui doivent être vrais à la fin de l'algorithme
- les opérateurs

Les opérateurs STRIPS sont composées de deux parties :

- les préconditions : elles sont représentées par une conjonction de prédicats. Chacun des prédicats doit être vrai pour pouvoir exécuter l'action.
- les modifications : elles sont constituées d'une liste d'ajouts et d'une liste de retraits de prédicats.

Pour comprendre la construction d'un opérateur STRIPS, prenons l'exemple d'une action connue de tous : prendre un café au distributeur. Cette action comporte des conditions : tout d'abord, il est nécessaire de se trouver à proximité du distributeur : on ne peut pas utiliser celui-ci à distance. La seconde condition est d'avoir un jeton[5]. Ces conditions sont les prérequis nécessaires pour pouvoir effectuer notre action *prendrecafé*. Evidemment, cette action a aussi des conséquences, sinon son exécution ne servirait à rien : nous avons maintenant un café, mais nous n'avons plus de jeton. Pour résumer, voici ce que fait l'action *prendrecafé* :

- `préconditions : {jeton, distributeur proche}`
- `ajouts : {café}`
- `retraits : {jeton}`

Et voici l'expression de cette action sous la forme d'un opérateur STRIPS :

```
(operator
        PRENDRECAFE
        (preconds (jeton) (distributeur proche))
        (effects (cafe) (del jeton)))
```

[5]Nous ne prendrons pas en considération d'autres conditions comme le fait que la machine ait été remplie et soit correctement branchée, ce qui n'est pas de notre ressort.

Ce formalisme est utilisé par l'algorithme Graphplan pour la phase d'expansion du graphe, qui alterne avec la phase de recherche de solutions dans ce graphe. Nous allons maintenant étudier ces deux phases.

La phase d'expansion

Dans cette première phase, l'algorithme construit une structure que l'on appelle *graphe de planification*. Cette structure est plus petite que le graphe d'états complet et permet une recherche rapide du plan-solution. Le graphe de planification se construit par couches, en alternant les couches d'état et les couches d'action. Pour commencer, on met à la couche 0 tous les fluents qui sont présents dans l'état initial (cf figure 2.9). La couche d'action n est constituée de toutes les actions dont les prérequis se trouvent à la couche $n-1$. On relie les actions à leur préconditions. Sur une couche d'action, on rajoute des actions particulières, appelées « *no-op* » qui ne font rien. Elles servent à assurer la persistance d'un fluent d'une couche d'état n à la couche d'état $n+2$. Il est donc nécessaire d'ajouter un *no-op* par fluent. Pour construire une couche d'état n, on ajoute tous les fluents de la couche d'état $n-2$ (ils sont maintenus par les *no-op*), ainsi que ceux qui sont ajoutés par les actions de la couche d'action $n-1$, en les reliant à l'action correspondante.

Les couches du graphe obtenu correspondent à des tranches de temps : une couche paire représente un instant où le monde est dans l'état représenté par l'ensemble des fluents présents, et une couche impaire représente un intervalle de temps durant lequel on peut exécuter simultanément les actions présentes à cette couche. Toutefois il n'est pas forcément possible d'exécuter simultanément deux actions qui se trouvent sur la même couche : certaines actions sont incompatibles entre elles. On dit qu'elle sont mutuellement exclusives, d'où la notion de *mutex* que nous allons aborder. Il existe plusieurs cas d'actions incompatibles :

1. **besoins inconsistants** : une action retire la précondition d'une autre action
2. **effets inconsistants** : une action retire un fluent ajouté par une autre action
3. **besoins concurrents** : une action a un besoin en concurrence avec une autre action (nœuds d'état mutuellement exclusifs)

Les mutex sont propagés à la couche suivante dans le graphe. On propage un mutex d'une couche d'action à une couche d'état de la manière suivante : un nœud représentant un fluent est *mutex* avec un autre si toutes les actions (*no-op* incluses) permettant d'obtenir le premier fluent sont *mutex* avec toutes les actions permettant d'obtenir le second.

Un *mutex* entre deux actions se représente graphiquement par un arc de cercle entre ces deux actions. Un mutex ne peut relier que deux nœuds du même niveau dans le graphe. Sur la figure 2.9 on peut par exemple observer un mutex de type 1 entre *vider* et *cuisiner*. Les fluents ajoutées (ou maintenus) par un opérateur sont représentés par un trait plein tandis que ceux qui sont enlevés sont représentés par un trait pointillé.

La phase de recherche

Cette phase a pour but de rechercher une solution dans le graphe de planification construit par la phase d'expansion. Une fois que l'algorithme a construit un nouveau niveau pair, il

recherche une solution dans ce graphe. S'il ne trouve pas de solution, le graphe est étendu jusqu'au niveau pair suivant et une nouvelle recherche est lancée sur le nouveau graphe.

Pour qu'une solution existe dans le graphe, il est impératif que tous les fluents composant le but souhaité soient présents au dernier niveau dans le graphe de planification. Sans cette condition, il est inutile de lancer la recherche d'un plan solution : il faut étendre à nouveau le graphe de planification. Si tous les fluents sont présents au dernier niveau du graphe de planification, cela ne signifie pas nécessairement qu'il y a une solution, à cause des *mutex* : dans ce cas, il est aussi nécessaire d'étendre le graphe de planification.

La méthode de recherche d'une solution dans le graphe est un chaînage arrière sur le graphe de planification. On choisit l'un des fluents composant le but au dernier niveau n, et on cherche au niveau $n-1$ une action permettant d'obtenir ce fluent, grâce aux arcs reliant les niveaux. On procède de même pour les autres fluents composant le but, en évitant les actions qui sont *mutex* avec une action déjà choisie. Un mécanisme de backtracking permet de tester toutes les possibilités afin de trouver une solution si elle existe. Dans le cas contraire, la fonction de recherche du plan-solution renvoie un échec : il n'y a pas de solution.

Une fois qu'on a trouvé un ensemble d'actions au niveau $n-1$ pour résoudre les fluents du niveau n, on fait de même pour les fluents du niveau $n-2$ qui sont les préconditions des actions choisies au niveau $n-1$. Si la fonction de recherche renvoie un échec pour la résolution du niveau $n-2$, une autre solution est cherchée au niveau n puis la recherche reprend au niveau $n-2$. On procède ainsi jusqu'au niveau 0. Si tous les fluents qui sont préconditions des actions choisies au niveau 1 sont présents au niveau 0, un plan-solution est trouvé. Dans le cas contraire, la fonction retourne un échec qui se propage de niveau en niveau afin de chercher une autre solution, si elle existe.

L'alternance des phases d'expansion du graphe et de recherche d'un plan solution pourrait continuer indéfiniment lorsque le plan solution n'existe pas. L'algorithme le détecte afin de s'arrêter et de prévenir l'utilisateur qu'il n'existe pas de plan. Lors de la phase d'expansion du graphe, si l'algorithme construit un niveau d'action n et un niveau d'état $n+1$ respectivement identiques aux niveaux $n-2$ et $n-1$, il est évident que les niveaux suivants seront à leur tour identiques. Si tous les fluents composant le but ne sont pas présents au dernier niveau d'état, ils ne le seront jamais si l'on continue à étendre le graphe. On peut donc arrêter la recherche. Si les fluents sont présents mais qu'une solution n'est pas trouvée, il est possible qu'une extension du graphe permette de trouver une solution. Un test plus complexe que nous ne détaillerons pas ici permet à l'algorithme de s'arrêter.

Les dérivés de Graphplan

Quelques algorithmes sont directement dérivés du Graphplan. On peut citer entre autres :

- **IPP (Interference Progression Planner)** [KNHD97] [KNH97] qui est une implémentation optimisée de Graphplan utilisant le formalisme ADL (Action Description Language) [Ped89]. Cet algorithme propose plusieurs améliorations : possibilité d'utiliser des effets conditionnels, expressivité plus grande grâce à l'utilisation du quantificateur universel, possibilité de raisonner sur la négation d'un fluent...
- **SGP (Sensory Graphplan)** [WAS98] qui est une implémentation en LISP de Graphplan utilisant le formalisme PDDL [McD98]. Cet algorithme permet la gestion du

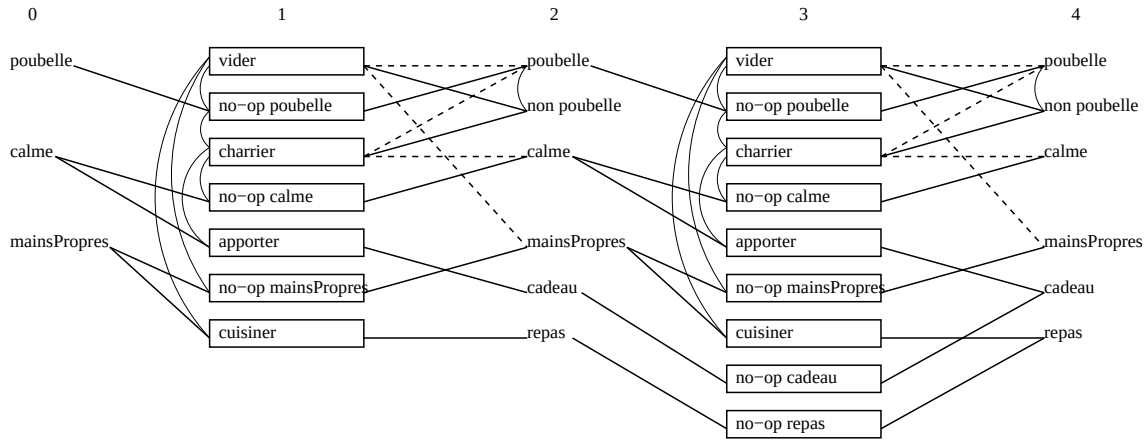


FIG. 2.9 – Exemple de graphe obtenu grâce à l'algorithme Graphplan sur le « Dinner Date Problem », extrait de [Wel99]

quantificateur universel, l'utilisation des effets conditionnels. Il utilise en outre des « actions sensorielles[6] » qui permettent à l'agent de disposer d'informations supplémentaires pour la planification. SGP remplace le vieillissant UCPOP (Partial Order Planning) [PW92] : il en étend les fonctionnalités et se montre beaucoup plus rapide.

- **BlackBox** [KS99] qui combine l'algorithme Graphplan et l'algorithme SATPlan [KS92] : une fois le graphe de planification créé par la première phase de l'algorithme Graphplan, BlackBox le transforme en problème de satisfaction de contraintes (CSP), le résout grâce à un algorithme SAT et déduit le plan de la solution CSP obtenue.
- **PetriPlan** [SCK00] qui utilise la première phase de l'algorithme Graphplan pour réduire la complexité du graphe de planification et le transforme en réseau de Petri.

2.6.3 SHOP

Les planificateurs HTN (Hierarchical Task Network) diffèrent des planificateurs classiques par leur manière de calculer les plans par décomposition successive des tâches en un ensemble de tâches plus élémentaires. Le planificateur SHOP[NCLMA99] et son successeur SHOP2[NAI+03] font partie de ces planificateurs. Le planificateur SHOP2 est basé sur des opérateurs et des méthodes. Les opérateurs décrivent les conditions nécessaires à l'accomplissement d'une tâche primitive ainsi que ses effets, tandis que les méthodes décrivent des décompositions possibles de tâches en un ensemble de tâches moins abstraites. On trouvera le détail de l'algorithme dans [SPW+04].

Les opérateurs

Un opérateur SHOP2 est une expression de la forme $h(\vec{v})$ *Pre Del Add* dans laquelle :

- $h(\vec{v})$ est une tâche primitive comportant une liste de paramètres $\vec{v}$
- *Pre* décrit les préconditions nécessaires à l'exécution de l'opérateur

[6] traduction approximative de *sensory actions*

– *Del* décrit l'ensemble des faits qui deviennent faux lors de l'exécution de l'opérateur
– *Add* décrit l'ensemble des faits qui deviennent vrais lors de l'exécution de l'opérateur

Les préconditions contiennent des atomes logiques portant sur les paramètres de h. Les atomes peuvent être combinés à l'aide de conjonctions, disjonctions, négations, implications ou des quantificateurs universels. Quant aux listes *Add* et *Del*, elles sont généralement définies par une conjonction d'atomes logiques, mais il est possible d'utiliser des expressions conditionnelles et des quantificateurs universels dans certains cas.

Les méthodes

Une méthode SHOP2 est une expression de la forme $h(\vec{v})\ Pre_1\ T_1\ Pre_2\ T_2 \ldots\ Pre_n\ T_n$ dans laquelle :

– $h(\vec{v})$ est une tâche composée comportant une liste de paramètres $\vec{v}$
– chaque Pre_i est une expression représentant une précondition
– chaque T_i est un ensemble partiellement ordonné de tâches

Les méthodes expriment les différentes décompositions possibles d'une tâche composée. Elles sont analogues à des expressions conditionnelles : si Pre_1 est vraie, alors la décomposition T_i est utilisée, sinon si Pre_2 est vraie alors c'est la décomposition T_2 qui est utilisée et ainsi de suite. Une liste de tâches peut être ordonnée ou non, et elle est composée de tâches et d'autres listes de tâches. De cette manière, les imbrications de listes ordonnées et non ordonnées permettent de décrire des ordres partiels complexes. Chaque tâche de la liste peut être une tâche primitive ou une tâche composée, elle sera résolue en utilisant respectivement un opérateur ou une méthode. Comme plusieurs méthodes peuvent contenir pour une tâche donnée, SHOP2 essaie toutes les décompositions possibles à l'aide d'un mécanisme de backtrack.

La description d'un problème

Une problème de planification de SHOP2 est un triplet (S, T, D) dans lequel S est un état initial, T est une liste de tâches (le but) et D est une description du domaine (l'ensemble des opérateurs et des méthodes). L'algorithme SHOP2 prend en entrée un triplet (S, T, D) et retourne un plan $P = (p_1 p_2 \ldots p_n)$ qui est une séquence d'opérateurs instanciés qui résout T à partir de l'état S dans le domaine D.

La planification

Pour toutes les tâches t de T, si t est une tâche primitive, l'algorithme cherche un opérateur dans $o = (hPreDelAdd)\ D$ tel que h s'unifie avec t et l'état actuel s satisfasse Pre. Il se rappelle ensuite pour résoudre (s', T', D) avec s' issu de s après application des listes *Del* et *Add* et $T' = T \setminus \{t\}$. Si t est une tâche composée, alors l'algorithme cherche une méthode $m = (h\ Pre_1\ T_1\ Pre_2\ T_2 \ldots\ Pre_n\ T_n)$ dans D telle que h s'unifie avec t. L'algorithme choisit l'ensemble de tâches T_i tel que s satisfasse Pre_i. L'algorithme se rappelle lui-même pour résoudre (s', T', D) avec s' issu de s après application des listes *Del* et *Add* et $T' = T_i \cup T \setminus \{t\}$.

Pourquoi n'avons-nous pas utilisé ces planificateurs ?

Graphplan

Le Graphplan propose une solution de planification efficace prenant en compte les éventuelles incompatibilités entre les actions (mutex). Toutefois il n'est pas envisageable d'utiliser ce planificateur dans ce travail, car celui-ci est basé sur une logique d'ordre 0 : même si certains opérateurs possèdent des variables (laissant ainsi croire que l'on travaille en ordre 1), le calcul du plan se fait avec une logique d'ordre 0 sur les opérateurs complètement instanciés (cf "A Simple Rocket Domain" dans [BF95]).

Dans des simulations géographiquement situées comme les nôtres se pose alors le problème de l'opérateur de déplacement :

```
(operator
        MOVE
        (params (?from) (?to))
        (preconds (at ?from))
        (effects (at ?to) (del at ?from)))
```

Afin d'être utilisé dans le graphe de planification, cet opérateur doit être instancié pour toutes les valeurs des paramètres *?from* et *?to*. Dans ce cas, le nombre d'opérateurs est fonction du carré du nombre de places composant l'environnement, et le graphe de planification est au moins de la longueur du chemin planifié. De plus, il n'est pas possible d'ajouter des conditions au franchissement de certaines places, afin de représenter les obstacles présents dans l'environnement, ce qui est l'un des points cruciaux de ce travail.

SHOP

SHOP est un planificateur très performant, qui a gagné l'un des quatre prix parmi quatorze planificateurs lors de la Compétition Internationale de Planification en 2002 [LF03]. Malgré ses bonnes performances, nous ne l'avons pas utilisé. Dans un premier temps nos recherches se sont essentiellement focalisées sur le modèle centré interaction (cf 3.1). Nous avons tout d'abord utilisé un moteur d'inférence simple pour faire nos premières expérimentations car la planification n'est pas l'objectif central de ce travail. Le planificateur s'est ensuite vu greffer un module de mise à jour permettant de faire une replanification partielle en fonction des modifications de l'environnement, ce que ne propose pas SHOP. Notre modèle permet de donner aux agents des buts sous forme d'action ou sous forme de prémisses, alors que les buts de SHOP sont obligatoirement des tâches. Le déroulement de la simulation n'est pas le même dans les deux cas. Ceci étant, notre modèle n'est pas attaché au moteur d'inférence que nous utilisons et il est possible d'en utiliser un autre.

2.6.4 Planification et replanification d'équipe

La résolution de certains problèmes n'est rendue possible que par l'utilisation de plusieurs agents. C'est le cas lorsque le problème fait appel à des capacités qui sont réparties entre les agents ou lorsque le problème impose la présence physique simultanée à des endroits différents,

comme la recherche d'un individu dans un labyrinthe [PF05] qui nécessite de contrôler simultanément plusieurs issues. Ce problème particulier qui impose la coordination spatio-temporelle des agents est résolu grâce à un modèle dialectique qui permet de construire le plan d'équipe.

Une autre approche de la planification d'équipe consiste à utiliser des réseaux de Petri. Ils permettent une planification par raffinement de plan (qui est assez proche de la décomposition de tâches proposée par SHOP) et se révèlent être par ailleurs un excellent outil pour la synchronisation [BCM96][EH96]. Bonnet-Torrès et al. proposent une approche basée sur les réseaux de Petri colorés afin de concevoir un plan et de l'exécuter en en surveillant le déroulement afin de pouvoir procéder éventuellement à une replanification [BTT06].

Dans ce travail, les agents offrent un ensemble de services. Pour chaque agent, un réseau de Petri décrit les contraintes d'utilisation de ses services. On peut ainsi représenter l'exclusion mutuelle de services ou d'ensembles de services, ou limiter le nombre de services pouvant être exécutés simultanément par l'agent. Un agent peut être composé d'un groupe d'agents grâce à une hiérarchie d'agenticité.

L'objectif de la mission est décomposé en un arbre ET/OU qui décrit les sous-buts à satisfaire pour résoudre le but initial. Les tâches élémentaires (i.e. non décomposables) peuvent être résolues grâce à des recettes définies lors de la préparation de la mission. Une recette consiste en un ensemble de services permettant de réaliser une tâche : elle est représentée par un réseau de Petri coloré constitué d'une place et deux transitions, lesquelles portent la liste des services. Une recette est en outre assortie d'une indication de durée. Plusieurs recettes peuvent exister pour une même tâche, ce qui permet de l'exécuter de différentes manières. Le plan à exécuter par les agents se résume finalement à l'ensemble des recettes portées par les feuilles de l'arbre ET/OU. Toute la difficulté consiste alors à trouver une affectation des tâches aux agents qui respecte les contraintes imposées par ces derniers sur les services.

Le solveur de contraintes fournit un réseau de Petri temporel dont chaque place représente une recette. Chaque marquage atteignable dans ce réseau de Petri représente une affectation possible des tâches aux agents qui résout la mission initiale. A partir de ce réseau, un plan d'équipe hiérarchique est calculé afin de procéder aux affectations au sein de chaque agent composite.

Lors de l'exécution du plan, des aléas peuvent se produire et perturber l'exécution du plan, comme des pannes ou l'intervention de facteurs extérieurs. Un aléa peut se traduire par exemple par la disparition d'un service, ce qui rend bien sûr inutilisables les recettes composées de ce service. La replanification est alors nécessaire : elle consiste soit à remplacer l'agent qui ne peut plus fournir le service, soit à remplacer la recette qui a échoué par une autre recette n'utilisant pas le service qui n'est plus disponible. Le plan hiérarchique permet de connaître les équipes affectées par l'aléa et l'étendue de la replanification afin de ne replanifier la mission que localement si l'impact de l'aléa est limité.

Les réseaux de Petri constituent un excellent outil pour la collaboration des agents au sein d'une équipe, que ce soit au niveau de la synchronisation ou de l'affectation des tâches respectant les exclusions mutuelles de services. Ils souffrent toutefois d'une limitation au niveau de l'expressivité des opérateurs qui, s'ils peuvent intégrer des aspects temporels, ne peuvent en revanche décrire facilement des interactions et leurs effets entre deux agents.

2.6.5 POMDP et DEC-POMDP

Le processus de prise de décision est un processus complexe. En effet, à partir d'un état donné du monde, un choix s'impose entre un nombre parfois élevé d'actions qui permettent d'atteindre un nouvel état. Dans un MDP (Markov Decision Process), le déclenchement d'une action à partir d'un état de départ donné ne débouche pas systématiquement sur le même état d'arrivée : l'état d'arrivée est conditionné par des probabilités associées aux transitions.

Un problème MDP est représenté par un n-uplet (S, A, T, R) où :
- S est un ensemble d'états
- A est un ensemble d'actions
- $T : S \times A \times S \rightarrow [0, 1]$ est la fonction de transition du système qui donne la probabilité de transition d'un état s à un état s' lors du déclenchement de l'action a
- $R : S \times A \rightarrow \mathcal{R}$ est la fonction de récompense du système qui associe une récompense au déclenchement d'une action a à partir d'un état s.

La politique $\pi : S \rightarrow A$ est la suite qui indique l'action a à effectuer lorsque l'on se trouve dans l'état s. Le but de la planification par un MDP est de trouver la politique π qui maximise la récompense.

Dans certains systèmes, l'état actuel ne peut pas être connu avec certitude. Cette incertitude supplémentaire peut être modélisée par les POMDP (Partially Observable Markov Decision Process).

Un problème POMDP est représenté par un n-uplet (S, A, T, R, Π, O) où :
- S, A, T, R représente un MDP
- Π est un ensemble de symboles que l'on peut observer
- $O : S \times \Pi \rightarrow [0, 1]$ donne la probabilité $P(\omega|s)$ d'observer le symbole ω lorsque l'on est dans l'état s.

A partir des observations, un algorithme de résolution permet de déterminer l'état dans lequel le système se trouve, avec une certaine probabilité : on peut ainsi faire évoluer un ensemble de croyances sur l'environnement afin de déterminer l'action qui a le plus de chances de mener au but de l'agent. L'une des application est par exemple la géolocalisation dans un labyrinthe dont on possède le plan grâce à des observations locales.

Dans les problèmes multi-agent, les observations des agents peuvent être différentes, de par leur hétérogénéité ou plus simplement leur localisation dans l'environnement. On peut représenter ce problème décentralisé grâce au formalisme DEC-POMDP (pour DECentralised Partially Observable Markov Decision Process) [TBC06]. DEC-POMDP est semblable au formalisme POMDP à cette différence qu'il y a autant d'ensembles d'actions A et d'ensemble d'observations Π qu'il y a d'agents. La résolution d'un DEC-POMDP consiste à trouver un n-uplet de politiques individuelles permettant de maximiser la récompense globale du système. Cette résolution est prouvée NEXP-complet [BGIZ02].

Le formalisme POMDP (et son extention décentralisée DEC-POMDP) permet de décrire des problèmes dans lesquels il existe une double incertitude : la première concerne l'état dans lequel se trouve le système, la seconde réside dans les conséquences des actions. Les situations multi-agent dans lesquelles les agents ne sont pas omniscients contiennent cette part d'incertitude. Les POMDP permettent de représenter cette incertitude et de maximiser la fonction d'utilité de l'agent en apportant la réponse la plus adaptée sur le long terme. Toutefois nous n'avons pas utilisé les POMDP car ce travail se place dans un contexte déterministe :

nous souhaitons que l'agent prenne ses décisions en fonction de l'état de l'environnement qu'il a observé, sans établir d'hypothèses sur la validité de ses connaissances.

2.7 Le pathfinding

Certains programmes ont recours à l'intelligence artificielle dans un contexte géographiquement situé, c'est le cas par exemple des jeux vidéo. Ce contexte pose la question du déplacement : afin de connaître la direction à prendre au prochain pas de simulation, il est nécessaire de calculer un trajet reliant la position actuelle à la position souhaitée. On s'attend de plus à ce que ce trajet soit efficace ! « Le plus court chemin entre deux points est la ligne droite », nous dit la sagesse populaire. Cette affirmation n'est vérifiée que si l'environnement est parfaitement dégagé ! En effet, que ce soit dans le contexte de la robotique ou celui des jeux vidéo, l'une des principales contraintes du déplacement est la présence d'obstacles qu'il convient de contourner sans détours inutiles. C'est l'algorithme A* [BF81] qui est l'algorithme de recherche du plus court chemin (*pathfinding*) le plus utilisé aujourd'hui, en particulier dans les jeux vidéo [Nar04]. D'ailleurs, pour les concepteurs de jeux vidéo, le problème qui se pose actuellement est plutôt le traitement préalable qui consiste à analyser le terrain pour construire le graphe sur lequel s'appuie l'algorithme, afin d'accélérer le traitement [Sur02]. Notons qu'il existe d'autres algorithmes basés sur les waypoints[7] placés dans l'environnement par le programmeur afin de faciliter la recherche d'emplacements stratégiques [Whi07].

L'algorithme A* (figure 2.10) est considéré comme un algorithme de planification dans le sens où il est capable de trouver une succession d'opérateurs permettant de passer d'un état initial à un état final. Il permet de rechercher le plus court chemin dans un graphe d'états. Les états peuvent représenter l'état d'un plateau de jeu de type taquin, ou des positions géographiques. C'est ce dernier cas qui nous intéresse : l'algorithme A* permet de faire de la recherche de chemin. Dans ce cas, la valeur portée par un arc entre deux nœuds A et B représente la distance entre ces nœuds, ou le coût nécessaire pour passer de l'état du nœud A à l'état du nœud B. La force de l'algorithme réside dans l'utilisation d'une heuristique permettant de guider la recherche en profondeur dans le graphe, à la différence de l'algorithme de Dijkstra qui fait un parcours en largeur d'abord. On peut d'ailleurs obtenir ce comportement avec un A* dont l'heuristique renvoie toujours 0. Ces deux algorithmes donnent la garantie de trouver la solution optimale, mais l'algorithme A* est beaucoup plus rapide.

On utilise une fonction f capable d'évaluer le coût pour aller du nœud courant n au nœud de destination : $f(n) = g(n) + h(n)$, où $g(n)$ représente le coût du meilleur chemin connu pour aller de l'état initial à n, $h(n)$ est l'estimation heuristique du coût pour aller de n à l'état final. $f(n)$ est donc l'estimation du coût d'un chemin allant de l'état initial à l'état final en passant par n. Pour que l'algorithme fournisse une solution optimale, il est indispensable que la fonction heuristique $h(n)$ ne surestime jamais le coût réel pour aller de n à l'état final. Pour la recherche de chemin, on peut utiliser comme heuristique la distance « à vol d'oiseau » entre n et l'état final. Comme il est impossible que la distance réelle soit inférieure, on remplit bien le critère pour l'heuristique h. En outre, il faut que l'heuristique $h(n)$ fournisse une valeur assez proche de la valeur réelle afin que l'exploration du graphe soit limitée.

[7] les waypoints sont des emplacements particuliers qui permettent de simplifier un environnement continu en un environnement discret basé sur un graphe. Ils servent en particulier à la navigation GPS.

```
Algorithme A-étoile
Début
  ouverts = { état initial }
  fermés  = vide
  succès  = faux
  Tant que (ouverts non vide) et (non succès) faire
    choisir n dans les ouverts tel que f(n) est minimum
    Si est_final(n) Alors succès=vrai
    Sinon ouverts = ouverts privé de n
          fermés  = fermés + n
          Pour chaque successeurs s de n faire
            Si (s n'est ni dans ouverts ni dans fermés)
            Alors
              ouverts = ouverts + s
              père(s) = n
              g(s) = g(n) + h(n,s)
            Sinon
              Si (g(s) > g(n) + h(n,s)) Alors
                père(s) = n
                g(s) = g(n) + h(n,s)
                Si (s se trouve dans fermés) Alors
                  fermés  = fermés  - s
                  ouverts = ouverts + s
                Fin si
              Fin si
            Fin si
          Fin pour
    Fin si
  Fin TQ
Fin
```

FIG. 2.10 – Algorithme A*

2.7.1 Pathfinding dynamique : l'algorithme D*

L'algorithme A* est tout à fait indiqué pour calculer un chemin dans un environnement totalement connu. Cette hypothèse d'omniscience est idéale mais assez rare en réalité. En effet, que ce soit pour un véhicule autonome ou dans les jeux vidéo, l'environnement est découvert au fur et à mesure du déplacement de l'agent, mais n'est pas totalement connu au départ. S'il est vrai qu'on peut utiliser quand même l'algorithme A* pour se déplacer dans la partie découverte de l'environnement (tant que celle-ci peut-être représentée par un graphe connexe, évidemment), il est toutefois impossible d'utiliser cet algorithme pour atteindre une position qui se trouve dans une zone encore inconnue, même si sa position géographique absolue est connue. En effet, l'algorithme A* suppose l'existence d'un graphe dans lequel il existe au moins un chemin de la position actuelle vers la position désirée.

Il est intéressant de noter que l'utilisation de l'A* dans les jeux vidéo ne se limite pas au seul cas du pathfinding : dans certains jeux de stratégie, cet algorithme est utilisé pour l'analyse du terrain, la détection de points stratégiques ou la création de murs d'enceinte [Hig02].

L'algorithme D* [Ste94] propose une adaptation de l'algorithme A* dans un contexte non omniscient. Cette adaptation est incrémentale : elle construit une solution à partir des éléments connus au moment du calcul, et répare ensuite localement le chemin calculé au fur et à mesure de l'arrivée de nouvelles informations. Le plan « réparé » est optimal et équivalent à ce que l'on aurait obtenu en replanifiant totalement, mais il est obtenu plus rapidement : sur de gros problèmes, la réparation du plan par l'algorithme D* est réputée être quelques centaines de fois plus rapide que le recalcul complet d'un nouveau plan.

Au départ (donc en l'absence de toute information sur l'environnement), le chemin calculé par l'algorithme est une ligne droite entre la position actuelle et la position d'arrivée. Le robot ou l'agent commence alors à se déplacer et ses capteurs l'informent de la présence de nouveaux obstacles. Au fur et à mesure de leur découverte, le plan est adapté pour intégrer ces nouvelles informations.

La figure 2.11[8] présente un environnement jonché d'obstacles, représentés par des zones grises. L'agent se trouve au milieu du bord gauche de l'environnement et doit atteindre le bord droit. Le chemin présenté sur la figure de gauche est le chemin optimal calculé par un A* lorsque l'environnement est complètement connu dès le départ. La figure de droite présente le chemin calculé par l'algorithme D* lorsque l'agent découvre son environnement au fur et à mesure qu'il avance dans celui-ci.

L'hypothèse que nous venons d'étudier est celle d'un environnement « binaire » dans lequel un arc est franchissable (pas d'obstacle) ou non (présence d'obstacle). L'algorithme D*, tout comme l'A*, peut aussi être utilisé avec des coûts continus (arcs plus ou moins coûteux à franchir, en fonction du type de terrain ou de la déclivité, par exemple) afin de calculer le chemin le moins coûteux.

2.7.2 L'intérêt d'un pathfinding efficace

La recherche du plus court chemin est une partie critique de la conception d'une entité intelligente dans un jeu vidéo. En effet, la position (et son évolution dans le temps, caractérisée par le déplacement) est certainement la propriété la plus visible d'une entité. Le moindre faux-pas dans le déplacement est très vite remarqué et nuit immédiatement à la crédibilité du comportement. L'algorithme de déplacement d'un agent est certainement la partie du comportement dont une déficience sera la plus visible, voire la plus impardonnable pour un observateur. De plus, l'efficacité de l'algorithme en terme de rapidité d'exécution conditionne la fluidité des animations dans le cas d'un jeu vidéo, la vitesse de prise de décision dans le cas d'un robot, et la durée totale d'exécution dans le cas d'une simulation. Le pathfinding constitue donc un point névralgique d'une simulation géographiquement située.

[8] images extraites de `http://www.frc.ri.cmu.edu/~axs/dynamic_plan.html`

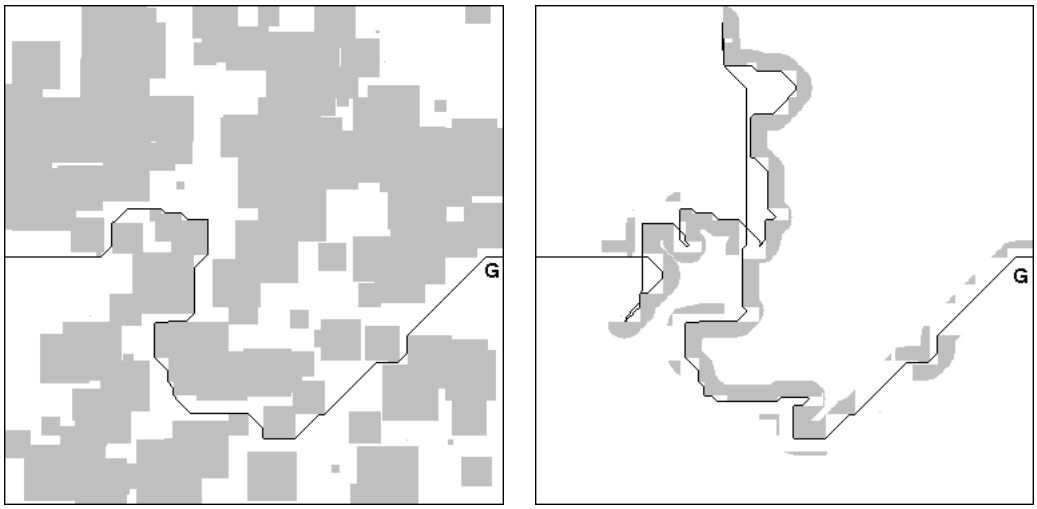


FIG. 2.11 – L'algorithme D*. Sur ces figures, les zones grisées correspondent à des obstacles infranchissables. La figure de gauche présente le trajet optimal calculé par l'algorithme A* qui connaît totalement l'environnement dès le départ. La figure de droite présente le trajet calculé par l'algorithme D* en découvrant les obstacles au fur et à mesure du déplacement. Le résultat obtenu par l'algorithme D* est proche de celui fourni par l'A*.

Calculs préliminaires

Les environnements utilisés dans les jeux vidéo sont très vastes. Il faudrait que la grille permettant de découper l'environnement en zones élémentaires (nœuds du graphe de l'A*) soit suffisamment fine pour deux raisons principales : obtenir des déplacements fluides (pas de "marches d'escaliers") et précis (pour entrer dans un bâtiment par la porte ou passer entre deux obstacles rapprochés). Il se poserait un problème d'échelle évident, car l'algorithme, même s'il est rapide, ne pourrait pas travailler sur le graphe énorme qui en résulterait. Un calcul préliminaire doit donc être fait sur l'environnement pour ne subdiviser que les zones qui le nécessitent. Ce calcul peut-être fait hors-ligne, et on obtient un graphe qui n'est plus uniforme au niveau des distances mais qui est en revanche assez léger (figure 2.12[9]).

2.8 Les simulations

Dans le processus de recherche scientifique, l'expérimentation est une phase bien souvent indispensable. Elle permet d'observer des phénomènes, attendus ou non, grâce à la mise en scène minutieuse de divers composants. Bien souvent, l'expérience modifie ou détruit tout ou partie des composants utilisés. Cela rend parfois nécessaire le recours à un substitut de la réalité lorsque les composants considérés sont uniques, chers ou non renouvelables, ou lorsque les grandeurs considérées rendent l'expérimentation trop dangereuse (simulation d'explosion nucléaire, simulation de mouvements de foule...).

[9]image extraite de `http://www.vieartificielle.com/article/index.php?action=article&id=180`

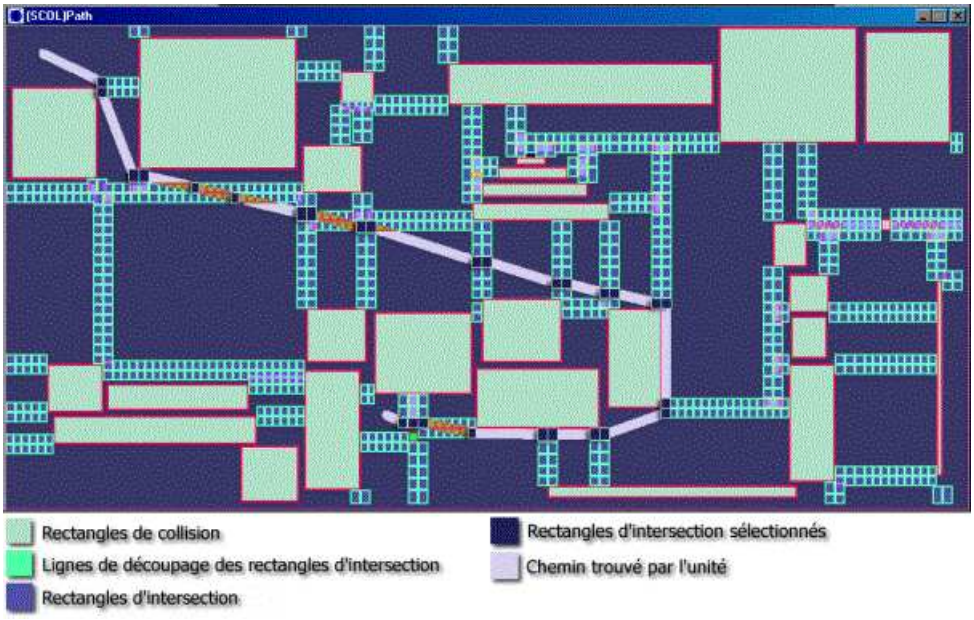


FIG. 2.12 – Exemple de prédécoupage d'un vaste environnement en vue de l'utilisation d'un A*.

La simulation comporte une phase préalable très importante qui est la modélisation : elle consiste à établir un modèle du phénomène que l'on souhaite simuler. Ce modèle peut-être obtenu par différents moyens :

- de manière empirique en observant quelques expériences et en essayant d'en décortiquer les rouages (certaines modélisations statistiques)
- mathématiquement en utilisant les lois physiques du système observé afin de le mettre en équation (simulation nucléaire, simulation d'éléments mécaniques, simulations météorologiques...)
- analytiquement en résolvant un système dynamique dont on cherche à retrouver les paramètres (simulations biologiques, par exemple simulation des interactions entre les protéines)
- par agents dont on programme le comportement (simulations animales, simulations économiques...)

La simulation proprement dite consiste ensuite à utiliser le modèle créé, c'est-à-dire à en appliquer les règles afin de pouvoir observer et analyser l'évolution du système.

Un modèle de simulation peut avoir une visée descriptive ou explicative. Dans le premier cas, il s'agit de déterminer l'état futur du système, c'est-à-dire comment celui-ci va évoluer, mais il n'est pas nécessaire de savoir pourquoi il évolue de cette manière. A l'inverse, la vocation d'un modèle explicatif est de rendre compte des mécanismes sous-jacents qui provoquent une évolution donnée du système. La simulation par équations différentielles permet de connaître la dynamique globale d'un système (visée descriptive) mais ses capacités explicatives sont limitées : il est possible de mesurer l'influence des différents paramètres de la simulation, mais il n'est pas aisé de comprendre les répercutions des interactions entre les entités sur l'évolution globale du système. Les simulations centrées individu mettent en revanche l'accent

sur les acteurs présents dans le système et sur le rôle qu'ils ont au sein de celui-ci (visée explicative) [BLM02]. Nous défendons un nouveau type de simulation basée sur un modèle centré interaction. Ce modèle permet les mêmes analyses que la simulation centrée agent dont elle se rapproche, mais elle offre deux avantages majeurs qui sont la réutilisabilité des interactions et la facilité de modification du comportement d'un agent par l'ajout ou le retrait d'interactions. Nous le verrons dans la section de la contribution consacrée au modèle.

On peut regrouper les simulations en deux grandes familles : les simulations à visée scientifique, éducative ou pédagogique (plates-formes de simulation par agents) et les simulations à visée ludique (jeux vidéo).

2.8.1 Un exemple de simulation par agents

Dans certains corps de métiers comme l'Armée ou les Sapeurs-Pompiers, il existe des situations types face auxquelles le professionnel doit savoir comment se comporter. L'étude de ces situations lors de la formation n'est pas toujours facile : en effet, certaines expériences ne peuvent pas être simulées facilement pour des raisons de sécurité ou de budget. C'est par exemple le cas de l'incendie d'un immeuble, qu'il est pas possible de reproduire à des fins d'entraînement. La simulation est donc la seule manière pour le futur professionnel de rencontrer ces situations et d'apprendre à les gérer. Elle offre en outre la possibilité de plonger l'apprenant dans l'environnement pour qu'il soit actif dans la formation au lieu de la subir. L'utilisation d'agents est alors essentielle pour que la simulation s'adapte au comportement de l'apprenant : leurs capacités de perception, de décision et d'action leur permettent de prendre en compte rapidement les modifications qu'il effectue dans l'environnement.

C'est dans ce cadre que se place la thèse de Ronan Querrec [Que02][Che06]. Dans le modèle MASCARET, il propose un système multi-agent hétérogène composé d'une part d'agents réactifs représentant les éléments physiques du système et d'autre part d'agents rationnels dotés de comportements réactifs, cognitifs et sociaux qui simulent le comportement des personnes intervenant dans la simulation. L'apprenant est considéré par le système comme un agent rationnel particulier. Il perçoit et agit dans la simulation par le biais d'un avatar en jouant un rôle dans la structure organisationnelle de l'équipe.

Les agents réactifs sont régis par des comportements simples dictés par une formule physique en fonction de leur état et des facteurs extérieurs. Ils interagissent ensemble par des influences locales, par exemple un transfert d'énergie. Les agents cognitifs ont une part de comportement réactif pour gérer les facteurs physiques qu'ils subissent de la part de l'environnement et des autres agents. Ils font partie d'une structure organisationnelle dans laquelle ils ont un rôle. Ce rôle est une procédure constituée d'un ensemble d'actions à effectuer. Leur moteur cognitif leur permet de décider de leur prochaine action en fonction le l'état du déroulement de leur procédure et des demandes des autres agents cognitifs.

Le modèle MASCARET est utilisé dans l'application SÉCURÉVI (figure 2.13) destinée à la formation des officiers de sapeurs-pompiers. Elle permet de simuler l'environnement physique d'une intervention ainsi que différents phénomènes (feu, fumée) ou outils (jeu d'eau, queue de paon). L'apprenant joue un rôle dans l'organisation tandis que le formateur peut intervenir simultanément dans la simulation pour jouer un rôle dans l'organisation ou provoquer des dysfonctionnements.

FIG. 2.13 – L'application SÉCURÉVI basée sur le modèle MASCARET.

2.8.2 Les plates-formes d'agent

Avec la montée en puissance des SMA, beaucoup de plates-formes de développement ont vu le jour ces dernières années. Ces plate formes proposent des outils qui donnent la possibilité de développer un système multi-agent, de le déployer et de le faire s'exécuter. La plate-forme offre des services au développeur pour l'aider dans sa tâche, mais aussi aux agents lors de l'exécution du système créé. Via ces services, la plate-forme abstrait le système sur lequel elle se trouve (à l'instar d'un système d'exploitation). Des efforts sont faits actuellement pour normaliser les agents afin d'assurer une conception uniforme permettant leur interopérabilité, en particulier par la FIPA (Foundation of Intelligent Physical Agents) avec *FIPA 97* qui établissait le premier ensemble de spécifications. Ce cadre normatif permet de faire communiquer et interagir des agents créés par des constructeurs différents.

Les plates-formes

Il existe beaucoup de plates-formes visant à aider à la conception de systèmes multi-agents. On peut citer entre autres :

- **Madkit** (développée par Olivier GUTKNECHT et Jacques FERBER au Laboratoire d'Informatique, de Robotique et de Micro-électronique de Montpellier (LIRMM) de l'Université Montpellier II). Elle est basée sur les trois concepts de base de Aalaadin : les agents, les rôles et les groupes afin de construire des applications utilisant des agents hétérogènes appartenant à des organisations.

- **SWARM** (développée initialement par Nelson Minar et Chris Langton au Santa Fe Institute) fournit des bibliothèques permettant de réaliser des simulations qui mettent en jeu des ensembles d'agents en concurrence.
- **SimuLE** (développée par l'équipe SMAC) qui permet de créer des simulations comportant un très grand nombre d'agents réactifs. Elle est basée sur le modèle centré interaction (cf 3.1).
- **NetLogo** (développée par Uri Wilensky et le CCL (Center for Connected Learning and Computer-Based Modeling)) qui permet d'étudier les systèmes décentralisés.

Peu de ces plate formes sont adaptées pour réaliser des simulations d'agents cognitifs avec des contraintes géographiques. Nous allons toutefois étudier l'une de ces plate formes en détail.

NetLogo

NetLogo est une plate-forme de développement d'agents adaptée à la simulation de phénomènes naturels et sociaux. Un langage de programmation proche du logo[10] est mis à disposition pour programmer les agents. Il est possible de faire évoluer des centaines d'agents simultanément sur ce type de plate-forme afin d'observer les comportements au niveau micro et macro du système multi-agent.

Les fonctionnalités de la plate-forme

La plate-forme NetLogo, illustrée en figure 2.14, est programmée en java, ce qui lui garantit par nature de pouvoir s'exécuter sur n'importe quelle machine ou système, pourvu que celui-ci soit équipé d'une JVM. Le langage utilisé pour coder les agents est complet : il est possible après quelques dizaines de minutes de prise en main de développer des agents simples afin de réaliser ses premières simulations. La plate-forme garantit la reproductibilité des résultats d'une simulation, quelle que soit la machine ou le système sous-jacent.

L'environnement est constitué de zones contiguës, appelées *patches*, dans lesquelles les agents évoluent. Les versions récentes de NetLogo permettent un affichage en 3 dimensions de l'environnement, ce qui permet de mieux visualiser le déroulement de certaines simulations.

L'interface est composée de plusieurs éléments modulaires que l'on peut disposer à son gré lors de la création d'une simulation :

- L'environnement dans lequel évoluent les agents. On peut voir les agents se déplacer et il est possible de visualiser certaines caractéristiques des patches en gradians de couleurs. Par exemple si l'on a réalisé une simulation de colonie de fourmis, il est possible de visualiser par un dégradé de couleur la quantité de phéromones déposée par celles-ci dans l'environnement.
- Boutons de contrôle. Lors de la création d'une simulation, on peut ajouter des boutons de contrôle de plusieurs types, offrant ainsi la possibilité à l'utilisateur de la simulation de paramétrer celle-ci simplement sans devoir toucher directement au code.
- Fenêtres de visualisation. Des graphiques prédéfinis sont mis à disposition du concepteur afin d'être incorporés à la fenêtre de simulation. Ceux-ci offrent la possibilité de visualiser l'évolution de paramètres de la simulation, par exemple le taux de requins et de poissons

[10] Le logo est un langage de programmation pédagogique inventé dans les années 1970. Il consiste à déplacer une « tortue » à l'écran à l'aide de primitives simples comme *avancer*, *tourner*...

dans une simulation du jeu de la vie *Wator* [Hay84], dérivé des automates cellulaires définis par J. Conway en 1970 [Gar70].

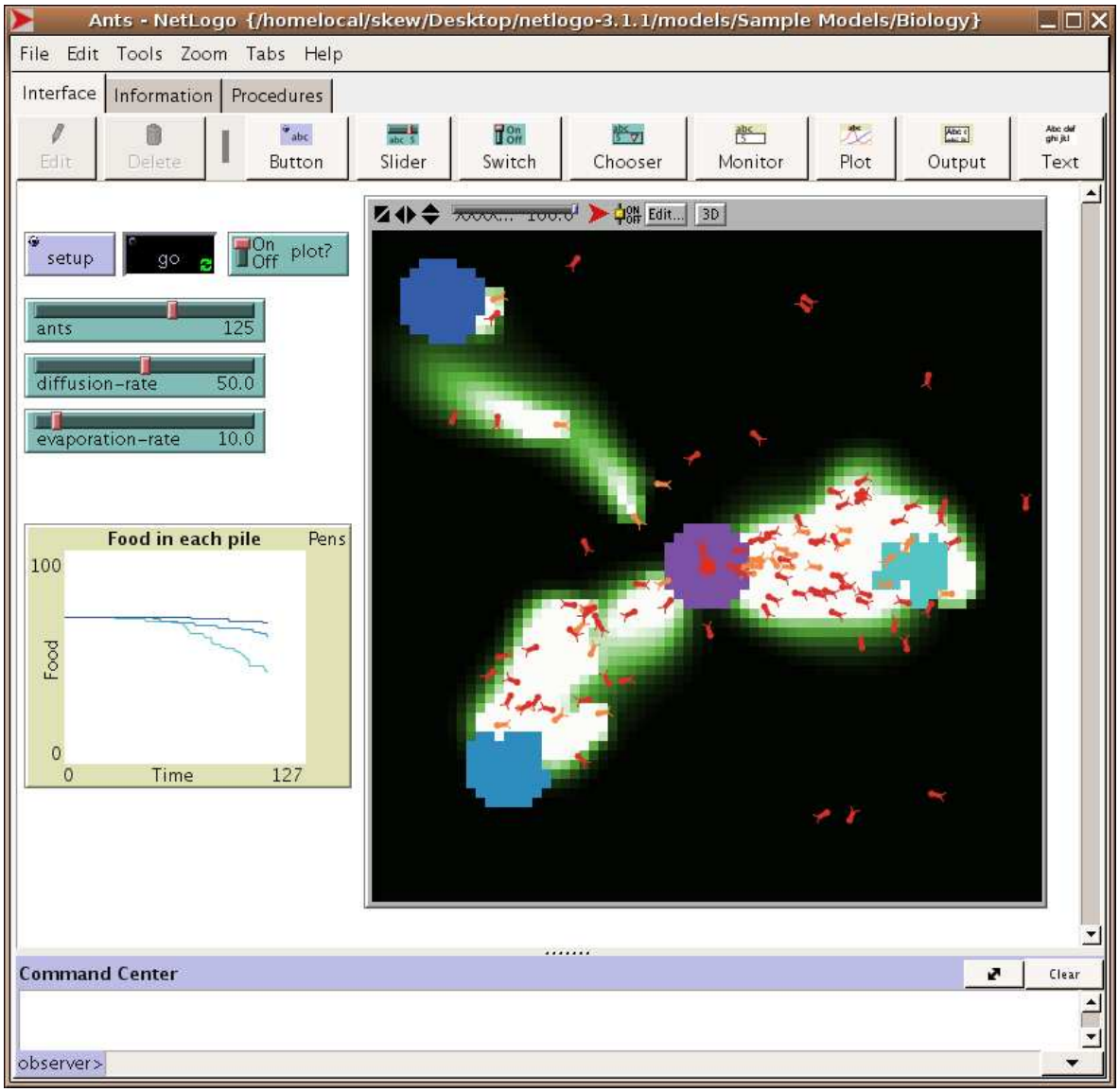


FIG. 2.14 – Simulation d'une colonie de fourmis dans la plate-forme NetLogo.

La plate-forme permet d'effectuer une grande variété de simulations : simulations écologiques, géologiques, sociales, mathématiques... Cet outil est très adapté pédagogiquement et accessible aux non-informaticiens grâce à la naïveté du langage utilisé. Toutefois, pour concevoir des simulations un peu complexes, les compétences d'un informaticien sont assez vite requises.

Pourquoi n'avons-nous pas utilisé NetLogo ?

Malgré tous les aspects positifs de cette plate-forme, nous ne pouvons pas l'utiliser. Elle est tout à fait adaptée pour créer des agents réactifs et les utiliser en grand nombre, mais on peut difficilement l'utiliser pour concevoir des agents cognitifs comme les nôtres. En effet, le langage qui est fourni pour coder les agents, même s'il est complet, n'est pas très adapté, d'une part parce qu'il est interprété et donc lent, et d'autre part parce qu'il ne permet pas de manipuler des données organisées de manière complexe, ce qui est pourtant indispensable pour concevoir un agent cognitif. D'autre part, l'environnement dans lequel se déplacent nos agents nécessite de coder des conditions entre les « patches », ce qui est difficilement exprimable dans la plate-forme NetLogo. De plus, les agents créés avec la plate-forme NetLogo sont ordonnancés par la plate-forme sans que le créateur de la simulation puisse intervenir. Il peut apparaître des effets de bords dus à l'ordonnanceur. Ces problèmes sont bien connus dans la communauté "systèmes embarqués" et "systèmes temps-réel" pour les cas d'ordonnancement de tâches sur plusieurs processeurs. Enfin, NetLogo ne permet pas de séparer les aspects déclaratifs et procéduraux d'une simulation, ce qui complique beaucoup la réutilisation des comportements.

SimuLE

La plate-forme SimuLE (SIMUlations Large Echelle) a pour objectif d'étudier les techniques logicielles mises en œuvre pour réaliser des systèmes multi-agents réactifs situés à large échelle. Les simulations sont *situées* dans la mesure où les agents sont positionnés dans un espace métrique et qu'ils sont obligés de se déplacer pour mener à bien leurs actions. Les agents utilisés dans ces simulations sont des agents réactifs : ils réagissent uniquement aux variations de leur environnement géographique et/ou social grâce à des règles. Il n'est pas possible d'utiliser des agents cognitifs étant donné que le but de la plate-forme est de faire évoluer des milliers ou dizaines de milliers d'agents, tout en conservant des comportements affichables en temps-réel. La plate-forme fournit un moteur de comportements, il n'est pas nécessaire de coder le comportement des agents. Elle est prévue pour réaliser des simulations multi-échelles et autorise la création dynamique d'agents par les autres agents de la simulation. Elle contient un modèle statistique qui permet d'exploiter les données de la simulation et d'en afficher l'évolution en temps-réel.

Le modèle d'agents est basé sur les interactions (c'est le modèle *peut-subir/peut-effectuer* qui est aussi utilisé dans notre plate-forme d'agents cognitifs) et sur la notion de hiérarchie. Les agents réactifs fonctionnent selon un modèle réactif à trois niveaux :

1. recensement des cibles possibles
2. recensement des interactions possibles avec ces cibles
3. choix d'une interaction par ordre de priorités

2.8.3 Les Jeux Vidéo

IA et Jeux

Actuellement, l'industrie du jeu vidéo brasse plus d'argent que l'industrie du cinéma. Pour beaucoup des jeux actuels, l'intérêt réside fortement dans le scénario entourant le jeu et qui en définit le contexte. L'aspect design, graphique et sonore contribuent à l'ambiance et permettent

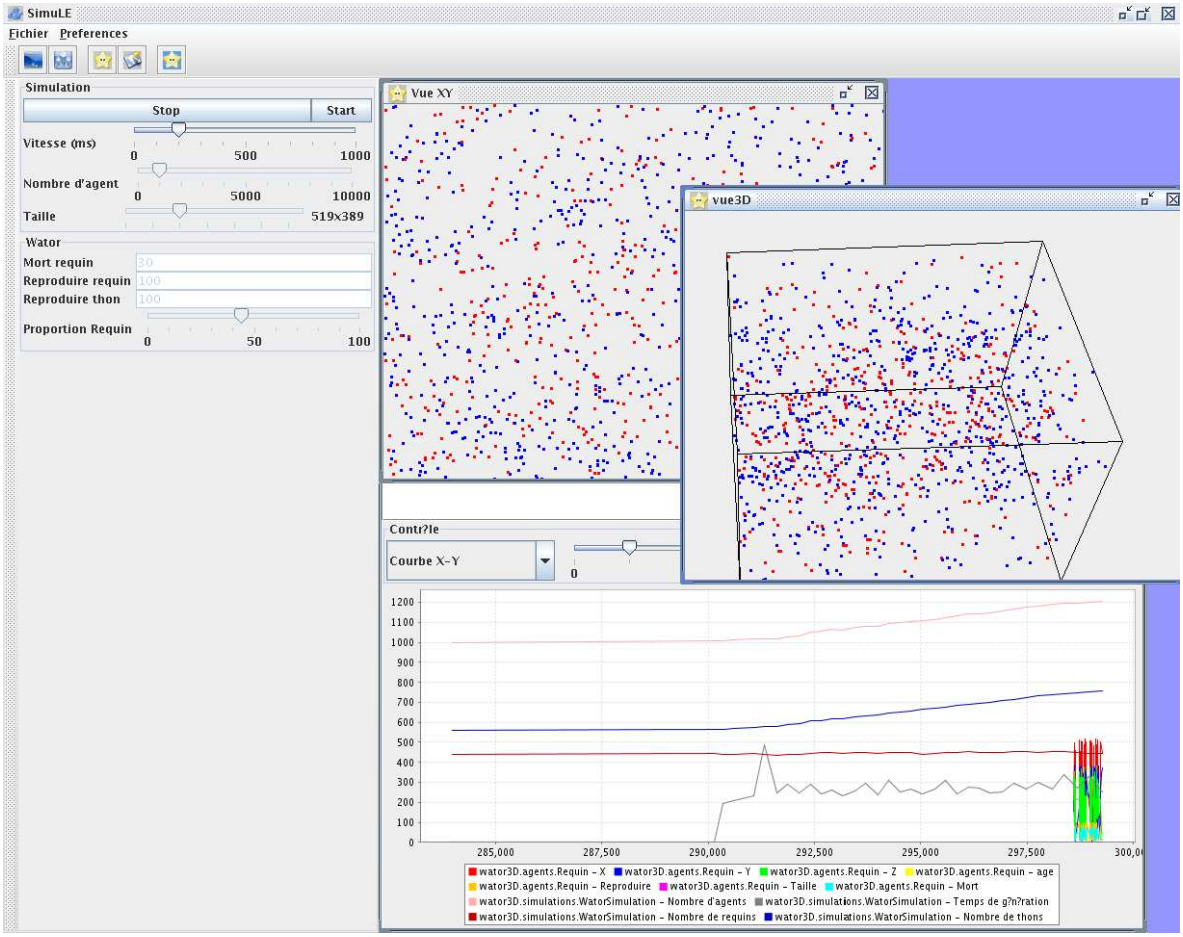


FIG. 2.15 – La plate-forme SimuLE. Le bandeau de gauche permet de contrôler dynamiquement les paramètres de la simulation. A droite, on trouve les visualisations 2D et 3D ainsi que l'affichage des courbes suivant l'évolution des paramètres de la simulation.

de renouveler le genre en créant des jeux nouveaux basés sur les mêmes moteurs. Toutefois une caractéristique récente des jeux vidéo est qu'ils nécessitent une intelligence artificielle de plus en plus complexe et donc gourmande en ressources, ce qui est rendu possible par l'évolution des processeurs. L'importance des jeux vidéo en terme d'enjeux pour l'intelligence artificielle n'est donc pas négligeable. Selon John Laird, les jeux vidéo constituent actuellement la « killer application » de l'intelligence artificielle [LVL00].

Les objectifs de l'IA dans les jeux vidéo

Les objectifs de l'intelligence artificielle dans les jeux sont multiples. L'IA a pour but d'animer un ou plusieurs PNJ[11] qui interviennent dans le jeu et entrent en interaction avec le joueur. Le PNJ peut avoir des statuts très différents dans les jeux :

[11] Personnage Non Joueur

- **Adversaire** : dans les FPS[12]. Cette catégorie regroupe les jeux dont le but est de tirer sur les ennemis afin de les éliminer et de pouvoir progresser dans le jeu. On peut citer parmi les plus anciens représentants *Doom*, *Wolfenstein*, ou *Duke Nukem* et plus récemment la série des *Quake*, *Unreal Tournament*, *Half Life*, *Far Cry*. Dans les jeux de simulation automobile, l'intelligence artificielle permet de piloter les véhicules des adversaires.
- **Coéquipier** : dans les RPG[13] ou RTS[14]. Le but des premiers est de faire évoluer un ou plusieurs personnages en réalisant des quêtes. Les seconds consistent assez souvent à une conquête de territoire ou de ressources. Le joueur dispose d'unités qu'il dirige et auxquelles il confie des objectifs. Le but de l'intelligence artificielle et de mener à bien ces objectifs : déplacer efficacement les unités en contournant les obstacles, combattre les ennemis rencontrés sur le chemin. On peut citer *Dune* (l'un des premiers représentants du genre), *Warcraft*, *Age Of Empires* et bien d'autres. Les coéquipiers peuvent aussi apparaître comme des alter ego dans des jeux comme *SWAT* ou *BattleField*. Le joueur est dans ce cas plongé au sein d'une équipe à la fois en tant qu'acteur et en tant que coordonnateur. On ne peut évidemment pas omettre de citer les jeux de football comme *FIFA* dans lesquels l'ordinateur gère une équipe de joueurs, tant au niveau micro (le comportement de chaque joueur) qu'au niveau macro (le comportement et la coordination de l'équipe).
- **Autres** : certaines entités ne sont ni des ennemis ni des coéquipiers : c'est le cas dans le jeu *Créatures* ou le joueur doit éduquer des *Norns* qui évoluent dans un environnement complexe et parfois hostile. Ces entités sont gérées par un réseau de neurones. Dans le jeu *Les Sims*, qui est le jeu le plus vendu au monde, le joueur dirige un personnage afin de lui faire exécuter et apprendre les gestes de la vie quotidienne. Les autres personnages qui interviennent dans le jeu sont dirigés intégralement par l'ordinateur.

Il est intéressant de remarquer que dans certains jeux l'IA est à la fois utilisée pour gérer les ennemis et les coéquipiers : c'est le cas, entre autres, pour les jeux de football où l'ordinateur gère non seulement l'équipe adverse, mais aussi les coéquipiers du joueur humain lorsqu'il ne les contrôle pas.

Critères d'une « bonne » IA

Pour le joueur, la capacité du jeu à reproduire le monde réel devient de plus en plus importante. Ceci est vrai tant au niveau du graphisme que des comportements. Le pouvoir immersif du jeu dépend fortement de ces deux critères. C'est pourquoi l'IA se doit de tendre vers l'irréprochable pour ne pas brutalement sortir le joueur de l'univers dans lequel il est plongé par un comportement inadapté. Mais qu'attendent exactement les joueurs ? Quelles sont les erreurs frappantes qui l'agacent et le ramènent brutalement à la réalité ?

Avant toute chose, le comportement d'un PNJ se doit d'être crédible. Ainsi il n'est pas question qu'un soldat coure après une grenade qu'il vient de lancer, ou qu'un pilote fasse demi-tour pour repasser un virage raté. Il n'est par contre pas rédhibitoire que le PNJ commette des erreurs, et cela est même souhaitable si l'on veut une IA réaliste. Toutefois, ces erreurs ne doivent pas s'apparenter à des bugs sous peine de décrédibiliser le comportement du PNJ.

[12] First Person Shooter
[13] Role PLaying Game
[14] Real Time Strategy

On peut citer plusieurs exemples de comportements aberrants dans des grands noms du jeu vidéo :

- **Dans le jeu *Counter Strike Condition Zero***, les ennemis possédaient un halo de détection au delà duquel ils ignoraient tout événement. Ainsi il était possible en étant un peu en retrait de tirer à plusieurs reprises sur un garde qui faisait des allées venues, sans que celui-ci n'arrête son tour de garde pour se défendre. Le moteur de type réactif basé sur des automates à états, bien que reconnu pour être excellent[15] dans ce jeu, montre ici clairement ses limites.
- **Le jeu *Neverwinter Nights*** proposait un éditeur pour créer des scénarios. Grâce à cet éditeur il était possible de tester le moteur d'intelligence artificielle des personnages. Certains défauts ont ainsi pu être mis en évidence[16] : dans le scénario créé, une statuette se trouve dans une maison. Cette maison est fermée à clef, et cette clef est disposée dans une autre maison, elle-même fermée à clef, et ainsi de suite. La dernière clef se trouve dans l'environnement. Le but du personnage est de s'emparer de la statuette. Pour cela, il se dirige vers la maison où se trouve la statuette, et se rend compte que la porte est fermée à clef. Il se dirige donc vers la maison où se trouve cette clef, et ainsi de suite jusqu'à ce qu'il ramasse la clef se trouvant au centre de l'environnement. Jusque là le comportement est tout à fait correct. Mais une fois la dernière clef ramassée, le personnage adopte un comportement étrange : il revisite l'une après l'autre toutes les maisons dans le même ordre parce qu'il a oublié la raison pour laquelle il a ramassé toutes ces clefs ! Il manque une pile de buts [Toz04] à l'IA pour retenir l'enchaînement des buts qui l'ont amené à résoudre le but actuel.
- **Dans le premier volet du jeu *Les Sims***, il n'y avait pas de réordonnancement des buts. En effet, si l'on demandait au personnage présent dans la cuisine d'aller chercher le courrier et de vider la poubelle, celui-ci faisait un premier aller-retour à l'extérieur pour aller chercher le courrier puis prenait ensuite le sac poubelle pour aller le porter à la benne. Or la benne et la boîte aux lettres se trouvant l'une à côté de l'autre devant la maison, il aurait été plus logique que le personnage prenne le sac poubelle avec lui pour aller chercher le courrier. Le moteur de comportement ne réorganisait pas les buts en fonction de leur proximité afin d'éviter des trajets inutiles. De manière générale, l'IA de ce jeu semble assez limitée, et on voit clairement que les personnages réagissent en fonction de critères simples : quand le niveau d'un indicateur (faim, fatigue, ennui...) passe en dessous d'un certain seuil, l'action permettant d'y remédier est immédiatement ajoutée à la pile de buts. Ces comportements simplistes n'ont pourtant pas empêché ce jeu de rester le titre le plus vendu pendant deux années consécutives...

Les deux derniers exemples montrent qu'un maintien et une réorganisation des buts sont souvent nécessaires afin d'obtenir un comportement réaliste, ou plutôt d'éviter certains comportements absurdes. Il est possible d'observer l'influence des différentes gestions de buts grâce à l'applet Princess présentée sur la page de l'équipe SMAC : `http://www.lifl.fr/SMAC/projects/cocoa/princesse.html`

L'un des problèmes qui paraît le plus trivial au joueur non-informaticien est certainement celui du déplacement, très intuitif. De plus, le déplacement est très facilement observable par le joueur. Ce problème est donc crucial : un mauvais pathfinding sera très vite remarqué et la

[15] selon les magazines spécialisés et les joueurs

[16] Ce travail encadré par Jean-Christophe Routier a été réalisé par Alexandre Vandenabeele et Alexandre Liagre dans le cadre de leur stage de Maîtrise en juin 2003

crédibilité du comportement du PNJ sera très vite affectée à cause d'une erreur de parcours. Ce fût par exemple le cas du jeu Diablo dans lequel il arrivait qu'un PNJ soit bloqué par un obstacle.

Dans les jeux vidéo, l'IA est soumis à une contrainte particulière : comme ailleurs, elle doit être efficace, mais dans le cadre des jeux vidéo, elle ne doit pas l'être trop. Un joueur ne supportera longtemps pas d'être battu à plate couture systématiquement et perdra goût à ce jeu. L'IA doit être adaptée au niveau du joueur afin que celui-ci ressente un « challenge » : elle ne doit être ni trop faible ni trop forte. Généralement l'IA est donc paramétrable en terme de difficulté, et le niveau choisi influe sur le comportement du ou des PNJ : c'est le cas par exemple pour un jeu d'échec où l'on peut limiter le nombre de coups calculés à l'avance par l'adversaire (et donc son niveau de difficulté) en changeant la profondeur de recherche dans l'algorithme MinMax.

Evidemment, tricher n'est pas un comportement acceptable dans les jeux. Un autre critère essentiel de l'IA dans les jeux vidéo est donc qu'elle respecte scrupuleusement les règles du jeu imposées au joueur, qui ne peut bien souvent pas les contourner. Pourtant dans certains jeux l'IA semble se permettre des écarts en accédant à des informations auxquelles le joueur n'a pas accès ou en construisant des unités sans avoir les ressources nécessaires (cela semble communément admis dans la communauté des joueurs de *Civilisation*). Outre la tricherie « volontaire », certaines IA sont contraintes de tricher. C'est le cas par exemple des bots qui peuplent les FPS : les PNJ jouent à partir d'une carte du terrain et non à partir d'une visualisation en 3D comme le fait le joueur. D'une part il serait quasiment impossible de calculer une vue pour chacun des joueurs (humain et PNJ) présents et d'autre part, l'analyse de ces vues afin de déterminer le comportement du PNJ serait beaucoup trop complexe. L'IA joue donc à l'aide d'une carte, mais cela peut l'avantager dans certains cas.

Parmi toutes les techniques d'IA, seules quelques-unes sont généralement employées dans les jeux vidéo pour leur facilité de mise en œuvre. Il s'agit des scripts (dans *Unreal Tournament* par exemple un éditeur permet de modifier ces scripts), ou des automates à états (*Half Life* les utilise avec brio). L'inconvénient des scripts est leur rigidité : les comportements ne sont pas assez variés et l'on peut assez rapidement deviner la réaction d'un adversaire. De plus, ils enferment le joueur dans un scénario rigide et ne lui permettent pas beaucoup de créativité pour résoudre les problèmes qu'on lui pose [Toz02]. Toutefois, certains jeux utilisent les scripts efficacement et le résultat est assez convaincant. Ceci soulève une réflexion cruciale sur l'intelligence artificielle dans les jeux vidéo : l'important est bien souvent le résultat, et pas la technique utilisée. Qu'un joueur soit encerclé parce que cela a été prévu au départ ou parce que le moteur d'intelligence artificielle l'a calculé en temps réel, cela ne change rien pour le joueur. De même, certaines animations participent énormément au réalisme du jeu : c'est le cas du bûcheron que l'on voit couper les arbres à la hache alors qu'à ce moment même son comportement est des plus simples puisqu'il consiste à incrémenter la quantité de bois de son inventaire à intervalles de temps réguliers. Les concepteurs de jeux l'ont bien compris et préfèrent miser sur des techniques simples à mettre en œuvre et peu coûteuses comme les scripts, secondés par des animations, quitte à ne pas pouvoir réutiliser leur IA dans d'autres jeux.

La réutilisation de l'IA est pourtant une préoccupation grandissante. Il y a quelques années, le graphisme avait une grande place dans le développement des jeux vidéo. Le besoin de bibliothèques réutilisables s'est rapidement fait ressentir, et ces bibliothèques ont fini par s'im-

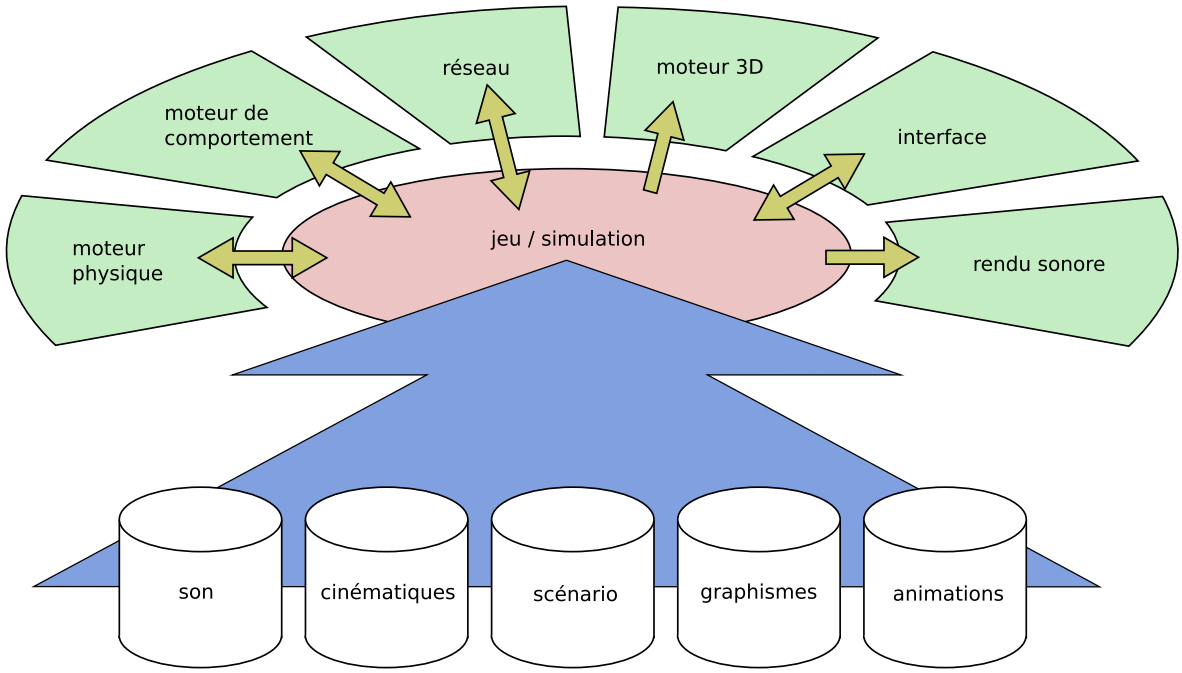


FIG. 2.16 – Modèle général d'un jeu vidéo : à partir des données composant les ingrédients du jeu, différents moteurs se chargent du rendu propre à chaque composante du jeu. Alors que le module graphique représentait par le passé une part très importante, aujourd'hui c'est le module de comportement qui demande le plus d'efforts au niveau du développement et qui est le plus gourmand en temps CPU. Cet essor de l'IA dans les jeux vidéo a été permis par la libération du processeur avec l'arrivée des cartes graphiques.

poser en même temps que les cartes graphiques, en permettant un développement plus rapide des graphismes. Actuellement, c'est l'intelligence artificielle qui a une place grandissante dans les jeux vidéo, au point de devenir une préoccupation centrale (figure 2.16). Il devient donc nécessaire de pouvoir réutiliser les moteurs d'IA d'un jeu à un autre, ou d'utiliser des bibliothèques proposant des comportements pour les PNJ. Dans cette optique Kynogon [Kyn06] et DirectIA [MAS06] proposent un ensemble de bibliothèques de comportements utilisables dans les jeux (ou d'autres applications) afin de réduire les coûts de développement tandis que AIseek [AIS06] propose une carte d'accélération matérielle pour les calculs d'intelligence artificielle. Simbionic [Sto06] propose quant à lui une interface qui permet, sans connaissances en programmation, de créer des comportements évolués pour les simulations. Des actions atomiques sont proposées au concepteur, et il peut les placer sur un graphe à l'aide de la souris en leur ajoutant des conditions et des déclencheurs afin d'obtenir un automate. Les primitives sont nombreuses et l'interface est simple, toutefois il n'est pas évident que créer un comportement complexe soit à la portée d'un utilisateur qui ne connaît pas les automates. Dans le même esprit, on peut aussi citer SpirOps [Spi06] qui est utilisé dans le jeu *Splinter Cell - Double Agent*.

Conclusion

L'intelligence artificielle a connu un essor grandissant ces dernières années, et occupe aujourd'hui une place prépondérante dans les jeux vidéos. Dans le domaine de la simulation de comportements, les systèmes multi-agents offrent une réponse intéressante aux problèmes actuels de l'IA, mais souffrent encore de lacunes : le modèle centré agent qui est communément utilisé ne permet pas de réutiliser facilement les comportements composants d'une simulation à une autre. Le domaine du jeu vidéo est très gourmand en comportements. Pourtant aujourd'hui, force est de constater que l'intelligence artificielle dans les jeux est limitée par les moyens adoptés : les scripts utilisés usuellement pour développer les comportements sont trop rigides et interdisent la réutilisation de parties du comportement, ce qui oblige à réécrire presque entièrement le moteur d'IA à la création d'un nouveau jeu. Les outils actuels n'intègrent pas les déplacements dans la planification à moins de les considérer comme des téléportations, ce qui nuit pourtant à la crédibilité des comportements observés. Dans une seconde partie je propose des solutions pour pallier ces différents problèmes : le modèle centré interaction permet de sortir les interactions du code des agents afin de rendre les comportements réutilisables. En anticipant la gestion des déplacements au moment de la planification, certains déplacements inutiles sont évités, ce qui renforce la crédibilité du comportement. Je propose aussi d'intégrer dans le modèle une gestion d'équipe qui préserve l'autonomie de décision des agents.

Deuxième partie

Contribution

Chapitre 3

Le modèle d'interactions

« L'arbre se venge de celui qui le coupe en s'abattant sur lui »
Léonard de Vinci

Introduction

Le modèle généralement utilisé pour créer les simulations de systèmes multi-agents est un modèle centré agent. Ce modèle permet une compréhension du comportement de chaque agent de la simulation, à l'opposé d'un modèle analytique par équations différentielles qui, s'il permet de déterminer efficacement l'évolution globale du système, est avare en explications sur le comportement de chaque entité de la simulation. Le modèle centré agent pose toutefois des problèmes de génie logiciel. Premièrement, le code décrivant les actions de l'agent est mélangé au code du moteur de comportement. De ce fait, il n'est pas facile d'ajouter ou de retirer une action aux capacités de l'agent : pour ce faire il est nécessaire d'intervenir dans le code de l'agent. Ensuite, le code correspondant à une action possible dans l'environnement est dupliqué entre les agents capables de l'effectuer, ce qui rend délicat la modification, l'ajout ou le retrait d'une action. Une intervention sur le code de l'agent ne peut pas s'effectuer *à chaud* : la modification effectuée ne sera prise en compte qu'après le redémarrage de la simulation. Enfin, le code des interactions n'est pas facilement réutilisable d'une simulation à une autre.

Afin de résoudre ces différents problèmes, nous souhaitons mettre au point un modèle modulaire dans lequel il est possible d'ajouter et de retirer des interactions, aussi bien lors de la modélisation de la simulation que durant l'exécution de celle-ci. Le modèle que nous proposons réifie le concept d'interaction, contrairement au modèle centré agent, dans lequel l'interaction est une conséquence émergeant du comportement des agents, un résultat que l'on observe. Nous prônons pour cela le principe de la séparation des aspects déclaratifs et procéduraux. Nous considérons que les interactions sont une donnée du problème (aspect déclaratif), au même titre que les agents, leurs propriétés et leurs positions. Ces données doivent être traitées par un moteur de comportement *générique* (aspect procédural) capable d'intégrer *à chaud* l'ajout, le retrait ou la modification d'interactions. L'interaction doit être

vue comme une recette, et le moteur de comportement comme la machine capable d'exécuter cette recette.

Nous proposons un modèle modulaire séparant clairement l'environnement, les interactions, les agents et leur moteur de comportement. Ce modèle ne permet pas de créer des simulations que l'on ne pourrait pas créer avec un modèle classique centré agent. Il se justifie avant tout par des préoccupations d'analyse et de génie logiciel. Nous allons maintenant étudier la représentation de la connaissance que nous avons choisie et le modèle centré interaction.

3.1 Le modèle

3.1.1 Le problème de la représentation des informations

Une des premières questions qui se pose lors de la conception d'un programme est celle de la représentation de la connaissance, en particulier lorsqu'il s'agit d'intelligence artificielle. Un agent qui évolue dans un monde et qui doit raisonner pour agir dans celui-ci doit être en mesure de représenter son environnement[17]. Parmi toutes les informations il faut concentrer la représentation sur celles qui sont utiles à l'agent dans la simulation. Par exemple si l'on souhaite réaliser un robot qui joue au football, il est nécessaire de connaître la position des joueurs de son équipe et de l'équipe adverse, la position des buts, celle du ballon, par contre il n'est pas utile qu'il connaisse la matière dans laquelle sont faits les poteaux de buts, le nombre de brins d'herbe composant la pelouse ou le nom de sponsors qui entourent le terrain. Même si ces constatations nous paraissent évidentes, elles ne le sont pas pour un programme informatique lorsqu'il est submergé par toutes les informations qui composent l'environnement. Ces informations doivent donc être judicieusement triées afin de ne conserver que celles qui sont pertinentes dans la simulation.

Une fois les informations triées il est nécessaire de trouver une manière de les représenter. Cette représentation doit permettre de stocker les informations mais aussi de les manipuler facilement : on ne peut donc pas se permettre de les stocker en langage naturel qui sera difficile (voire impossible) à manipuler. La connaissance doit être représentée formellement. Pour un agent embarqué dans un robot, une difficulté supplémentaire se situe au niveau de l'extraction d'informations à partir des images (reconnaissance de forme...). Ce problème ne se pose pas pour les agents logiciels qui évoluent dans un environnement formel : si un objet se trouve dans le champ de vision de l'agent, l'information perçue par le capteur contient déjà le nom de l'objet et ses propriétés, sans passer par une couche de reconnaissance de forme.

Les connaissances sont stockées par l'agent pour former une représentation de l'environnement qui l'entoure. Cette représentation constitue une mémoire : elle permet à l'agent d'accéder facilement et rapidement aux informations dont il a besoin, même s'il s'agit d'informations qu'il ne peut pas percevoir actuellement. Ces informations lui servent à faire des calculs afin d'effectuer un déplacement ou de résoudre un problème qui l'empêche d'atteindre son but.

[17] Ceci n'est pas toujours vrai pour les agents réactifs car ils ne raisonnent pas sur la totalité du monde.

3.1.2 La représentation choisie

Nous avons vu en 2.2 que le fonctionnement d'un agent consistait à traiter des informations (qui peuvent être très nombreuses) afin de décider de ses actions. Ces informations sont codées formellement pour que l'agent puisse les manipuler. Il y a diverses informations à représenter : la topologie de l'environnement, la position des agents dans l'environnement, les agents eux-mêmes et les règles qui permettent d'agir dans le monde, que nous appelons interactions. Pour représenter tout cela nous utilisons un modèle centré interaction, illustré en figure 3.1. Chaque pièce du puzzle représente un agent, tandis que les découpes des pièces représentent les interactions que les agents peuvent subir ou effectuer. L'agent de gauche est un bûcheron : il peut effectuer *ouvrir*, *couper* et peut subir *écraser*. L'agent de droite est un arbre : il peut subir *couper*, *brûler*, *casser* et peut effectuer l'interaction *écraser*. Ici deux interactions peuvent se produire entre les agents : *couper* ou *écraser*. En effet, pour qu'une interaction soit déclenchable, il faut à la fois un agent capable de l'effectuer, et un agent pouvant la subir. C'est le moteur de comportement des agents qui décidera de déclencher une interaction déclenchable si cela permet d'avancer dans la résolution de ses buts. Il apparaît une limitation évidente de ce modèle : il n'est pas possible d'effectuer une interaction entre plus de deux agents. Le modèle en tant que tel ne suppose aucune orientation sur la nature du moteur de comportement qui permettra de décider du déclenchement des interactions. Il est possible de l'utiliser conjointement à un moteur réactif comme à un moteur cognitif, ce qui lui confère une certaine généricité. D'ailleurs, au sein de l'équipe SMAC, le modèle est utilisé avec un moteur réactif dans le projet SimuLE et avec un moteur cognitif dans le projet CoCoA, sujet de cette thèse. Ce modèle centré interaction, et la dynamique peut-subir/peut-effectuer ont été publiés dans [DMR05a].

Exemple de description à l'aide du modèle centré interaction :

La réaction chimique d'oxydoréduction met en oeuvre deux composants chimiques entre lesquels ils se produit un échange d'électrons. Considérons l'exemple de la réduction du cuivre et de l'oxydation du zinc. Nous avons deux réactions (interactions) : l'oxydation et la réduction, ainsi que deux composants (agents) en présence : l'ion cuivre Cu^{2+}, qui peut effectuer une oxydation et subir une réduction, et le zinc Zn^0 qui peut effectuer une réduction et subir une oxydation. Le modèle centré interaction permet de décrire la réaction :

	cuivre	zinc
peut-effectuer	oxydation	réduction
peut-subir	réduction	oxydation

Cette description permet d'exprimer la dynamique :
- de la réaction d'oxydation : $Zn^0 \rightarrow Zn^{2+}$
- de la réaction de réduction : $Cu^{2+} \rightarrow Cu^0$
- de la réaction d'oxydoréduction : $Zn^0 + Cu^{2+} \rightarrow Zn^{2+} + Cu^0$

Les deux réactions (oxydation et réduction) doivent se produire simultanément, si elles se produisent. Il est du ressort du moteur de comportement associé de gérer ces aspects : le rôle du modèle centré interaction se limite à décrire les interactions qui peuvent se produire entre les agents.

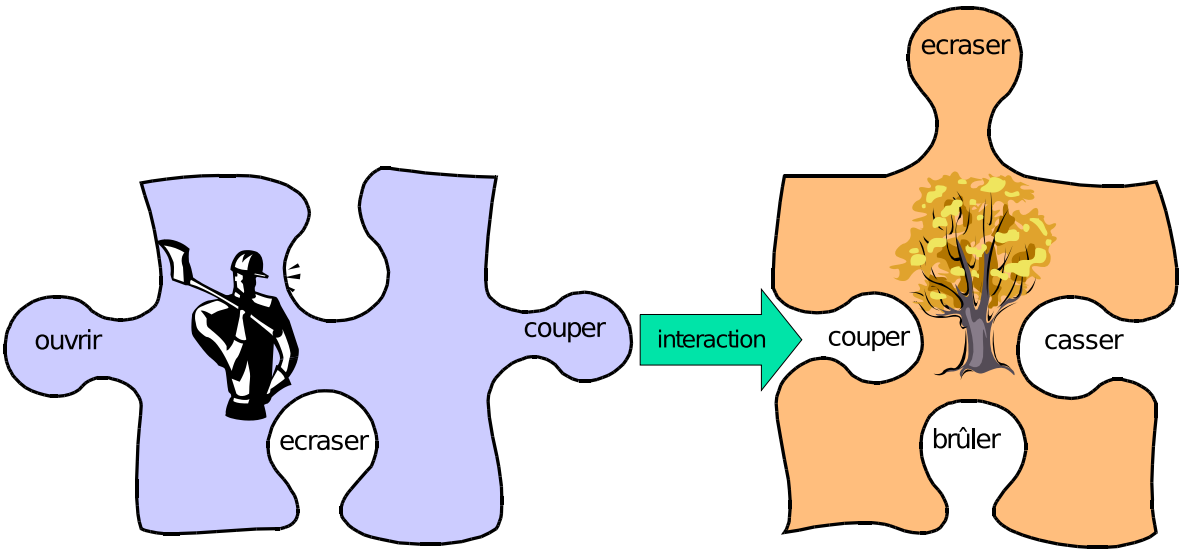


FIG. 3.1 – Le modèle d'interaction permet de décrire la dynamique d'une simulation : ici, deux agents sont en présence. Un bûcheron, qui sait effectuer les interactions *ouvrir*, *couper* et peut subir *écraser*, ainsi qu'une arbre qui peut subir *couper*, *brûler*, *casser* et peut effectuer l'interaction *écraser*. On voit rapidement que lors d'une simulation, deux interactions (une seule est illustrée ici) peuvent se déclencher : *couper* et *écraser*.

3.1.3 L'aspect génie logiciel

Les interactions représentent la connaissance relative aux actions qu'un agent peut effectuer pour agir sur les autres agents dans l'environnement. Cette connaissance est indépendante d'un contexte de simulation : l'interaction *ouvrir* exprime la même action (faire passer un objet ouvrable de l'état fermé à l'état ouvert) quelle que soit la simulation dans laquelle elle est utilisée. De plus, lors de la réalisation de simulations, il est souvent nécessaire de modifier le modèle afin de l'étendre ou de le rendre plus fidèle au phénomène que l'on souhaite simuler. Il est donc primordial que la connaissance relative aux interactions soit facilement récupérable dans une simulation et facilement réintégrable dans une autre. Le modèle centré agent ne répond pas à ces contraintes : le code des interactions se trouvant dilué dans le code des agents, il est à la fois peu aisé d'extraire la connaissance relative aux interactions et délicat d'intégrer cette connaissance dans une autre simulation. Ces interventions demandent bien souvent la modification du code des agents.

Voici par exemple le code d'une simulation de termites (évoquée en 2.3.3) dans la plateforme NetLogo :

```
to go
  search-for-chip
  find-new-pile
  put-down-chip
end
```

```
to search-for-chip
  ifelse pcolor = yellow
    [ set pcolor black
      set color orange
      fd 20 ]
    [ wiggle
      search-for-chip ]
end


to find-new-pile
  if pcolor != yellow
   [wiggle
    find-new-pile]
end


to put-down-chip
  ifelse pcolor = black
   [set pcolor yellow
    set color white
    get-away]
   [rt random-float 360
    fd 1
    put-down-chip]
end


to get-away
  rt random-float 360
  fd 20
  if pcolor != black
    [get-away]
end


to wiggle
  fd 1
  rt random-float 50 - random-float 50
end
```

Ce code mélange les aspects liés aux actions des agents (se déplacer, ramasser du bois) et le code lié à l'implémentation dans cette simulation : ainsi, ramasser un morceau de bois correspond à changer la couleur de la case sur laquelle le termite se trouve ainsi que la couleur du termite lui-même. Ce mélange rend assez délicat la modification d'une action de l'agent, ne serait-ce parce qu'il faut localiser la portion de code à modifier. De plus, il est impossible de

réutiliser le code d'une action dans une autre simulation. Par exemple, si l'on considère l'action *ramasser*, on n'identifie pas à première vue la partie du code concernée, d'une part parce que ce code est noyé dans la procédure *search-for-chip* mais surtout parce que l'implémentation de cette action est spécifique à cette simulation (changement de couleur).

Pour résoudre ces différents problèmes, la solution consiste à reconsidérer la nature des actions : elles ne doivent pas être vues comme du code (aspect procédural) mais comme une donnée du problème (aspect déclaratif). Notre modèle sépare donc les aspects déclaratifs et les aspects procéduraux de la manière suivante : la connaissance (aspect déclaratif) inclut non seulement l'environnement, les agents et leurs propriétés, mais aussi les interactions, celles-ci exprimant ce qu'il est possible de faire dans le monde. Le moteur de comportement (aspect procédural) est quant à lui générique dans la mesure où il est capable d'utiliser n'importe quelle interaction à partir du moment où elle répond au formalisme. Ceci permet de réutiliser les agents d'une simulation à une autre. Les interactions sont quant à elles abstraites par rapport aux simulations. Ainsi, notre modèle permet facilement l'ajout d'interactions : il suffit de créer l' interaction si elle n'existe pas ou de la recopier à partir d'une autre simulation, puis d'en ajouter le nom dans la liste des propriétés *peut-effectuer* et *peut-subir* des agents *source* et *cible* de cette interaction. Immédiatement, les deux agents sont susceptibles d'utiliser cette nouvelle connaissance pour entrer en interaction. Ce point de vue est également défendu dans [RD00] même si le sens donné aux interactions n'est pas le même qu'ici.

Ce modèle centré interaction n'est pas lié à un moteur de comportement. Il est utilisé dans l'équipe SMAC dans différentes applications : il est associé à un moteur réactif dans le projet SimuLE[18] et associé ici à un moteur cognitif dans le projet CoCoA (Collaborative Cognitive Agents).

3.1.4 Adaptation de la méthodologie IODA

Habituellement, dans un modèle de simulation *centré agent*, la conception se concentre essentiellement sur les agents afin de les doter du comportement souhaité dans la simulation. La partie du code servant à décrire les interactions entre les agents est diluée dans le code des agents et il est difficile d'intervenir sur une interaction si elle est partagée par plusieurs agents. Nous défendons un modèle centré interaction dans lequel les interactions sont clairement séparées des agents, ce qui permet à la fois de pouvoir modifier facilement une interaction, de la rajouter au comportement d'un agent, mais aussi de la réutiliser dans d'autres simulations. Pour réaliser cette analyse *centrée interaction*, l'équipe SMAC propose la méthodologie IODA (Interaction Oriented Design of Agent simulations) [MP06] [MP05] qui sépare la conception des agents et celle des interactions. Cette méthodologie est initialement prévue pour être utilisée conjointement à un moteur réactif dans l'application *SimuLE*. Elle peut toutefois être utilisée pour concevoir des simulations utilisant un moteur cognitif moyennant quelques adaptations. La méthodologie IODA se décompose en trois phases : l'identification des interactions, l'identification des caractéristiques des agents et l'identification de la dynamique de la simulation.

[18] Simulation Large Echelle

Identification des interactions

Cette première phase consiste à identifier toutes les interactions qui peuvent se produire dans la simulation. Les interactions sont placées dans une matrice afin d'en déterminer les sources et les cibles potentielles. Un exemple est donné par le tableau 3.2. A l'intersection de la ligne représentant l'agent source a_s et de la colonne représentant l'agent cible a_c on inscrit la ou les interactions i telles que la situation a_s *déclenche l'interaction* i *sur la cible* a_c est souhaitable dans la simulation. Le fait de déclarer qu'une interaction est « souhaitable » ne signifie pas qu'elle se produira obligatoirement dans la simulation : l'interaction i sera potentiellement déclenchable par l'agent a_s sur l'agent a_c si le moteur de comportement du premier le juge opportun pour avancer dans la résolution de ses buts. Il est possible de réaliser ce raisonnement sur des classes d'agents au lieu de réagir individuellement sur chacun des agents de la simulation.

L'agent *lapin* peut-être la cible de l'interaction *manger* (par l'agent *humain*), et peut aussi en être l'agent source (sur la cible *carotte*). On s'aperçoit que par effet de bord, il est possible dans la simulation qu'un agent de la classe *lapin* déclenche l'interaction manger sur un agent de la classe *lapin*, y compris lui-même. Il sera donc important au moment du codage de la simulation de remédier aux interactions rendues possibles implicitement.

Sources / Cibles	humain	lapin	carotte	porte
humain				
lapin	manger			
carotte	manger	manger		
porte	ouvrir forcer			

FIG. 3.2 – Phase 1 de IODA

Cette première phase permet en outre d'identifier le type de problème auquel on est confronté en fonction de la manière dont est rempli le tableau, grâce à la taxonomie définie dans [MP07]. Il est par exemple clair que si le tableau met en évidence des interactions nécessitant plusieurs agents sources ou plusieurs agents cibles, on sort du cadre de cette thèse.

Identification des caractéristiques des agents

Cette seconde phase, lors de l'utilisation conjointe à un modèle réactif, sert à identifier les priorités et gardes des interactions. Ceci ne nous étant d'aucune utilité dans le cas de l'utilisation avec un moteur cognitif, nous allons utiliser cette phase à une autre fin : l'identification des propriétés des agents. Dans le code des interactions, des propriétés apparaissent (ex : inventaire, force, couleur ...). Ces propriétés, qui sont relatives à l'agent source de l'interaction ou à l'agent cible, sont accédées « en lecture » dans la partie *conditions* de l'interaction et « en lecture/écriture » dans la partie *conclusion* de l'interaction. Il est primordial que les agents possèdent les propriétés correspondantes aux interactions dont ils peuvent être l'acteur ou la cible, sous peine de ne pouvoir déterminer le résultat de l'exécution d'une interaction.

Agents / Propriétés	humain	lapin	carotte	porte
peut-effectuer	deverrouiller forcer manger	manger		
peut-subir		manger	manger	deverrouiller
propriétés obligatoires	energy inventory	energy	energy	isLocked key
propriétés facultatives	size name			color

TAB. 3.1 – Phase 2 de IODA pour des agents cognitifs

On réalise un tableau dans lequel on fait apparaître pour chaque agent la liste des interactions qu'il peut subir ou qu'il peut effectuer. On recherche alors dans ces interactions les propriétés que doivent avoir les agents. Les propriétés accédées en lecture comme en écriture doivent être ajoutées à la liste des propriétés de l'agent. Pour une interaction que l'agent peut-effectuer, on ajoutera uniquement les propriétés qui sont préfixées par *actor* et pour les interactions que l'agent peut-subir, on ajoutera uniquement les propriétés préfixées par *target*. On peut ajouter à ce tableau des propriétés facultatives qui ne sont pas nécessaires au fonctionnement du modèle mais qui peuvent apporter des informations complémentaires dans la simulation.

Le tableau 3.1 donne un exemple de réalisation de la phase 2 de la méthodologie IODA pour une simulation utilisant les agents cognitifs du tableau 3.2. Cette phase permet de répertorier les propriétés à donner aux agents afin d'assurer un bon fonctionnement de la simulation. Le code des interactions utilisées est donné ci-après :

```
manger
cond -
garde distance(actor, target) < 1
action actor.energy += target.energy
target.destroy()

ouvrir
cond target.isLocked = true
actor.inventory.contains(target.key)
garde distance(actor, target) < 1
action target.isLocked = false

forcer
cond target.isLocked = true
garde distance(actor, target) < 1
action target.isLocked = false
```

Identification de la dynamique de la simulation

Cette dernière phase concerne l'évolution des caractéristiques des agents au cours du temps, en particulier l'évolution de la liste des interactions qu'ils peuvent réaliser ou subir au fur et à mesure de l'exécution des interactions dans la simulation. Cette dernière partie n'est nécessaire que lorsque le modèle est utilisé conjointement à un moteur réactif, ce qui est le cas dans l'application *SimuLE*. Nous ne la détaillerons pas ici.

3.2 Les interactions

Parmi toutes les connaissances qu'il est nécessaire de représenter pour réaliser une simulation par agents cognitifs, les plus importantes sont certainement celles qui expriment les opérations que l'on peut réaliser dans l'environnement. Ces opérations servent à changer l'état des éléments (y compris les autres agents) qui composent le monde, à l'instar des actions que l'on effectue tous les jours comme ouvrir une porte, prendre un objet ou appuyer sur un bouton. Une action peut être vue comme l'application d'une *méthode* particulière dans le but d'obtenir une *conséquence* sur l'état du monde. La représentation d'une action doit donc appréhender ces deux aspects : d'une part la méthode qui décrit la manière d'effectuer l'interaction, et d'autre part la conséquence de l'action, que l'on verra plutôt comme sa finalité. La représentation choisie doit bien sûre être formelle afin de pouvoir être manipulée par les agents.

Nous appelons *interaction* la représentation formelle d'une action. Les interactions (cf figure 3.3) ressemblent à des règles STRIPS [FN71]. Une interaction est un triplet $I = \{C, G, A\}$ où C (la partie *conditions*) définit les conditions à satisfaire pour pouvoir déclencher l'interaction, G (la partie *garde*) est une condition particulière relative à l'aspect situé de nos simulations et A (la partie *actions*) définit les conséquences de l'exécution de l'interaction. La méthode est décrite par C et G tandis que la finalité de l'interaction est exprimée par A. La figure 3.4 donne un exemple d'interaction.

```
<interaction>          ::=  <name>, <condition>*, <guard>, <action>*
<name>                 ::=  Symbol
<guard>                ::=  <d> <op> Integer
<d>                    ::=  distance between actor and target
<op>                   ::=  = | > | < | ≤ | ≥
<condition>            ::=  <test-property> | <predicate-primitive>(args)
<test-property>        ::=  { actor | target }.<property-name> <op> Value
<predicate-primitive>  ::=  <primitive>
<action>               ::=  <affect-property> | <primitive>
<affect-property>      ::=  { actor | target }.<property-name> = Value
<primitive>            ::=  { actor | target }.<primitive-name>(args)
<primitive-name>       ::=  Symbol
<property-name>        ::=  Symbol
```

FIG. 3.3 – Une interaction est composée d'un nom, d'un ensemble de conditions, d'une garde et d'actions.

ouvrir	
CONDITIONS :	acteur.possede(cible.clef) = vrai
	cible.ouverte = faux
GARDE :	distance(acteur, cible) < 1
ACTIONS :	cible.ouverte = vrai

FIG. 3.4 – Exemple d'interaction : *ouvrir*. Pour pouvoir exécuter cette interaction il faut que l'acteur dispose de la clef de la cible de l'interaction et que la cible ne soit pas déjà ouverte. De plus la garde exprime le fait que l'agent doive se trouve à proximité de la cible. Le but de l'interaction est que la cible soit ouverte après l'exécution. Cette interaction est générique. Elle peut être appliquée par un *acteur* sur une *cible* sous réserve que ceux-ci satisfassent au modèle général : l'*acteur* doit pouvoir effectuer l'interaction *ouvrir* et la *cible* doit pouvoir la subir.

Les primitives sont des actions élémentaires qui permettent d'agir réellement dans l'environnement. Sans ces primitives (et en particulier sans leur implémentation), l'exécution du plan serait impossible et celui-ci resterait purement conceptuel. Voici les primitives principalement rencontrées dans les interactions :

- **add :** permet d'ajouter une valeur à une propriété ou un agent à une collection
- **destroy :** détruit un agent et le supprime de l'environnement
- **move away :** permet de s'éloigner d'une cible (un pas)
- **move to :** permet de s'approcher d'une cible (un pas)
- **put :** permet de déposer un agent à la position actuelle de l'agent
- **remove :** permet d'enlever un agent de l'environnement (sans le détruire)
- **substract** permet de soustraire une valeur à une propriété ou un agent à une collection
- **= :** permet d'affecter une valeur à une propriété

Les interactions constituent une représentation générique des actions. Dans ce formalisme, rien ne lie une interaction à un ou plusieurs agents en particulier : les deux agents apparaissant dans le code de l'interaction sont *actor* et *target*. Le premier désigne l'agent qui effectuera l'interaction tandis que le second fait référence à l'agent qui la subira. Ainsi, l'interaction *ouvrir* peut aussi bien s'appliquer à une porte qu'à un coffre, ou en général tout objet « ouvrable[19] » de la simulation, voire même d'une autre simulation.

Nous allons maintenant détailler les trois parties composant une interaction.

Les conditions

Les conditions sont des expressions booléennes portant sur les propriétés des agents. Si l'évaluation de l'expression instanciée avec la source et la cible de l'interaction est vraie, la condition représentée par l'expression est satisfaite. Si par contre l'évaluation de l'expression est fausse, la condition n'est pas satisfaite. Les figures 3.5 et 3.6 donnent deux exemples de conditions.

Les gardes

[19] c'est-à-dire un agent qui peut-subir l'interaction ouvrir.

```
actor.inventory.contains(knife)
```

FIG. 3.5 – exemple de *condition* : cette condition porte sur l'acteur. Il doit posséder un couteau pour pouvoir déclencher cette interaction. Il peut s'agir de l'interaction couper ou tuer...

```
target.powered = false
```

FIG. 3.6 – second exemple de *condition* : cette condition porte cette fois sur la cible. Elle ne doit pas être sous tension pour que l'interaction soit déclenchée. Il peut s'agir par exemple de changer une ampoule.

Les gardes sont des conditions particulières relatives aux aspects situés. Elles permettent de définir une condition sur la distance séparant l'objet de la cible pour pouvoir effectuer l'interaction (figure 3.7). La distance est mesurée en termes de « pas à effectuer » c'est-à-dire le nombre d'arcs qu'il faut franchir dans le graphe sous-jacent pour atteindre la cible. Dans le cas d'un environnement 2D constitué d'une grille (que nous utilisons dans nos simulations) il s'agit du nombre de cases à franchir.

Dans certains cas l'acteur doit être contre la cible, par exemple pour appuyer sur un bouton, tandis que dans d'autres cas il doit se trouver à une distance minimale de la cible, par exemple tirer au fusil sur une proie, il ne faut pas être trop près pour ne pas qu'elle s'échappe. Dans ce dernier cas, on pourra avoir deux gardes de distance : une première qui nous oblige à nous tenir à distance de la cible, et une seconde qui pousse à se rapprocher de l'objet pour être à portée de tir. Sans cette garde de distance, un agent pourrait déclencher une interaction dès qu'il a résolu toutes les conditions de celle-ci. Il est alors fort probable qu'il se trouve loin de la cible de l'interaction, car la résolution des conditions l'a amené à se déplacer. Le rôle de la garde de distance (quand elle existe) est donc de forcer l'agent à revenir près de la cible pour pouvoir exécuter l'interaction.

Nous verrons que l'interaction qui permet de résoudre une garde de distance est l'interaction "aller" dont l'exécution modifie la position géographique de l'agent. Les gardes sont donc génératrices de déplacements : elles sont indissociables du caractère situé des simulations.

Toutes les conditions de l'interaction, en particulier la garde de distance, doivent être vraies pour pouvoir exécuter l'interaction. La plupart du temps, la garde stipule qu'il faut se trouver à côté de la cible de l'interaction (i.e. à une distance inférieure à 1). Lors de l'exécution, la résolution des autres conditions peut amener l'agent à se déplacer, ce qui invalide la garde de distance. Le déplacement vers la cible doit donc être la dernière action à effectuer par l'agent avant de tenter de déclencher l'interaction, afin d'éviter un déplacement initial inutile vers la cible.

Si nous prenons l'exemple d'un distributeur de boisson nécessitant une pièce, il est évident qu'il est inutile de s'approcher de la machine tant que l'on ne possède pas de pièce : en effet, trouver une pièce va nous amener à nous déplacer dans l'environnement et par conséquent nous éloigner de la machine.

```
distance(acteur, cible) < 1
```

FIG. 3.7 – exemple de *garde* : l'acteur doit être à proximité immédiate de la cible pour déclencher l'interaction. C'est le cas pour beaucoup d'interactions : ouvrir, appuyer, prendre...

Les actions

La partie action est un ensemble de propositions sur les propriétés des agents qui décrivent l'état atteint une fois l'interaction exécutée. Elle permet de décrire les conséquences de l'exécution de l'interaction, les effets produits sur le monde. On peut d'ailleurs voir ces conséquences comme la *finalité* de l'interaction puisque l'on choisira d'exécuter une interaction pour obtenir le résultat qu'elle produit. Certaines des propositions de la partie action peuvent en revanche être considérées comme des effets de bord : l'interaction *manger* possède par exemple deux expressions dans sa partie action :

1. $actor.energy = actor.energy + target.energy$
2. $destroy(target)$

La première expression est la finalité de l'interaction *manger* : augmenter l'énergie de l'agent qui effectue l'interaction. La seconde expression, quant à elle, exprime un effet de bord : la cible de l'interaction manger est détruite. Toutefois, un agent qui souhaite faire disparaître un autre agent de l'environnement peut le manger (si celui-ci peut subir cette interaction). Dans ce cas, c'est la première expression qui constitue un effet de bord, puisque ce n'est pas pour obtenir cette conséquence que l'interaction a été exécutée.

La partie action permet au moteur de comportement de choisir parmi les interactions celle qui est adaptée au problème qu'il doit résoudre. Lors de l'implémentation d'une interaction, il est impératif de s'assurer que celle-ci réalise effectivement la partie action lorsqu'elle est déclenchée.

La figure 3.8 donne un exemple d'utilisation d'une primitive dans la partie action d'une interaction. Celle-ci exprime que l'agent *target* sera ajouté à l'inventaire de l'agent *cible* lors de l'exécution de l'interaction. L'inventaire est une propriété particulière de l'agent, de type collection. Il permet de représenter les agents (généralement inanimés) en possession de l'agent. Cette action apparaît dans l'interaction *prendre*, où elle sera associée à l'action *remove(target)* qui permet d'enlever la cible de l'environnement. Ainsi, l'interaction *prendre* peut être utilisée à deux fins : ajouter un objet à l'inventaire, ou en débarrasser l'environnement.

```
add(actor.inventory, target)
```

FIG. 3.8 – exemple d'*action* utilisant une primitive

3.2.1 Discussion sur l'écriture des interactions

L'exécution d'une interaction n'est possible[20] que si toutes les conditions et gardes de l'interaction sont vraies. C'est sur ce principe que se base le moteur de planification de l'agent : s'il doit exécuter une interaction et que les conditions de celle-ci ne sont pas remplies, l'agent a alors pour but de les rendre vraies. Nous verrons dans la partie 4.2 le mécanisme permettant de générer un plan à partir de ce principe.

[20] Ce point est vérifié par l'environnement lors de l'exécution de l'interaction afin de l'interdire si les conditions ne sont pas remplies et que l'agent essaie de « tricher »

Il est donc important de ne pas se méprendre sur la nature des conditions : habituellement en programmation, la condition d'une structure « if then » exprime le fait qu'on effectue une action si la condition est vraie et qu'on ne l'effectue pas dans le cas contraire (ou que l'on effectue une action alternative dans le cas où il existe une partie « else »). Cette condition est un test, et une des deux possibilités sera exécutée en fonction du résultat de ce test. Nos conditions sont syntaxiquement semblables : il s'agit aussi d'un test dont le résultat est un booléen. En revanche, la sémantique de la condition n'est pas la même : elle ne permet pas de décider entre deux possibilités (exécuter l'interaction ou ne pas l'exécuter) mais exprime une condition qu'il est impératif de remplir pour pouvoir exécuter l'interaction. Il ne s'agit pas de « si cette *condition* est vraie alors j'exécute cette *interaction* » mais de : « pour exécuter cette *interaction* je dois remplir cette *condition* ». La condition d'une interaction est un prérequis que l'agent doit rendre vrai s'il souhaite exécuter l'interaction. Dès lors, une interaction ne peut pas exprimer un scénario tel que « si le feu est vert alors je passe ». Dans un tel cas de figure, le moteur de comportement de l'agent considérerait la proposition *le feu est vert* comme un nouveau but et chercherait à le résoudre : si l'agent connaît les interactions adaptées et possède les outils nécessaires, il s'empressera de démonter le feu pour le forcer à passer au vert !

3.3 Les agents

Les entités qui interagissent dans les simulations sont représentées par des agents. Elles peuvent être de deux types : les entités actives et les entités passives. Les premières ont un comportement propre : ce sont celles qui déclencheront des interactions sur les autres entités. Nous les représenterons dans les simulations par des agents animés. Les secondes ne feront en revanche que subir les interactions et seront représentées par des agents inanimés. Nous allons voir que les entités actives et inactives possèdent beaucoup de points communs. Ceci justifie le fait que, mis à part l'environnement, tout soit représenté par des agents, dans un souci d'unification et de simplification du modèle : les entités inactives font partie intégrante de la dynamique de la simulation.

3.3.1 Les agents inanimés

Les agents inanimés sont les entités inertes de la simulation. Ils servent à représenter les entités qui ne sont pas dotées d'un comportement propre, c'est-à-dire qui ne participent pas activement à la simulation. Ces agents possèdent un ensemble de propriétés qui décrivent leurs caractéristiques, comme leur taille, leur couleur, ou toute autre information pertinente pour une simulation 3.9. En particulier, l'une de leurs propriétés est la propriété *peut-subir* qui contient la liste des interactions dont l'agent peut être la cible. Dans une simulation, les objets sont représentés par des agents inanimés.

3.3.2 Les agents animés

Les agents animés sont une extension des agents inanimés. Ils possèdent une propriété supplémentaire qui est la propriété *peut-effectuer* qui liste les interactions que l'agent dont l'agent connaît le code et qu'il peut déclencher sur un autre agent. En plus des propriétés,

```
<agent>          ::=  <passive-agent> | <active-agent>
<passive-agent>  ::=  { ("name", Symbol),
                      ("can-suffer", <interaction>*),
                      <property>*}
<active-agent>   ::=  <passive-agent> ∪
                      { ("can-perform", <interaction>*),
                      ("goals", goal*),
                      ("memory", (degraded) environment),
                      ("engine", engine)}
<interaction>    ::=  Symbol
<property>       ::=  Symbol, value
```

FIG. 3.9 – Définition d'un agent.

les agents animés ont un moteur de comportement et une mémoire (figure 3.9). Le moteur de comportement des agents animés (détaillé dans 4.2) qui leur permet de choisir les actions qu'ils vont effectuer pour résoudre leurs buts, en fonction des informations présentes dans leur mémoire.

On remarquera que la définition d'un agent ne comporte pas le code des interactions, mais seulement leurs noms. L'agent animé se réfèrera au code de l'interaction correspondante dans la simulation.

Les interactions spécifiques

Pour certains agents cibles, une interaction peut avoir un fonctionnement particulier. C'est le cas des *LockableDoor* (portes verrouillables) dont l'interaction *open* est plus restrictive que pour les agents *Door* classiques : elle impose que la cible soit déverrouillée, en plus des autres conditions. La sémantique de l'interaction est la même : le but est bien de faire passer un agent « open-able[21] »du statut fermé ($door.isOpen = false$) à ouvert ($door.isOpen = true$). Le mécanisme des interactions spécifiques permet de gérer cela. Une propriété particulière de l'agent permet de spécialiser le code d'une interaction : cette spécialisation consiste en l'ajout d'une ou plusieurs conditions, garde ou actions qu'il s'agira de prendre en compte pour interagir avec cette cible.

Lorsqu'un agent doit faire passer un agent open-able du statut fermé au statut ouvert, son moteur de raisonnement lui indique d'utiliser l'interaction *open*. Dans le cas d'une cible possédant une interaction *open* spécifique, l'agent utilisera le code fourni par la cible, mais il conservera néanmoins dans ses connaissances le code original de l'interaction *open*, qu'il utilisera sur tous les autres agents open-able : la spécificité ajoutée par une cible n'est valable que pour cette dernière. Elle est totalement transparente pour l'agent : son raisonnement est identique, qu'une interaction spécifique soit disponible ou non. Il intégrera toutefois dans son plan la spécificité éventuelle ajoutée par la cible.

Le tableau suivant donne le code de trois interactions. A gauche, l'interaction *open* générique permettant d'ouvrir tous les agents open-able. Au centre, le code spécifique ajouté par

[21] c'est-à-dire qui peut subir l'interaction *open*

```
<environnement>  ::=  place*, <passage>*
<passage>        ::=  {place, place, <condition>}
<condition>      ::=  même définition que dans une interaction
```

FIG. 3.10 – Définition de l'environnement.

les agents lock-able. Enfin, à droite, l'interaction *unlock* connue de l'agent, qu'il utilisera le cas échéant pour résoudre la condition spécifique.

open	open	unlock
target.isOpen = false		target.isUnlocked = false
	target.isUnlocked = true	actor.own(target.key)
distance(actor, target) < 1		distance(actor, target) < 1
target.isOpen = true		target.isUnlocked = true

3.3.3 Animé ou inanimé ?

Pour la plupart des agents, l'appartenance à la catégorie *agent animé* ou *agent inanimé* est sans équivoque. Un termite sera par exemple représenté par un agent animé puisqu'il doit avoir un comportement propre dans la simulation, tandis qu'un morceau de bois sera représenté par un agent inanimé puisqu'il représente un objet qui n'a pas de comportement. Le cas d'un arbre fruitier est plus problématique : à première vue il semble inanimé. Toutefois, si celui-ci produit périodiquement des fruits dans la simulation, il doit être représenté par un agent animé puisqu'il modifie son environnement. Il est important de bien comprendre que la différence entre agent animé et agent inanimé ne porte pas sur sa capacité à se déplacer.

3.4 L'environnement

Nous travaillons dans le contexte de la simulation située : la position géographique des agents (relativement les uns par rapport aux autres) est très importante. Il est donc impératif que les agents puissent appréhender les aspects géographiques de la simulation. Pour cela, les agents appartenant à une même simulation sont placés dans un environnement qui leur fournit des repères géographiques : un agent peut ainsi connaître la position relative des autres agents, afin de calculer la distance qui le sépare d'un agent ou un chemin permettant de se déplacer jusqu'à cet agent.

L'environnement repose sur un graphe (figure 3.10) : il est constitué de places (les nœuds du graphe) et de passages entre ces places (les arcs du graphe).

3.4.1 Les places

Les places sont des unités géographiques indivisibles qui peuvent contenir des agents. A l'intérieur de ces places, il n'existe aucune contrainte spatiale : nous le verrons, la distance entre deux agents qui se trouvent sur la même place est nulle. Selon la granularité de l'environnement de la simulation, une place peut représenter une ville (si l'environnement représente un

pays), une pièce (si l'environnement représente un bâtiment) ou une quelconque subdivision géographique de l'espace en fonction des besoins de la simulation.

3.4.2 Les passages

Deux places peuvent être reliées par un passage si l'on souhaite qu'un déplacement soit possible de la première vers la seconde. Un passage est unidirectionnel : le retour vers une place n'est possible que s'il existe un passage dans l'autre sens. Les passages sont matérialisés par des arcs dans le graphe représentant l'environnement. Un arc peut porter une condition, qui est de la même forme qu'une condition d'une interaction. Elle sera d'ailleurs résolue de la même manière par le chaînage : la condition portée par un arc doit être vraie pour qu'un agent puisse le franchir. Les conditions portent souvent sur l'agent qui se trouve sur une des places connectées ou sur l'agent qui veut traverser l'arc. Pour une simulation 2D classique, les passages n'existent qu'entre deux places adjacentes, toutefois il est possible de relier des places non-adjacentes par un passage (pour modéliser un téléporteur par exemple) : dans ce cas le graphe obtenu n'est pas planaire.

La figure 3.11 donne un exemple d'environnement de deux pièces séparées par une porte. La figure 3.12 donne une partie du graphe représentant cet environnement. Entre une place p et un mur (1) il n'existe pas d'arc sortant ni d'arc entrant. Entre deux places adjacentes d'une même pièce (3) les arcs portent la condition *true* qui exprime le fait qu'ils sont franchissables inconditionnellement. Entre une place p_1 et une place p_2 contenant un agent porte (2), la condition de l'arc (p_1, p_2) est $door.isOpen = true$. L'arc (p_2, p_1) comporte en revanche la condition *true* : seule l'entrée vers la place contenant l'agent porte est réglementée par une condition.

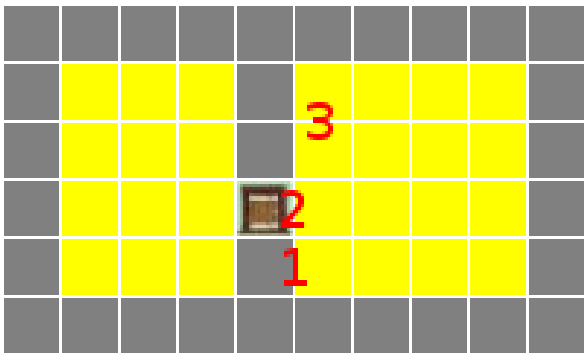


FIG. 3.11 – Environnement et conditions de franchissement.

3.4.3 La notion de distance

Pour réaliser des simulations situées, il est nécessaire de définir une notion de distance entre les agents. En effet, c'est sur cette notion que s'appuient les gardes qui permettent d'intégrer les déplacements dans la planification. Les agents n'ont pas connaissance de leur position absolue dans l'environnement (ce n'est pas nécessaire) : ils connaissent leur position relativement aux

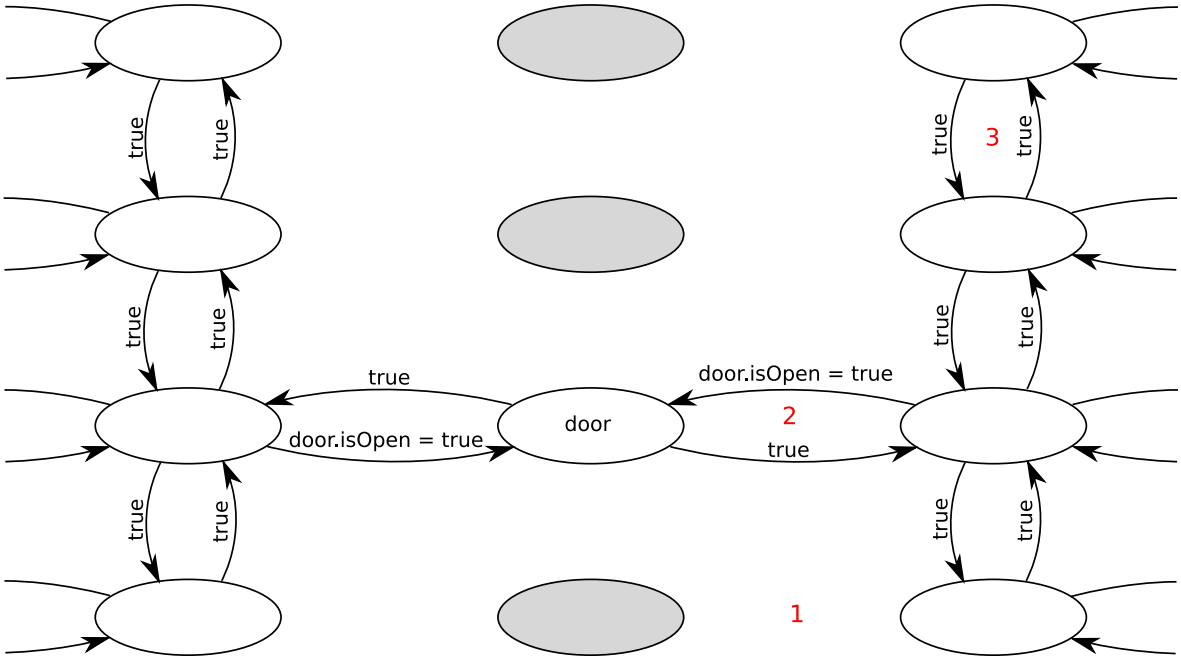


FIG. 3.12 – Une partie du graphe sous-jacent à l'environnement de la figure 3.11

autres agents. Pour calculer la distance qui sépare deux agents a et b, on considère le plus court chemin dans le graphe, c'est-à-dire le nombre minimal d'arcs que l'agent a devrait franchir pour se déplacer jusqu'à l'agent b. Notez que la distance entre a et b peut-être différente de la distance entre b et a puisque les arcs ne sont pas bidirectionnels : les chemins aller et retour ne passent pas forcément au même endroit.

3.4.4 Discussion sur la modélisation de l'environnement

Certaines particularités géographiques peuvent donner lieu à des représentations différentes, et il faut faire un choix de représentation en fonction de ce que l'on souhaite observer. C'est le cas en particulier pour la traversée de zones soumises à des conditions. Par exemple, si l'on souhaite représenter une rivière que l'agent ne peut traverser que s'il sait nager, on a le choix entre deux représentations (figure 3.13). Dans le premier cas (en haut), la rivière apparaît dans l'environnement en tant que place : l'agent qui traverse la rivière se trouve dans la rivière à un moment de la simulation. La condition qui requiert que l'agent sache nager se trouve sur l'arc qui permet d'entrer dans la rivière. Dans le second cas (en bas), la rivière est virtuelle car elle n'est pas représentée par une place. L'agent ne se trouvera donc à aucun moment dans la rivière mais la franchira virtuellement lors d'un pas de déplacement.

Conclusion

Le modèle centré interaction et la dualité peut-effectuer/peut-subir définissent un cadre dans lequel des agents peuvent entrer en interaction en fonction de leur caractéristiques : un agent ouvrable peut être ouvert par un agent qui sait ouvrir, un agent cassable peut être cassé

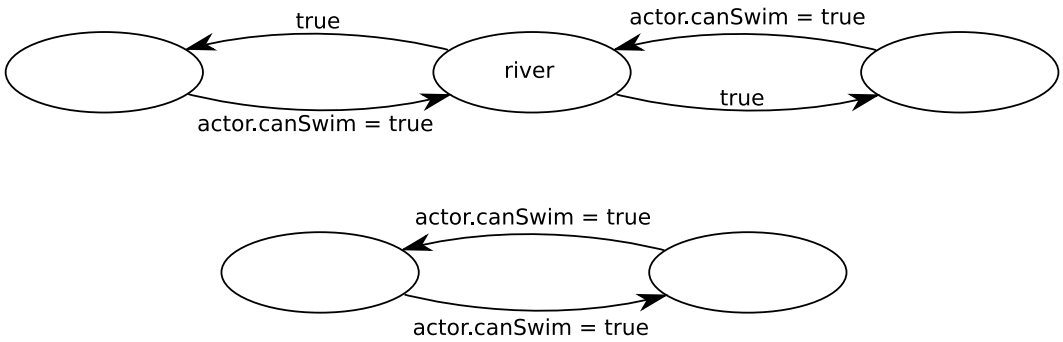


FIG. 3.13 – Représentations possibles d'une rivière : concrétisation ou non de l'agent.

par un agent qui a la capacité de casser. Plus généralement, ce cadre permet de décrire le fait que tout agent qui sait effectuer l'interaction peut la déclencher sur un agent capable de la subir. La séparation des aspects déclaratifs et procéduraux des simulations permettent de créer des simulations modulaires. On peut ainsi réutiliser les éléments d'une simulation, que ce soient des agents, des environnements ou des interactions. La méthodologie IODA guide le créateur d'une simulation dans la création de celle-ci. La représentation de la connaissance permet d'intégrer les déplacements dans les connaissances. Nous allons voir maintenant le moteur de comportement qui utilise ces données pour calculer le comportement d'un agent, que ce soit dans un contexte mono-agent, un contexte de concurrence ou un contexte collaboratif.

Chapitre 4

Le moteur de comportement

« Il savait ce qu'il devait faire. Bien sûr, c'était une tâche impossible. Mais il en avait l'habitude. Traîner un rat depuis la forêt jusqu'au terrier avait été impossible. Mais le traîner sur une petite distance ne l'était pas. Alors, c'est ce qu'il fallait faire. Ensuite, on se reposait et on le traînait encore un peu plus loin... Pour accomplir une tâche impossible, on la débitait en petits bouts de tâches simplement très difficiles, qu'on divisait ensuite en tâches horriblement pénibles, qu'on segmentait à leur tour en travaux délicats et ainsi de suite... »
T. Pratchett, Le grand livre des gnomes.

4.1 Introduction

Nous avons vu en 2.3 que les agents d'un système multi-agent pouvaient être confrontés à différents modes d'interaction : la concurrence, la compétition et la collaboration. Le moteur de comportements que je propose intègre deux de ces trois modes, à savoir la *concurrence* et la *collaboration*. Avant de détailler le fonctionnement du moteur décliné dans chacun de ces deux modes, nous allons en étudier le fonctionnement général en dehors de tout contexte.

4.2 Le cas d'un agent isolé

4.2.1 Les contraintes de simulation

Nous avons vu en 2.8 les différentes formes de simulation et leur domaine d'application. Au terme de la création d'un modèle du phénomène à étudier, on peut simuler ce phénomène. La plupart du temps, la simulation se fait grâce à des moyens informatiques, et elle se résume généralement à l'exécution d'un programme dont on peut analyser les résultats. Si l'on crée le modèle physique d'un futur ouvrage architectural (bâtiment, pont suspendu...) pour en étudier

la résistance face à des phénomènes physiques comme le vent ou les secousses sismiques, on s'attend bien sur à ce que le résultat de la simulation nous indique si l'ouvrage peut résister aux conditions que l'on a reproduites ou s'il a été détruit. Toutefois, cette connaissance ne suffit pas : on aimerait savoir le cas échéant quelle partie de l'ouvrage a cédé en premier, au bout de combien de temps, dans quelles conditions... Le déroulement d'une simulation (la succession d'états du système) est parfois aussi important que le résultat de la simulation (l'état final du système). C'est le cas de nos simulations par agents : nous ne pouvons nous contenter d'analyser de l'état final de la simulation, ou de savoir si un état peut-être atteint. Cela ne fournit aucune explication permettant de déterminer comment on en est arrivé à cet état, et c'est pourtant cette explication que l'on cherche à obtenir. En conséquence, dans nos simulations, c'est le déroulement de celle-ci qui constitue en elle-même le résultat souhaité.

4.2.2 L'approche naïve pour les déplacements

La problématique de la planification située est double : il s'agit non seulement de faire résoudre à l'agent un problème pour lequel il va utiliser des « opérateurs » (les interactions) et les autres agents présents dans la simulation, mais surtout de prévoir les déplacements nécessaires dans l'environnement au moment de la planification. Ces déplacements, nous le verrons, peuvent eux aussi amener l'agent à résoudre un problème pour pouvoir se déplacer.

La résolution d'un déplacement consiste à trouver une succession d'opérateurs de déplacements élémentaires (par exemple parmi {*nord*, *nord-est*, *est*, *sud-est*, *sud*, *sud-ouest*, *ouest*, *nord-ouest*} dans le cas d'une grille en 2D) permettant de passer d'un état initial (agent dans la position actuelle) à un état final (agent à la position souhaitée). Le problème du calcul de chemin est donc un problème de planification à part entière. Nous avons donc affaire ici à une double planification. Dès lors, pourquoi ne pas simplement trouver une représentation de l'environnement qui soit unifiée et qui permette d'utiliser un unique moteur de planification pour résoudre le plan et calculer les déplacements ? De cette manière, les problèmes de planification induits par les déplacements (par exemple, ouvrir une porte qui se trouve sur le chemin) se trouveraient automatiquement résolus. Pour ce faire, on pourrait créer dynamiquement les interactions de déplacement de la manière suivante : mettre dynamiquement dans la partie *condition* de l'interaction *true* si le déplacement est possible (ni mur ni obstacle), *false* s'il n'est pas possible (mur) ou la condition permettant le déplacement en cas d'obstacle, et mettre dynamiquement dans la partie *action* de l'interaction la nouvelle position de l'agent.

Les raisons pour lesquelles une telle unification de la connaissance n'est pas envisageable sont multiples :

algorithmes différents et adaptés à chacun des deux problèmes : le problème de planification classique et le problème du déplacement ne sont pas de même nature. L'algorithme A* est très adapté pour faire un calcul de chemin, par contre il ne serait pas adapté pour la planification classique avec nos interactions puisqu'on ne pourrait pas lui fournir la fonction heuristique g (cf 2.7) permettant de guider ses choix. De plus, comment raisonner sur le fait que le déplacement nous a rapproché de la cible ? En effet, dans des environnements comportant des obstacles et des murs, un déplacement en direction de la position souhaitée ne rapproche nécessairement pas de celle-ci : il est très probable que l'on soit en train de se diriger vers un mur (voire un cul-de-sac) et qu'il soit plus court de prendre une autre direction (voire une direction complètement opposée en

cas de cul-de-sac) afin de contourner l'obstacle. Raisonner indépendamment sur les pas constituant un chemin n'est pas envisageable, il faut considérer un déplacement dans sa globalité si l'on veut obtenir une solution optimale, ce que garantit en revanche l'A*.

taille du problème : l'environnement est constitué d'un grand nombre de cases, auxquelles correspondent autant d'états possibles, ce qui augmente beaucoup la taille du problème à résoudre par le planificateur. De plus, un algorithme de planification classique ne dispose pas d'une heuristique permettant de guider ses choix comme dans l'A*, ce qui engendre beaucoup de calculs inutiles. Les algorithmes de planification classiques, connus pour avoir un comportement exponentiels lorsque le nombre d'états et d'opérateurs augmentent, risquent d'être très lents à calculer un pas de simulation, ce qui exclut toute utilisation "en temps réel", embarquée dans un robot par exemple.

difficulté de compréhension du plan : le plan obtenu entremêle les déplacements atomiques et les actions, ce qui rend le plan lourd et difficile à comprendre. De plus, il n'est pas possible facilement de faire calculer un plan « abstrait » à l'algorithme. On entend par plan abstrait un plan qui ne tient pas compte des déplacements, et qui est donc générique par rapport à la configuration géographique de l'environnement.

4.2.3 La planification située

La planification dans un contexte situé impose de prendre en compte les déplacements. La problématique posée par l'intégration des déplacements dans la planification est double : il faut non seulement prévoir les déplacements qui permettent l'exécution des interactions (satisfaction de la garde de l'interaction) mais aussi la planification parfois complexe générée par un déplacement (satisfaction des conditions imposées par les obstacles dans l'environnement).

Etant donné que notre but principal est la simulation pas à pas, nous devons garder en tête le fait qu'un plan calculé est destiné à être exécuté réellement dans l'environnement. Nous distinguons donc les *plans abstraits* et les *plans concrets* :

un plan abstrait est un plan dans lequel les positions relatives des agents ne sont pas prises en compte

un plan concret que l'on pourrait nommer aussi **plan d'exécution** est un plan représentant l'exécution du plan abstrait dans un contexte situé particulier. Pour un plan abstrait donné, il y a autant de plans d'exécution possibles que de configurations différentes de l'environnement.

Des problèmes se posent lorsqu'un agent exécute son plan. La prochaine action à effectuer par l'agent est choisie dans le plan abstrait. Celui-ci peut offrir plusieurs exécutions possibles en commutant les actions. Un problème se produit lorsqu'une interaction doit se produire avec un autre agent, ce qui est souvent le cas. Les notions de distance et de voisinage interviennent ici, et le réalisme exige parfois que deux agents soient suffisamment proches pour entrer en interaction. Dans ce cas l'agent *acteur* doit tout d'abord se déplacer afin de se rapprocher de l'agent *cible* de l'interaction. C'est ici qu'une différence essentielle se produit entre le plan abstrait produit par un planificateur classique et le plan réel nécessaire à l'exécution dans un contexte situé. Considérons le plan abstrait suivant : $\{a_1, a_2, a_3\}\, puis\, \{a_4, a_5\}$ dans lequel les positions des agents n'ont pas d'incidence. Même dans un contexte situé, ce plan a un sens : il s'agit d'un plan générique, abstrait par rapport aux aspects situés, que l'on peut donner à exécuter à quelqu'un pour lui expliquer la manière générale de résoudre ce problème.

Charge à lui d'y apporter ensuite les adaptations nécessaires à l'exécution dans un contexte géographique particulier. Par exemple « Pour construire une étagère, vous avez besoin de planches, de clous, d'une scie et d'un marteau. Découpez ensuite les planches à la dimension désirée à l'aide de la scie et assemblez les avec des clous en vous servant du marteau ». Ce plan est valide dans beaucoup de contextes situés, toutefois, les exécutions seront différentes en fonction de chaque contexte particulier, par exemple s'il est nécessaire d'aller chercher les clous dans une autre pièce ou d'aller acheter une scie dans un magasin. De plus, certaines exécutions de ce plan sont impossibles, par exemple si les planches sont dans un atelier qui est fermé à clef ou si l'on n'a pas assez d'argent pour acheter la scie. Ces plans particuliers doivent être étendus à leur tour afin de résoudre les sous-problèmes posés par leur contexte, en l'occurrence déverrouiller la porte et retirer de l'argent.

Le contexte situé et le besoin d'exécuter rationnellement le plan abstrait amène à intégrer les déplacements dans le plan, étendre le plan abstrait pour planifier les déplacements, puis à trouver un ordre d'exécution des actions respectant la notion de rationalité. Nous allons expliciter ces points. Nous verrons que la prise en compte de l'aspect situé dans la planification ne s'arrête pas à ajouter des déplacements dans le plan.

L'intégration des déplacements.

Lorsqu'un plan abstrait est exécuté, il est nécessaire de le confronter à la géographie de l'environnement : l'acteur d'une interaction est obligé de s'approcher de sa cible[22] pour pouvoir l'exécuter. Le déplacement étant une action, il est nécessaire de l'intégrer au plan. A chaque action du plan abstrait correspond un déplacement, ceux-ci sont greffés au plan initial. Considérons par exemple le plan abstrait $\{a_1, a_2, a_3\}$. Si l'on appelle m_i le déplacement permettant d'atteindre la cible de l'action a_i, on obtient un nouveau plan dans lequel les actions ont été entrelacées : $\{\{m_1\ puis\ a_1\}, \{m_2\ puis\ a_2\}, \{m_3\ puis\ a_3\}\}$. Nous voyons que l'exécution d'un plan dans un contexte situé particulier entraîne l'ajout de déplacements. Ceci entraîne un nouveau problème, car les interactions de déplacement entraînent à leur tour une planification.

Planification des déplacements.

Pour être correctement exécutés, les déplacements ont besoin d'être planifiés. Nous avons discuté de ce problème dans [DMR04]. Par exemple, dans le cas où le déplacement vers la cible d'une interaction oblige un agent à passer par une porte qui est fermée, l'agent doit planifier l'ouverture de la porte. Cette action n'apparaît nulle part dans le plan abstrait, ce qui est logique puisque la porte n'est présente que dans certains contextes géographiques, et peut parfois être déjà ouverte lorsqu'elle est présente. Si en plus la porte est verrouillée, cela oblige l'agent à trouver la clef de celle-ci avant même d'essayer de déverrouiller et ouvrir celle-ci. Cette nouvelle action (prendre la clef) entraîne à son tour un déplacement, qui entraîne une nouvelle planification, et ainsi de suite. Pour un même plan abstrait, nous avons donc plusieurs exécutions possibles, comme le montre le tableau 4.1.

[22] Ce comportement dépend de la garde de l'interaction : il n'est pas systématique pour toutes les interactions !

plan abstrait	interagir avec O
plan situé	aller à O, interagir avec O
plan situé avec porte fermée	aller à la porte, ouvrir la porte, aller à O, interagir avec O
plan situé avec porte fermée et verrouillée	aller à la clef, ramasser la clef, aller à la porte, déverrouiller la porte, ouvrir la porte, aller à O, interagir avec O

TAB. 4.1 – Différents plans concrets peuvent découler d'un même plan abstrait en fonction de la configuration de l'environnement dans lequel on l'exécute.

Monkey-Banana Problem : planification située ?

Certains travaux sur la planification résolvent des problèmes qui semblent être situés. Pourtant les déplacements ne sont réellement exécutés à aucun moment, et ils ne sont pas confrontés à la réalité de l'environnement. On ne tient pas compte du fait que ces déplacements soient possibles ou non. C'est le cas d'un problème de planification très classique : le *monkey-banana problem* [Ba85]. Ce problème donne l'impression d'un problème situé, pourtant même si le plan permettant de le résoudre fait apparaître des déplacements, les positions relatives des agents ne sont pas réellement prises en compte et les déplacements ne sont pas réellement exécutés. La position de l'agent est considérée comme n'importe laquelle des propriétés de l'agent et un déplacement ne peut pas mener à un échec, contrairement à ce qui se passe dans la réalité, et qu'intègre notre modèle.

Le problème est le suivant : un singe affamé se trouve à l'entrée d'une pièce au milieu de laquelle est suspendue une banane, trop haute pour être attrapée par le singe. Devant la fenêtre se trouve une boîte. Les actions que le singe peut entreprendre sont *se déplacer*, *pousser* la caisse sous la banane, *grimper* sur la caisse et *attraper* la banane.

On peut coder un état du problème avec le quadruplet (M, B, O, G) où :
- M représente l'endroit où est le singe. Les valeurs possibles sont `{porte, fenêtre, milieu}`
- B représente l'endroit où est la boite. Les valeurs possibles sont `{porte, fenêtre, milieu}`
- O représente le fait que le singe est ou non sur la boîte. Les valeurs possibles sont `{à terre, sur la boîte}`
- G représente le fait que le singe possède ou non la banane. Les valeurs possibles sont `{possède, ne possède pas}`

L'état initial du problème est donc `(porte, fenêtre, à terre, ne possède pas)` tandis que l'état final est `(_, _, _, possède)` où les "`_`" représentent n'importe quelle valeur.

Voici la description des opérateurs. Un opérateur est accompagné de variables. Il est constitué d'un état initial et d'un état final. Pour pouvoir appliquer un opérateur, le monde doit se trouver dans un état compatible avec l'état actuel, et se trouve ensuite dans l'état final proposé par l'opérateur. Les tirets bas permettent de représenter les éléments dont la valeur n'a pas d'importance pour permettre le déclenchement de l'opérateur.

```
(marcher(p1, p2) // aller de la place p1 à la place p2
(p1, _, _, _)
(p2, _, _, _))
```

```
(pousser(p1, p2) // pousser la caisse de p1 à p2
(p1, p1, à terre, _)
(p2, p2, à terre, _))

(grimper() // monter sur la caisse
(_, _, à terre, _)
(_, _, sur la boîte, _))

(attraper() // attraper la banane
(milieu, milieu, sur la boîte, ne possède pas)
(milieu, milieu, sur la boîte, possède))
```

Un chaînage arrière simple permet de trouver la solution, à partir de l'état final souhaité : `(_, _, _, possède)` nous oblige à utiliser `attraper` puisque c'est la seule action qui permet d'obtenir `G = possède`. Pour pouvoir utiliser `attraper`, on doit avoir `(milieu, milieu, sur la boite, ne possède pas)` ce qui nous force à utiliser `grimper` pour avoir `O = sur la boîte`. On doit donc avoir `(milieu, milieu, à terre, ne possède pas)`.

De la même manière on déduit ensuite `pousser(fenêtre, milieu)` et `marcher (porte, fenêtre)`.

On obtient finalement le plan :

1.	`marcher(porte, fenêtre)`
2.	`pousser(fenêtre, milieu)`
3.	`grimper()`
4.	`attraper()`

Le singe doit donc aller à la fenêtre, pousser la boîte sous la banane, monter sur la boîte, et prendre la banane. Le problème du singe et de la banane est très facile à résoudre avec un algorithme de chaînage arrière classique.

Le problème, les opérateurs disponibles et la solution du problème nous apparaissent comme étant « situés » puisqu'ils font appel à des opérateurs de déplacement. Pourtant, ces déplacements n'en sont pas vraiment : on peut remplacer les opérateurs qui semblent situés par d'autres qui ne le sont pas sans que cela ne change rien au problème :

Réécrivons l'histoire avec un caméléon amélioré. Pour attraper la banane, il doit adopter la même couleur que celle-ci et utiliser son couteau suisse déplié, lui aussi de couleur jaune. Le caméléon peut s'aider d'une faculté supplémentaire qui lui permet de faire changer la couleur d'un objet en même temps que la sienne.

On obtient alors un problème qui s'écrit ainsi :
- M représente la couleur du caméléon. Les valeurs possibles sont `{vert, rouge, jaune}`
- B représente la couleur du couteau. Les valeurs possibles sont `{vert, rouge, jaune}`
- O représente le fait que couteau est déplié. Les valeurs possibles sont `{ouvert, fermé}`
- G représente le fait que le caméléon possède ou non la banane. Les valeurs possibles sont `{possède, ne possède pas}`

L'état initial du problème est donc `(vert, rouge, fermé, ne possède pas)` tandis que l'état final est `(_, _, _, possède)` où les "`_`" représentent n'importe quelle valeur.

Voici les nouveaux opérateurs :

```
(changercouleur(c1, c2) // passer de la couleur c1 à la couleur c2
(c1, _, _, _)
(c2, _, _, _))

(déteindre() // changer de couleur ainsi que celle du couteau
(c1, c1, fermé, _)
(c2, c2, fermé, _))

(déplier() // ouvrir le couteau
(_, _, fermé, _)
(_, _, ouvert, _))

(attraper() // attraper la banane
(jaune, jaune, ouvert, ne possède pas)
(jaune, jaune, ouvert, possède))
```

Le problème se résout exactement de la même manière et l'on obtient le plan suivant, mis en rapport avec le plan obtenu pour le problème original :

1.	`changercouleur(vert, rouge)`	`marcher(porte, fenêtre)`
2.	`déteindre(rouge, jaune)`	`pousser(fenêtre, milieu)`
3.	`déplier()`	`grimper()`
4.	`attraper()`	`attraper()`

Il est clair que le problème n'a pas été changé en substance, mais seulement en apparence. Il n'y a aucun obstacle au changement de couleur tout comme il n'y avait aucun obstacle au déplacement dans le problème original. Or la particularité d'un problème situé réside justement dans les problèmes liés aux déplacements et la manière de les résoudre.

4.2.4 Le moteur

Les agents disposent d'une représentation du monde qui les entoure : l'environnement, les autres agents et les interactions. Les agents doivent utiliser ces connaissances afin d'atteindre leur(s) but(s). C'est le rôle du moteur de comportement.

Le moteur de comportement est une "boîte noire" disposant d'interfaces avec le monde environnant : les capteurs et les effecteurs. Les capteurs permettent à l'agent de prendre les informations qui lui sont nécessaires dans l'environnement, et les effecteurs lui permettent d'interagir avec les autres agents, en utilisant les interactions qu'il connaît.

Le moteur de comportement d'un agent est composé de plusieurs modules (4.1). Le module de perception permet de prendre des informations dans l'environnement, et le moteur de mise à jour se charge de répercuter les changements dans la mémoire et dans le plan. Le module de planification calcule un plan à partir des informations présentes dans la mémoire, tandis que le module de sélection détermine la prochaine action à accomplir parmi toutes celles envisageables dans le plan. Enfin, le module d'exécution tente d'exécuter l'action choisie, et informe le module de mise à jour de la réussite ou de l'échec de l'exécution.

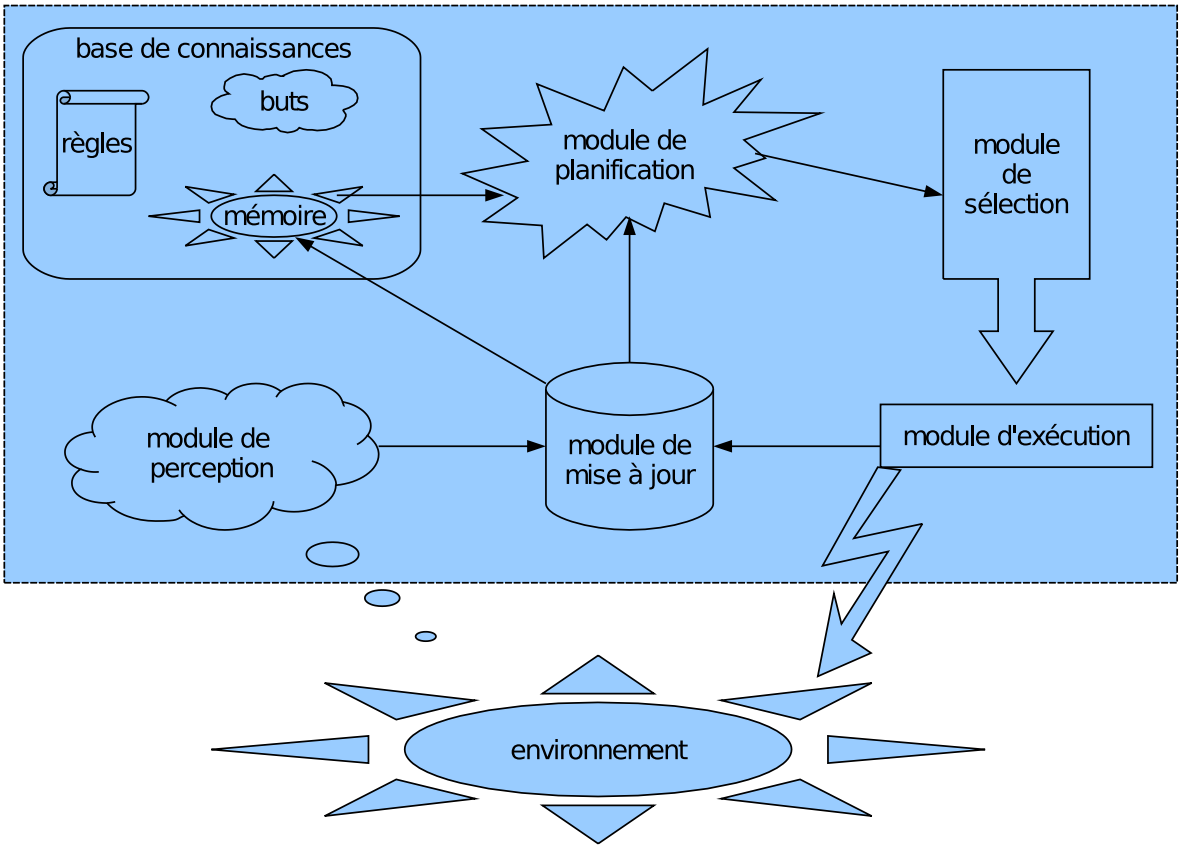


FIG. 4.1 – Le moteur de comportement est constitué de divers modules : certains sont dédiés au stockage des informations, d'autres au calcul ou recalcul du plan.

Le module de perception

Le module de perception implémenté dans nos agents ne prend en compte pour l'instant que les informations "visuelles". On pourrait rajouter facilement la prise en compte des informations sonores ou olfactives. Il prend en considération l'environnement à l'intérieur d'un halo de vision, ce qui permet de simuler la vue, qui est limitée à une certaine distance. Il serait envisageable, pour simuler plus fidèlement la vue, de paramétrer la distance de vision en fonction de la taille des objets considérés. Toutes les informations perçues dans le halo de perception sont envoyées au module de mise à jour.

Dans nos simulations, le halo de perception est défini comme un carré de 11 cases de côté centré sur l'agent, dans lequel les cases masquées par un mur sont supprimées (figure 4.2). Ces cases sont détectées par un mécanisme de lancé de rayons. Ce halo de perception définit un masque à l'intérieur duquel l'agent perçoit les places, les passages et les agents de l'environnement réel.

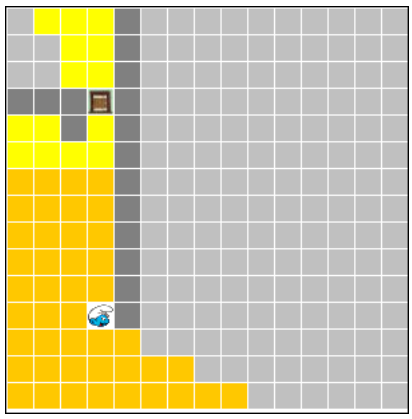


FIG. 4.2 – Le halo de perception d'un agent est représenté par les cases orange.

Le module de mise à jour

Le module de mise à jour fait le lien entre l'environnement réel et l'environnement "virtuel" stocké dans la mémoire de l'agent. Il calcule les différences entre l'environnement perçu par le biais du module de vision et l'environnement réel. Ainsi, seuls les changements intervenus dans la partie considérée de l'environnement depuis qu'on l'avait vu pour la dernière fois sont considérés. Les différences sont calculées "case par case" dans l'environnement. Voici les changements auxquels on peut s'attendre :

- nouvel agent
- agent connu disparu d'une place
- agent connu apparu sur une place
- propriété modifiée
- nouvelle place
- place supprimée
- nouveau lien entre deux places
- lien supprimé entre deux places

Les changements sont répercutés dans la mémoire, afin de mettre à jour l'environnement virtuel sur lequel s'appuie le moteur de planification.

Dans une seconde phase, on regarde si le plan actuel est toujours valide au regard des changements relevés, et on le modifie localement si nécessaire. Cette mise à jour du plan est détaillée dans la partie 4.3.

La mémoire

La mémoire d'un agent est une copie imparfaite de l'environnement, sur lequel l'agent raisonne. Dès lors, on peut se demander pourquoi on travaille avec cette copie de l'environnement plutôt qu'avec l'environnement original. La première partie de la réponse doit se chercher au niveau du cahier des charges que nous nous étions fixés au départ, qui définissait l'agent comme étant non omniscient. Il est donc nécessaire à l'agent de découvrir son environnement, et de retenir le fruit de ses découvertes. La seconde raison est la dynamicité de l'environnement, rencontrée dès qu'un autre agent animé intervient dans la simulation : en copiant dans sa mé-

moire l'environnement tel qu'il l'a perçu, l'agent est amené à raisonner sur des informations potentiellement inexactes. Ainsi, s'il a croisé un agent A à dans la place p à un instant donné, l'agent va baser son raisonnement sur cette croyance, même si l'agent a bougé, comme le ferait naturellement un être humain. Ceci va se traduire par des comportements surprenants, car l'agent va parfois se tromper lors de l'exécution de son plan. Toutefois, nous verrons que ces erreurs de l'agent sont souhaitables et qu'elles ne traduisent en aucun cas un problème dans le moteur de comportement !

Le calcul du plan

Les modules situés en amont se sont chargés de récolter l'information, de l'organiser et de la stocker dans la mémoire. L'ensemble de ces informations peut maintenant être utilisé par le moteur de planification afin de calculer une séquence d'actions qui mène au but de l'agent. La représentation de connaissances que nous avons choisie nous permet de calculer le plan à l'aide d'un chaînage arrière. On peut donner deux types de buts à un agent : un but *prémisse* (exemple : $button.pushed = true$) qui est une expression à rendre vraie, ou un but *interaction* (exemple : *eat apple*) qui est une action que l'agent doit déclencher.

La résolution d'un but se fait grâce à un chaînage arrière. Ce chaînage se décompose en deux phases en fonction du type de but à résoudre :

La résolution d'un but de type interaction

Il s'agit du cas le plus simple car il ne fait pas appel au chaînage. La solution se trouve dans la sémantique des interactions décrite en 3.2 : pour pouvoir exécuter une interaction, il faut que toutes les conditions et la garde de l'interaction soient vérifiées. Ces dernières deviennent donc naturellement les nouveaux buts de l'agent, après avoir été instanciées en fonction de la source et de la cible considérées. Les sous-buts obtenus sont des sous-buts de type prémisse, dont la résolution est expliquée ci-après. Dans l'arbre de buts représentant le plan de l'agent, on dérive le but de type interaction à résoudre en un *nœud ET* dont les fils sont les conditions et la garde de l'interaction (figure 4.3).

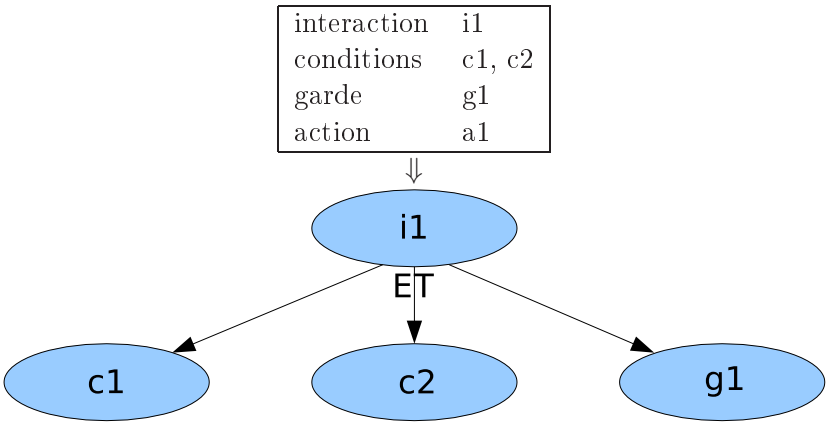


FIG. 4.3 – Nœud ET produit par la résolution d'un but de type interaction $i1$: les conditions et la garde de l'interaction deviennent les nouveaux buts à résoudre.

La résolution d'un but de type prémisse

Cette résolution est moins triviale, c'est ici qu'intervient le chaînage. Ce chaînage ce fait en deux phases :

sélection de l'interaction candidate : parmi toutes les interactions disponibles dans les *peut-effectuer* de l'agent, le moteur sélectionne les interactions qui peuvent aider à la résolution de la prémisse, grâce au concept de *backward action* défini ci-dessous.

instanciation de l'interaction : dans un second temps, le moteur essaie d'instancier chacune des interactions choisies avec des cibles connues qui peuvent la subir, afin de déterminer un couple interaction cible dont l'exécution rend vraie la prémisse à résoudre.

Chaque couple interaction/cible comportant dans sa partie action la prémisse souhaitée est retenu comme solution pour résoudre celle-ci. Lorsqu'il existe plusieurs couples interaction/cible, l'agent se trouve face à une alternative : au moment de l'exécution, il aura le choix entre ces différents couples pour résoudre la prémisse. Dans l'arbre de buts représentant le plan de l'agent, on dérive le but de type prémisse à résoudre en un *nœud OU* dont les fils sont les différents couples interaction/cible déterminés par le chaînage (figure 4.4).

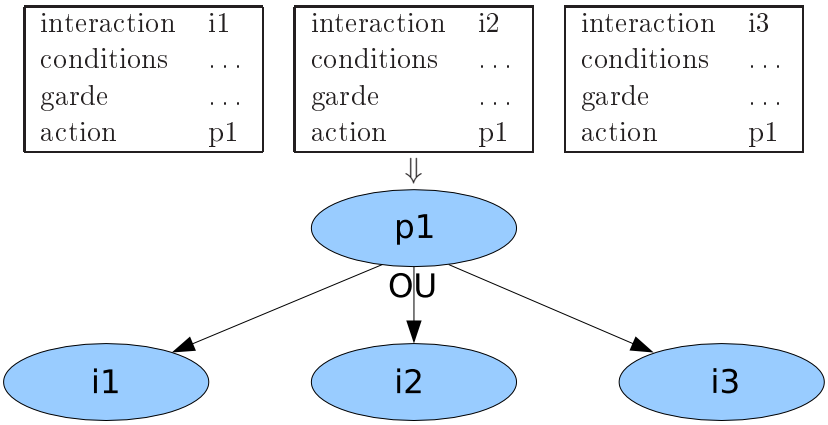


FIG. 4.4 – Nœud OU produit par la résolution d'un but de type prémisse $p1$: les interactions dont la partie action est adéquate deviennent des sous-buts alternatifs.

Nous allons étudier le fonctionnement du moteur de planification au travers d'un exemple. L'agent dispose de trois interactions qui sont données par la table 4.2. Le but de l'agent est *open blue_ door* . Pour pouvoir exécuter cette interaction il faut résoudre les conditions de celle-ci. En particulier il faut résoudre *actor.own(target.key)*. Cette condition est instanciée : *actor.own(blue_ door.key)* puis déréférencée : *actor.own(blue_ key)*. Le moteur de planification cherche alors une interaction dont la partie action comporte le même schéma, c'est le cas de l'interaction *take* dont l'action est *actor.own(target)*. Le moteur tente alors d'instancier cette interaction avec tous les objets connus pouvant subir l'interaction *take*. En instanciant l'interaction avec l'objet *blue_ key*, la partie action devient *actor.own(blue_ key)*, qui est la prémisse recherchée.

	open	take	eat
C	actor.own(target.key) target.open = false	-	actor.own(target)
G	d(actor, target) < 1	d(actor, target) < 1	d(actor, target) < 1
A	target.open = true	actor.own(target)	actor.health += target.energy destroy(target)

TAB. 4.2 – code des interactions *open*, *take* et *eat*

La résolution d'un but par le chaînage arrière produit de nouveaux buts qui sont ajoutés au plan. Le plan obtenu est un arbre ET/OU alternant des buts de type interaction et des buts de type prémisse.

Le concept de *backward action*

Pour résoudre un but de type prémisse p, le moteur de chaînage arrière doit trouver une interaction i dont une action a rende vraie la prémisse p considérée. Une interaction est donc choisie en fonction des actions qu'elle comporte. Dans un chaînage arrière classique à la STRIPS, une prémisse se réduit à un fluent, tout comme chaque action d'une interaction. Le chaînage est donc immédiat car basé sur une simple identification des actions de l'interaction :

```
for each interaction i in actor.canPerform
  if p is in i.actions
    add i to the result
```

Dans le cas qui nous concerne, les prémisses ne sont pas limitées à des fluents. Elles peuvent être de trois types :

1. **propriété égale à *true* ou *false* :** ce cas se ramène au cas STRIPS puisqu'il suffit de trouver une expression identique dans la partie action d'une interaction. Ceci permet de gérer les propriétés booléennes, comme l'état allumé ou éteint d'une lampe.
2. **comparaison d'une propriété à une valeur entière :** selon que la prémisse fixe une borne supérieure ou inférieure à une propriété, il faudra respectivement soustraire ou ajouter une quantité à celle-ci. Le cas de l'égalité est géré imparfaitement pour le moment : il consiste à se rapprocher de la valeur souhaitée en ajoutant une quantité à une valeur trop basse et en soustrayant une quantité à une valeur trop haute. Ce type de prémisse permet de gérer des propriétés comme l'énergie d'un agent, sa richesse... De même, c'est ainsi que sont gérés les déplacements : les actions *moveTo* et *moveAway* comportent respectivement l'action permettant de diminuer ou d'augmenter la distance à une cible.
3. **appartenance à une collection :** l'action permettant de garantir qu'un objet appartient à une collection est l'ajout dans celle-ci. Ainsi, une prémisse de la forme *collection.contains*(*objet*) sera résolue par l'action *collection.add*(*objet*). Ceci permet en particulier de gérer l'inventaire de l'agent.

Nous connaissons et utilisons ces règles dans la vie de tous les jours : lorsque je souhaite que le réservoir de ma voiture dispose de plus de 40 litres de carburant, je sais que je dois en ajouter dans le réservoir. Si je souhaite que mon sac contienne un portefeuille, je sais que je dois ajouter celui-ci dans mon sac. Ces informations indispensables au chaînage font partie intégrante de la représentation de la connaissance. Nous les appelons ces règles *backward actions*, en voici la liste schématique :

type de prémisse	backward action correspondante
propriété > entier	add(propriété, *)
propriété < entier	substract(propriété, *)
propriété = entier	add(propriété, *) ou substract(propriété, *)
collection.contains(x)	collection.add(x)

L'algorithme permettant d'effectuer le chaînage entre une prémisse et une interaction ne pouvant pas se contenter d'identifier une prémisse et une action comme dans le cas de STRIPS, il identifie la *backward action* d'une prémisse et une action :

```
for each interaction i in actor.canPerform
  if backwardAction(p) is in i.actions
    add i to the result
```

Les *backward actions* permettent d'augmenter l'expressivité des interactions et de réaliser des chaînages que ne permet pas le langage STRIPS.

L'algorithme

L'algorithme de planification est donné en figure 4.5. Il donne la méthode récursive d'expansion d'un nœud en fonction de son type. Un nœud *interaction-node*, *condition-node* ou *move-node* correspond respectivement à un but de type interaction, de type prémisse ou au cas particulier de l'interaction de déplacement.

La sélection d'actions

Une fois que l'agent a calculé le plan, il doit en déclencher l'exécution. Nous avons vu que le plan a une structure d'arbre ET/OU. La résolution d'un nœud de l'arbre dépendant de la résolution de ses fils, l'exécution d'une interaction qui se trouve à un niveau intermédiaire de l'arbre n'est pas possible tant que les interactions portées par les fils de ce nœud n'ont pas été exécutées. En conséquence, seules les interactions présentes au niveau des feuilles de l'arbre sont susceptibles d'être exécutées. Un choix se pose toutefois à l'agent, car les feuilles peuvent être nombreuses. Dans un contexte non situé, l'ordre dans lequel on exécute ces interactions n'a pas grande importance. En revanche, dans le contexte situé dans lequel nous nous plaçons, cet ordre aura une influence sur la distance totale parcourue par l'agent pour exécuter la totalité du plan. Les interactions sont accompagnées de cibles, et la garde de distance stipule généralement qu'il est nécessaire d'être proche de la cible pour pouvoir exécuter l'interaction. En conséquence, on peut estimer que pour réaliser un plan, un agent devra s'approcher de toutes les cibles qui accompagnent les interactions. On pourrait utiliser un TSP [TSP] pour résoudre ce problème, mais les méthodes de résolution de cette classe de problèmes sont trop

```
expand()
  if this == interaction-node then
    for each condition c in this.condition and this.guard
      newNode = createConditionNode(c)
      this.sons.add(newNode)
      newNode.expand()
  else if this == condition-node then
    if this.condition.isSatisfied()
      then finished
      else
        // recupération de la backward action permettant le chaînage
        BA = this.condition.getBackwardAction()
        // récupération des interactions candidates pour BA
        LI = actor.getCanPerform(BA)
        for each I in LI
          if I is "moveTo"
            then this.sons.add(createMoveNode())
          else
            // liste des agents connus pouvant subir I
            lAgents = actor.getKnownAgents(I)
            for each a in lAgents
              // si l'application de I sur a satisfait la condition
              if actor.execute(I,a) satisfies this.condition
                 newNode = createInteractionNode(I,a)
                 this.sons.add(newNode)
                 newNode.expand()
              end if
          end if
          if this.sons.isEmpty()
            then this.sons.add(create-exploration-node)
  else if this == move-node then
    path = actor.computePath()
    // s'il n'existe pas de chemin
    if path == null
      then this.sons.add(createConditionNode("false"))
      else
        // énumération des conditions relevées sur le chemin
        conditionsList = path.getPathsConditions()
        if conditionsList.isEmpty()
          then this.sons.add(createConditionNode("true")
        else for each condition c in conditionsList
          newNode = createConditionNode(c)
          this.sons.add(newNode)
          newNode.expand()
```

FIG. 4.5 – L'algorithme de planification. *actor* désigne l'agent qui construit le plan

gourmandes quand le nombre de cibles augmente, et ne sont donc pas adaptées à la prise de décision qu'un agent doit prendre à chaque pas de simulation. Nous avons donc opté pour une heuristique simple et rapide qui nous semble proche de la manière dont un humain se comporte en pareille situation : aller au plus près, et de proche en proche. A tout moment, l'agent choisit donc l'interaction dont la cible se trouve la plus proche de lui à cet instant (figure 4.6). Cette heuristique n'est pas optimale, elle peut-être mise en défaut en créant des cas très particuliers qui obligeront l'agent à faire de nombreux aller-retours, mais elle est efficace dans la plupart des cas et son faible coût calculatoire la rend très adaptée à ce problème.

```
select-action()
  selected = null
  for each leaf l in tree
    if selected == null or cost(l) < cost(selected)
      selected = l
  return selected

cost(goal)
  return distance(actor, goal.target)
```

FIG. 4.6 – Une sélection d'action

Un mécanisme de sélection d'action autorisant une plus grande diversité de comportements constitue actuellement l'objet de recherches menées par Tony Dujardin dans le cadre de sa thèse. Cette sélection d'action considère trois facteurs : les priorités dynamiques, l'opportunisme et la personnalité de l'agent [DMR07].

L'exécution du plan

L'exécution du plan est réalisée « pas à pas ». A chaque pas d'exécution, une action est choisie dans le plan par le mécanisme de sélection d'action et elle est exécutée de manière atomique. Dans le contexte d'une simulation ne faisant intervenir qu'un agent animé, l'exécution est triviale : elle se résume simplement à mettre à jour les propriétés des agents (acteur y compris) dans la mémoire et dans l'environnement réel. Aucun élément inconnu ne peut venir perturber l'agent puisqu'il est seul à agir dans l'environnement : l'état de l'environnement au moment de l'exécution est le même que lors de la planification.

Le cas des déplacements est toutefois singulier : un déplacement vers une cible ne peut pas être exécuté de manière atomique. En effet, dans ce cas, il s'agirait d'une téléportation sur laquelle les obstacles ne pourraient pas avoir d'effet. Pour l'exécution, un déplacement est donc découpé en « pas de déplacement » atomiques, correspondant au franchissement d'un arc dans le chemin calculé par l'agent. Un pas de déplacement constitue une action atomique, au même titre que le déclenchement d'une interaction.

Nous disposons maintenant d'un agent doté d'un moteur de comportements composé de divers modules : le module de perception assurant la prise d'informations, le module de planification qui prend les décisions par rapport aux buts de l'agent et à l'état de sa mémoire, et

enfin le module d'exécution qui se charge d'appliquer le plan dans l'environnement. Le moteur de planification de l'agent est capable d'intégrer les déplacements au raisonnement afin de résoudre le plus tôt possible les problèmes imposés par celui-ci. Nous allons voir maintenant les contraintes apportées par la présence d'autres agents et la manière de les résoudre.

4.3 La concurrence

Les simulations dans lesquelles un seul agent animé intervient sont rares : dans une grande majorité des cas, une entité extérieure à l'agent effectue des modifications de l'environnement à son insu. Dans le cas où un concepteur réaliserait une simulation ne comportant qu'un agent animé, il jouerait lui-même le rôle de l'entité perturbatrice en faisant des modifications dans l'environnement pour tester l'agent. Le cas d'un agent isolé dans l'environnement est aussi marginal que peu intéressant, aussi nous allons maintenant détailler le fonctionnement du moteur de comportements dans le contexte de concurrence, après avoir fait un rapide rappel sur ce dernier : même si le mot « concurrence » désigne dans le langage courant une rivalité entre deux entités possédant un objectif similaire, nous le considérons dans son sens strictement étymologique qui signifie *courir ensemble (concurrere)* et non pas *courir contre*. Il s'agit en fait d'une *exécution concurrente* dans laquelle les ressources sont partagées sans contrainte explicite de compétition, comme défini en 2.3. La présence (incarnée ou non) d'une autre entité dans la simulation entraîne un décalage entre l'état de la mémoire de l'agent et l'état réel de l'environnement. C'est ce que nous appelons la *non-monotonie* ou la *dynamicité* de l'environnement.

La dynamicité de l'environnement

Le contexte de concurrence fait peser sur les agents la contrainte de la dynamicité de l'environnement. Contrairement au cas d'un agent isolé (strictement, c'est-à-dire sans intervention d'une entité extérieure, pas même le créateur/utilisateur de la simulation) où il n'est pas nécessaire de vérifier que l'état de la mémoire correspond à celui de l'environnement, les agents se trouvant dans un contexte de concurrence ne peuvent pas tenir pour définitivement vraie une connaissance qu'ils viennent de retirer de l'environnement, si récente soit elle. Un autre agent peut avoir modifié l'environnement pour ses propres besoins, ou le créateur/utilisateur de la simulation peut avoir volontairement effectué des modifications afin d'observer la réponse de l'agent. Cette dynamicité de l'environnement se retrouve aussi dans le cas d'un agent incarné dans un robot qui évolue dans un environnement soumis à l'influence de nombreux facteurs extérieurs.

Conséquences de la dynamicité

Dans un environnement non monotone, des différences peuvent apparaître entre la mémoire de l'agent et l'environnement. L'agent ne peut donc plus considérer que les connaissances présentes dans sa mémoire sont des vérités absolues, et doit vérifier ses connaissances à certains moments. Lorsqu'il se déplace dans une zone déjà découverte de l'environnement, l'agent doit mettre à jour les changements qui se sont produits dans l'environnement. C'est le rôle du module de mise à jour. Nous verrons que cette mise à jour déclenche des modifications dans

le plan. De même, au moment où un agent exécute une interaction à la suite du processus de planification et sélection, l'environnement vérifie que les conditions de l'interaction sont effectivement remplies au moment de l'exécution. Dans le cas contraire, l'agent se trouve dans une situation d'échec. Il tient alors compte des changements grâce au module de mise à jour et son plan est recalculé afin d'intégrer les nouvelles données et de trouver une manière de résoudre le problème qui empêche de déclencher l'interaction.

Exécution et échecs

Lorsque l'agent exécute l'action qu'il a choisie dans le plan, cette exécution ne peut pas se limiter à une simple mise à jour des propriétés dans l'environnement et dans sa mémoire. En effet, nous l'avons souligné auparavant, la mémoire de l'agent peut-être en contradiction avec la réalité de l'environnement du fait de sa non-omniscience et de la dynamicité de l'environnement en présence d'autres agents. Lorsque l'agent tente d'exécuter l'interaction, l'environnement — qui possède le code de toutes les interactions — est chargé de vérifier que son état est compatible avec les conditions de l'interaction que l'agent veut déclencher. Si ce n'est pas le cas, il informe l'agent de l'état réel de l'environnement, en particulier de la propriété qui pose problème pour exécuter l'interaction. L'agent va intégrer cette nouvelle information dans son plan afin de l'adapter grâce à un mécanisme de replanification partielle (figure 4.7).

Il est important de souligner le fait que lorsqu'un agent se trouve en situation d'échec, ce n'est pas son plan qui est faux, mais les connaissances utilisées pour le construire. Par voie de conséquence, le plan n'est pas adapté à la réalité de l'environnement et il échoue. Ce comportement est tout à fait souhaitable pour simuler des environnements dynamiques dans lesquels l'état du monde change durant l'exécution du plan calculé par un agent.

Mise à jour des informations et replanification partielle

Au fur et a mesure de son évolution dans l'environnement, un agent reçoit des nouvelles informations grâce au module de perception. Ces nouvelles informations peuvent avoir plusieurs origines :

dynamicité de l'environnement : l'agent évolue dans un environnement peuplé d'autres agents. Ces agents se déplacent dans l'environnement, interagissent entre eux et sur l'environnement. Il en résulte des modifications extérieures des propriétés et positions des agents.

non omniscience : l'agent n'a pas connaissance de la totalité de l'environnement au début de la simulation. Il découvre l'environnement, les obstacles, les autres agents et leurs propriétés.

échec d'exécution : l'agent peut échouer lors de l'exécution d'une interaction et être informé de l'inexactitude d'une de ses informations par l'environnement.

Afin d'intégrer ces informations, nous avons choisi d'effectuer une replanification partielle en fonction du type d'information. La figure 4.8 illustre cette replanification : les nouvelles informations sont propagées dans l'arbre et réagissent avec les nœuds lorsque cela est justifié. A droite dans l'arbre, la nouvelle information fait passer une condition du statut *vrai* à *faux*, justifiant de greffer un sous-plan pour résoudre cette condition. A l'inverse, à gauche, une nouvelle information fait passer une condition du statut *faux* à *vrai*, justifiant la suppression du sous-plan devenu inutile. Au milieu, la nouvelle information n'a aucune influence.

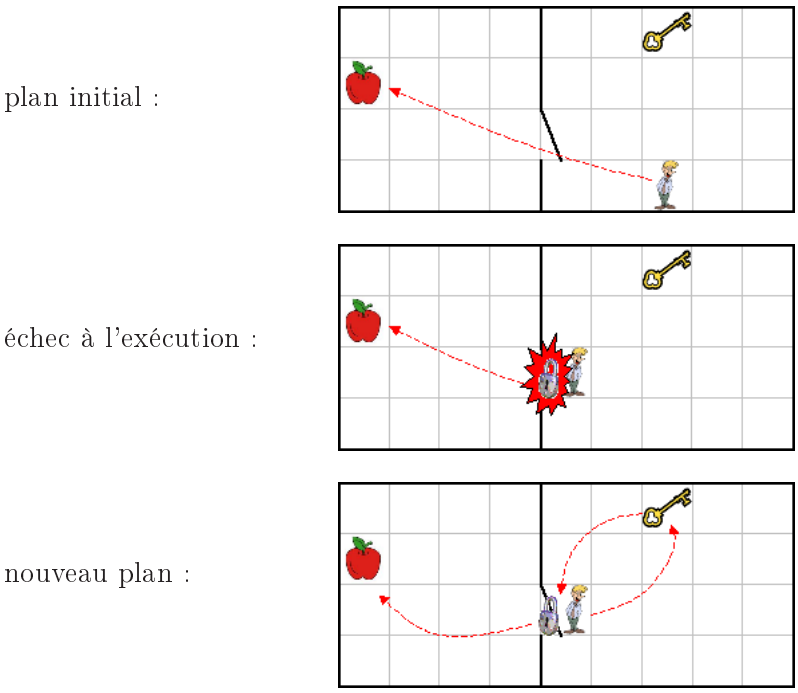


FIG. 4.7 – Echec et replanification : dans la mémoire de l'agent, la porte n'est pas verrouillée. Son plan pour prendre la pomme est donc de se déplacer directement jusqu'à celle-ci. Lors de la tentative d'exécution de son plan, l'agent se trouve en situation d'*échec* car la porte est en réalité verrouillée. L'agent prend connaissance de cette nouvelle information et établit un nouveau plan à partir de sa situation actuelle pour aller chercher la clef, déverrouiller la porte et atteindre la pomme.

L'algorithme de replanification est donné en figure 4.9. La fonction `relatesTo(info, node)` permet de déterminer la pertinence de l'information par rapport au type de nœud dans l'arbre de but.

La figure 4.10 décrit le fonctionnement général de la fonction `relatesTo(info, node)`. Une croix dans le tableau signifie que l'information peut avoir une influence sur un nœud de ce type. Lorsqu'une condition est précisée, cela signifie qu'un test supplémentaire est réalisé pour juger de la pertinence de l'information. Le cas de l'information `agentMoved` est subdivisé en trois cas, selon que l'agent a quitté une position connue pour une position inconnue `(KU)`, qu'il est apparu a une position alors qu'on ne savait plus où il se trouvait `(UK)` ou qu'il a quitté une position connue pour une position inconnue `(KK)`. Les nombreuses cases vides de ce tableau indiquent les cas dans lesquels l'information n'a aucune influence sur le nœud.

Lorsque la fonction `relatesTo(info, node)` détermine que l'information a une influence sur un nœud, la fonction `modify(info, node)` est invoquée. Le nœud est réévalué, et le sous-arbre n'est recalculé (fonction `expand()`) qu'en cas changement de valeur.

Il a été prouvé que dans le pire des cas, la complexité de la replanification est la même que celle de la planification complète [NK93]. Toutefois, le simulateur que nous avons créé permet de constater que le plan est rarement modifié durant l'évolution de l'agent.

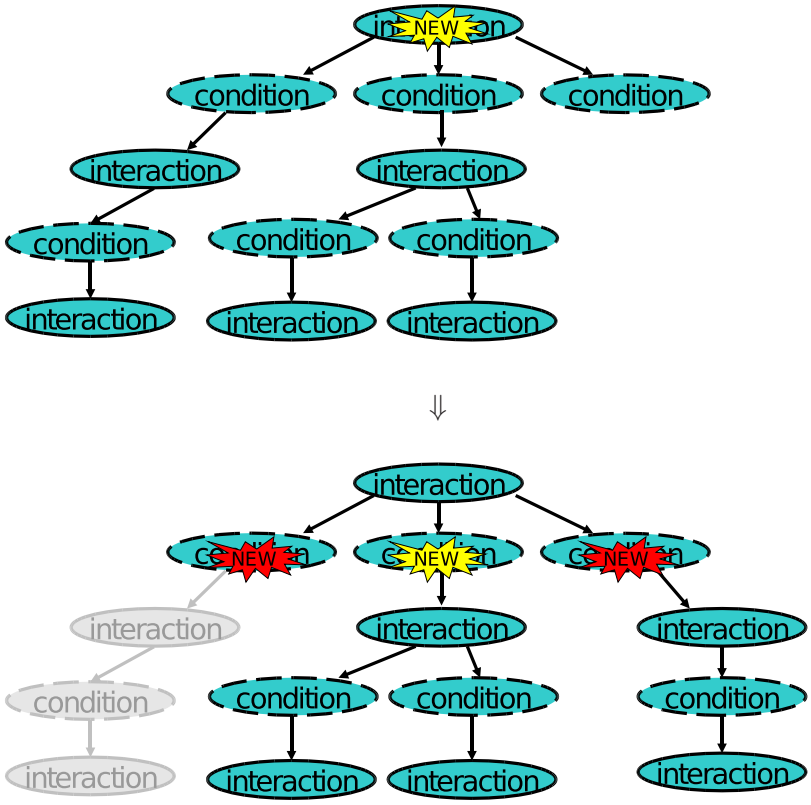


FIG. 4.8 – Mécanisme de replanification.

```
apply(node, newinfos)
  for each info in newinfos
    if info.relatesTo(info, node)
      modify(info, node)
  for each child in getChilds(node)
    apply(child, newinfos)
```

FIG. 4.9 – L'algorithme de replanification partielle

Node / Info	InteractionNode	MoveNode	PremissNode
actionPerformed		x	x
actorMoved		x	x
agentMoved (KU or UK)			agent == target
agentMoved (KK)		agent == target	
agentModified	agent == target		x
newAgent			x
newArcCondition		x	
newPlace		x	

FIG. 4.10 – Pertinence des nouvelles informations en fonction du type de nœud

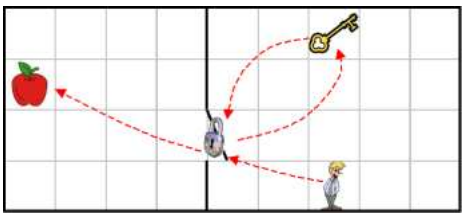
le cas particulier des déplacements dans la planification

Nous avons vu en 3.2 que les gardes de distances permettaient d'éviter le déclenchement d'interactions à distance. Leur résolution entraîne un raisonnement un peu particulier. Pour résoudre une garde de distance, il est nécessaire d'utiliser l'interaction *move* : ce chaînage est réalisé grâce à une *backward action* particulière, comme nous l'avons vu précédemment. Cette interaction est particulière parce qu'elle ne comporte aucune condition ni aucune garde. A priori, on pourrait croire qu'elle ne génère donc aucune planification, c'est d'ailleurs ce qui se passe lors d'une planification « classique » dans lesquels les déplacements sont gérés comme des téléportations. En réalité, les conditions de l'interaction *move* sont dynamiques : lorsque le moteur de planification doit résoudre une interaction *move*, il calcule le chemin qui permet de se rendre à la cible, grâce à l'algorithme A* [BF81] communément utilisé dans les jeux vidéos pour faire du *pathfinding*. Le chemin calculé par l'algorithme est constitué d'une liste de liens qui portent éventuellement des conditions. Ces conditions dépendent de la configuration actuelle de l'environnement, à savoir la position actuelle de l'agent, des autres agents et de leurs propriétés. Les conditions relevées dynamiquement deviennent les sous-buts à résoudre pour pouvoir effectuer le déplacement, comme si elles étaient dans le code de l'interaction. A leur tour ces conditions entraîneront une planification : ce mécanisme permet d'intégrer à la planification les déplacements et leurs contraintes (porte, piège...).

Les déplacements ne sont pas des interactions comme les autres : leur exécution n'est pas atomique comme celles des autres interactions : l'exécution d'un déplacement est réalisée par plusieurs déplacements atomiques d'un pas, correspondant au franchissement d'un arc dans le graphe représentant l'environnement. A la différence d'un déplacement de type téléportation (par simple mise à jour de la position d'un agent pour exécuter le déplacement), la simulation *pas à pas* d'un déplacement peut aboutir à un échec, par exemple si l'agent rencontre une porte verrouillée. L'agent dispose initialement de cette connaissance, mais il ne peut en tirer parti que s'il intègre les déplacements dans la planification, *ainsi que les conditions permettant ce déplacement*. Ce point est important dans notre approche car il permet d'anticiper au moment de la planification des problèmes qui peuvent survenir à l'exécution d'un déplacement. Mon approche permet d'intégrer les conditions permettant le déplacement dans la planification. Ainsi, si le déplacement est conditionné par la possession d'une clef pour déverrouiller une porte, l'agent ira chercher la clef avant de se rendre à la porte, tirant partie au plus tôt de cette connaissance sur l'environnement.

Le moteur de comportement des agents raisonne sur une mémoire, ce qui permet d'introduire d'autres agents animés dans la simulation tout en conservant des comportements crédibles : les agents utilisent des connaissances qui sont potentiellement obsolètes, ce qui les amène à commettre des erreurs qui sont souhaitables pour le réalisme de la simulation. Le module de mise à jour et de replanification corrige localement le plan afin d'intégrer l'information qui a causé l'échec de l'exécution du plan initial. A l'aide des outils dont nous disposons, il est possible de créer des simulations comportant plusieurs agents animés. Nous allons voir maintenant une solution permettant de les intégrer dans une équipe afin de résoudre un problème commun.

sans intégration des déplacements :



avec intégration des déplacements :

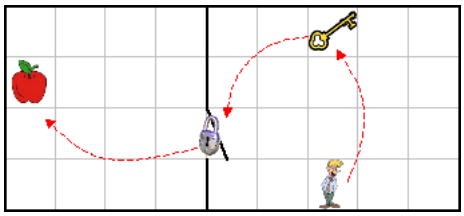


FIG. 4.11 – Importance de l'intégration des déplacements dans la planification : si l'on ne considère pas les déplacements au moment de la planification, on ne tire pas partie de toutes les informations disponibles. On planifie alors le déplacement passant par la porte alors que celle-ci est verrouillée. La rationalité du comportement observé est dégradée.

4.4 La collaboration

La résolution de certains problèmes fait appel à des capacités très variées qu'il sera rare de trouver réunies dans un unique agent. Pour résoudre ces problèmes, plusieurs agents doivent donc intervenir afin que toutes les capacités nécessaires soient réunies. L'utilisation de plusieurs agents pour résoudre un même problème introduit plusieurs difficultés : en effet, il est nécessaire de coordonner les agents afin que chacun d'entre eux puisse connaître son rôle dans la résolution du but commun. Nous montrons qu'un agent n'a pas la possibilité de calculer les actions qu'il doit effectuer si elles n'ont pas un rapport direct avec le but de l'équipe. Un second problème se pose lors de l'exécution du plan par les agents : il est nécessaire d'organiser leurs actions afin qu'elles soient exécutées dans un ordre cohérent par rapport au plan global de l'équipe. Ces deux points requièrent la coordination des agents. Nous avons choisi de réaliser cette coordination grâce à un agent particulier, le chef, qui établit le plan d'équipe et distribue les tâches aux agents. Cette centralisation du calcul du plan pose toutefois le problème de l'autonomie des agents : l'approche entièrement centralisée les réduit à de simples outils exécutant les ordres reçus du chef, tandis que le chef subit toute la charge de calcul représentée par l'élaboration du plan d'équipe. Nous proposons une méthode permettant de répartir l'effort de planification entre les agents afin que chacun participe à la construction du plan global. Cette méthode s'appuie sur l'hypothèse d'agents impliqués à 100% dans l'équipe, ce qui n'est pas toujours le cas comme nous l'avons vu en 2.3.

4.4.1 Planification en équipe

Une équipe d'agents (figure 4.12) est un regroupement d'agents dont l'ensemble des comportements tend à satisfaire un but global B. Les agents a_i composant l'équipe n'ont pas nécessairement connaissance de B. Chaque agent ne connaît que le but b_i ordonné par le

> *« L'important quand on est chef n'est pas d'avoir tort ou raison, mais d'être catégorique. »*
> T. Pratchett, Le grand livre des gnomes.

chef, qui permet d'avancer dans la résolution du but B. Bien que les membres de l'équipe ne connaissent pas le but B, il faut quand même que cette information soit présente dans l'équipe, ainsi que le plan global permettant d'atteindre ce but. Nous avons décidé qu'un agent particulier dans l'équipe porterait ces informations : le chef. Ce dernier est en outre chargé d'établir le plan d'équipe, de distribuer des tâches à effectuer aux agents et de vérifier le bon déroulement de celles-ci. Il possède donc toutes les informations relatives à la constitution de l'équipe. De l'extérieur, une équipe peut être vue comme un agent dont les capacités sont la réunion des *peut-effectuer* des membres de l'équipe.

L'objectif est de gérer le plan de l'équipe d'une manière centralisée, sans pour autant tomber dans l'un des pièges de la centralisation où certains agents doivent exécuter des tâches primitives sans aucun raisonnement. Une vision complètement centralisée ne convient pas car le comportement obtenu n'est pas fidèle à la réalité que nous cherchons à simuler : c'est comme si un chef de chantier expliquait à un électricien comment dénuder un fil afin de poser une prise ! Notre approche s'efforce donc de conserver l'autonomie de décision des agents, afin de permettre à chaque agent de remplir pleinement son rôle, en utilisant ses capacités cognitives.

On donne un but à l'équipe par l'intermédiaire du chef. Celui-ci planifie alors en fonction des capacités de l'équipe, puis distribue les actions à effectuer aux membres en fonction de leurs capacités individuelles. Nous travaillons dans un environnement ne comportant qu'une équipe, et tous les agents animés présents dans l'environnement font partie de cette équipe, notamment le chef.

```
<team>          ::=  {<leader>, <members>, <can-perform>, goal}
<leader>        ::=  agent
<members>       ::=  agent*
<can-perform>   ::=  ⋃_i agent_i.can-perform
```

Fig. 4.12 – Définition d'une équipe.

A partir du but de l'équipe, le chef doit calculer un plan. Pour cela, il planifie en utilisant toutes les interactions que peuvent effectuer les agents qui composent l'équipe. Il établit ainsi un plan tirant profit des capacités de tous les membres de l'équipe. Il est nécessaire de limiter la planification du chef afin de délimiter le plan de l'équipe et celui des agents. Pour cela, on pourrait forcer le chef d'équipe à ne pas planifier au-delà d'une certaine profondeur fixée n. Si certaines branches de l'arbre de planification se terminent avant d'avoir atteint la profondeur n, alors la feuille obtenue est une action élémentaire qui ne peut plus être décomposée. Le chef ordonne alors à un agent membre de l'équipe d'exécuter cette tâche, et les agents ont un rôle limité à l'exécution de tâches primitives déterminées par le chef. Cela revient à considérer les membres de l'équipe non comme des agents mais comme des outils commandés à distance, sans autonomie de décision. Si par contre, une branche atteint la profondeur n alors que la planification de cette branche n'est pas terminée, le chef arrête la planification de cette branche.

La tâche portée par la feuille obtenue est donnée comme but à un agent membre de l'équipe. Il sera chargé de terminer le travail de planification pour résoudre ce but. Il est possible que le but donné à l'agent soit trop abstrait et que ses connaissances ne lui permettent pas de résoudre ce but. Une difficulté réside dans le fait qu'on ne peut pas savoir avant d'avoir planifié quelle sera la profondeur du plan. Pour pouvoir déterminer la profondeur de planification n qui laisserait suffisamment d'autonomie aux agents sans leur donner de tâche trop abstraite, il faudrait d'abord planifier entièrement... De plus, la profondeur de l'arbre est rarement identique pour toutes les branches.

Nous proposons une solution qui permet d'arrêter la planification dès que le travail peut être délégué à un agent, grâce à la répartition de la connaissance.

Répartition de la connaissance

Le principe que nous adoptons est que le chef connaît les actions à accomplir pour exécuter une tâche complexe, mais ne sait pas comment effectuer ces actions. Il en délègue la réalisation à des spécialistes. Pour empêcher le chef de planifier trop profondément, on peut donc limiter l'étendue de ses connaissances relatives aux capacités des membres de l'équipe pour qu'il arrête naturellement sa planification sans empiéter sur leur travail(figure 4.13). Les membres de l'équipe sont chargés de poursuivre la planification en fonction de leurs capacités, ce qui leur laisse une autonomie de décision : ils peuvent s'adapter à la configuration de l'environnement, contrairement à des agents qui recevraient un plan préétabli. Nous avons exposé le principe de la répartition de la connaissance et de l'autonomie des agents dans [DMR05c].

Pour réaliser cela, lors de la création de l'équipe, on ajoute aux connaissances du chef toutes les interactions que peuvent effectuer les agents, après avoir effacé les conditions et les gardes de ces interactions. De cette manière, le chef peut utiliser ces interactions dans la planification, mais cela entraîne l'arrêt de la planification : le chef sait à quoi servent ces interactions, mais il ne sait pas comment les réaliser. Lors de l'exécution du plan par le chef, les tâches qu'il ne sait pas effectuer sont distribuées aux agents. Ceux-ci se réfèrent à leur connaissance de l'interaction et établissent un plan qui tient compte des conditions et de la garde de l'interaction. Dans l'expression des *peut-effectuer* des agents et du chef, la dynamique de la simulation apparaît à nouveau : on voit immédiatement les ordres qui peuvent être donnés à un agent par son chef.

Pour mieux comprendre ce découpage, prenons un exemple constitué d'une équipe simple composée d'un chef a_0, et de deux agents (identiques) a_1 et a_2. Leur connaissances sont présentées dans la table 4.3. Nous voyons que certains des *peut-effectuer* des agents de l'équipe sont inclus aux connaissances du chef sans leurs conditions ni leurs gardes.

On donne au chef le but p_0. Pour résoudre ce but, le moteur de planification détermine qu'il faut utiliser l'interaction I_0, qui possède le but p_0 souhaité dans sa partie *actions*. Pour pouvoir exécuter I_0, il faut d'abord résoudre les conditions p_1 et p_2. Dans les *peut-effectuer* du chef, aucune interaction ne peut résoudre ces conditions. Par contre, les interactions I_1 et I_2 présentes dans les *peut-effectuer* de l'équipe permettent de résoudre respectivement les conditions p_1 et p_2. Le moteur de planification du chef ajoute donc ces interactions dans l'arbre de planification. Comme ces deux interactions ne possèdent pas de *condition* (dans les connaissances du chef), la planification s'arrête automatiquement.

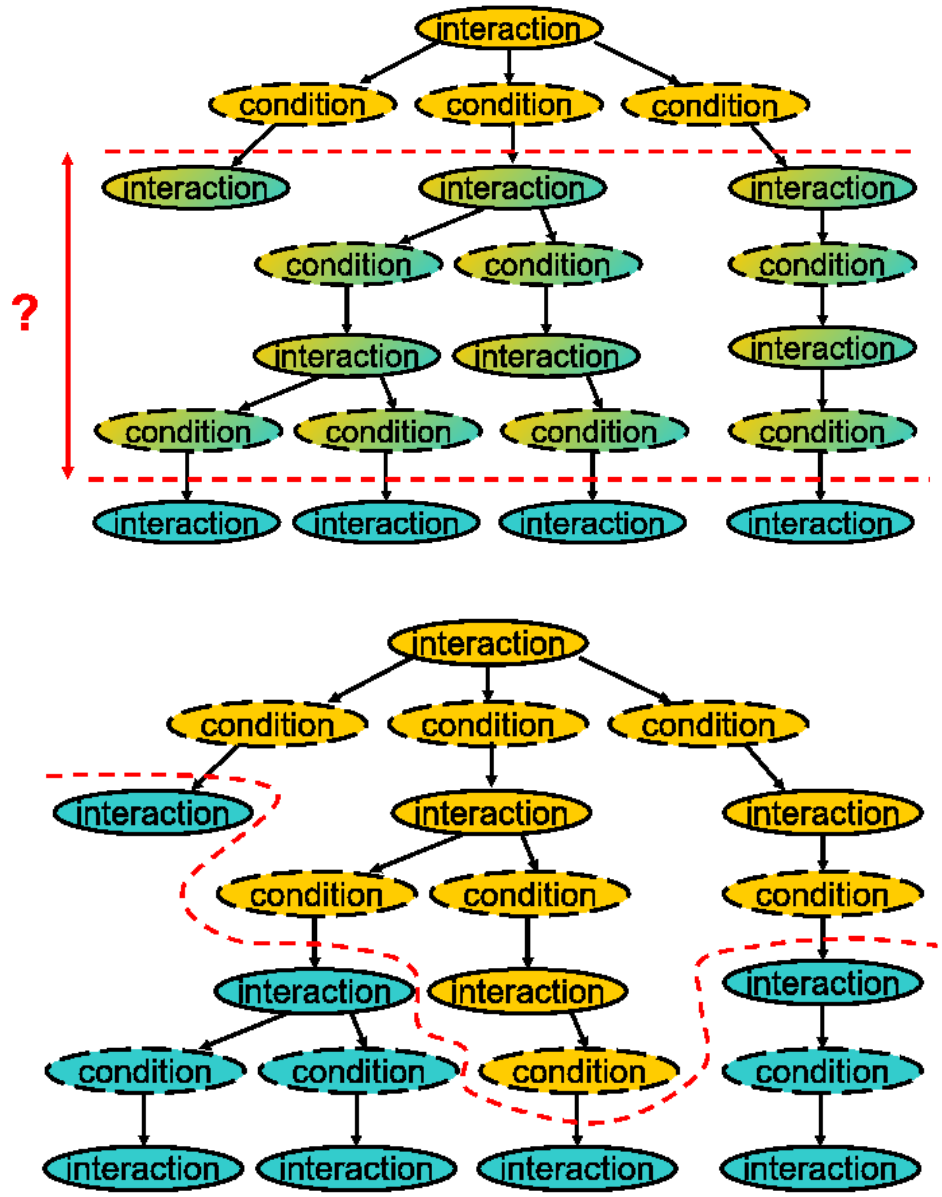


FIG. 4.13 – Répartition de la connaissance. Pour donner une autonomie de décision aux agents de l'équipe, le chef doit arrêter la planification à une certaine profondeur. Il est difficile de fixer cette profondeur arbitrairement (schéma du haut) : si la profondeur choisie est trop grande, les agents ont un simple rôle d'exécutant. Si elle est trop faible, les agents se verront peut-être attribuer une tâche trop abstraite qu'ils ne savent pas résoudre. Il est nécessaire d'adapter la profondeur de la planification aux problèmes (schéma du bas) : ceci est possible grâce à la répartition de la connaissance : le chef s'arrête de planifier dès qu'il utilise une interaction connue par l'un des agents. Il est alors du ressort de celui-ci de poursuivre la planification.

	chef.p-e	equipe.p-e	
nom	I_0	I_1	I_2
conditions	p_1, p_2	true	true
garde	G_1	true	true
actions	p_0	p_1	p_2

	connaissances des agents				
nom	I_1	I_2	I_3	I_4	...
conditions	p_3, p_4, p_5	p_6, p_7			
garde	G_2	G_3			
actions	p_1	p_2	p_3	p_4	

TAB. 4.3 – Connaissance du chef (tableau de gauche) et des agents (tableau de droite) : le chef peut-effectuer I_0 et possède donc la connaissance complète sur cette interaction. En revanche, il ne connaît que la partie *actions* des autres interactions (I_1 et I_2), dont la version complète n'est connue que de certains membres de l'équipe. La dynamique de la simulation transparaît à nouveau de la description des connaissances : à la simple vue de ces interactions, on peut déduire les ordres qui peuvent être donnés aux agents par leur chef.

Lors de l'exécution du plan, les interactions utilisées qui ne sont pas dans les *peut-effectuer* du chef sont distribuées à des agents capables de les réaliser. L'agent a_1 reçoit l'ordre d'exécuter l'interaction I_1, et donc établir un plan pour y parvenir. Il utilise cette fois sa connaissance concernant l'interaction I_1 : il prend alors en compte les *conditions* p_3, p_4 et p_5 de l'interaction dont le chef n'avait pas connaissance. Pour résoudre ces conditions, il utilise les interactions I_3, I_4 et I_5. L'agent a_2 reçoit quant à lui l'ordre d'exécuter l'interaction I_2. Il se réfère à sa connaissance concernant cette interaction, et doit résoudre à son tour les conditions p_6 et p_7. Ce schéma de planification répartie est illustré par la figure 4.14.

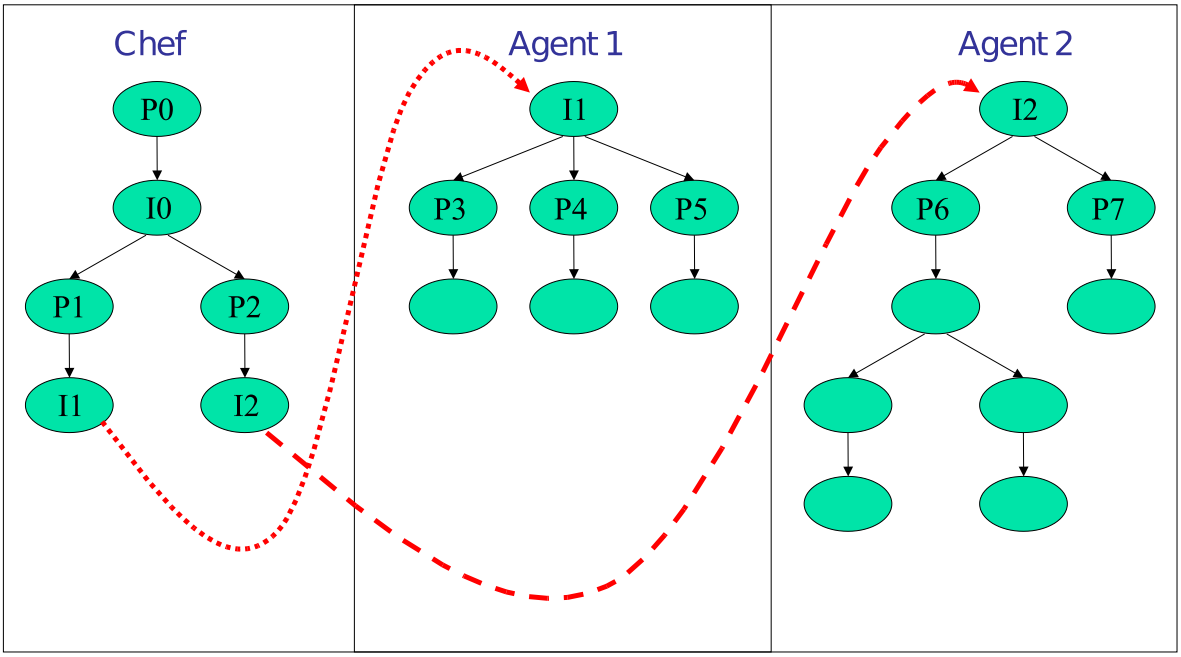


FIG. 4.14 – Le chef établit un plan abstrait : il utilise une version incomplète des interactions I_1 et I_2. Il distribue ensuite des tâches aux agents, et ceux-ci se réfèrent à la version complète de l'interaction afin de calculer un plan. Le travail de planification est partagé entre les agents et le plan d'équipe permettant d'atteindre le but initial est réparti : le plan complet n'apparaît jamais.

Dans les organisations humaines, il est rare de trouver des entités formées uniquement par un chef et des subalternes. Souvent, les subalternes délèguent une partie de leur travail, ou le chef est lui-même subalterne d'un autre chef. Ces organisations sont formées de hiérarchies à plusieurs niveaux. Construire une telle équipe d'équipes avec notre modèle est très simple. En effet un chef d'équipe est un agent comme les autres. Lorsqu'il s'enregistre dans une autre équipe E en tant que membre, il fournit la liste des capacités de son équipe (y compris les siennes). Le chef de l'équipe E n'a pas besoin de savoir si les capacités déclarées sont celles de l'agent ou des membres de son équipe si c'est un chef. Lorsque le chef de l'équipe E donne des buts à exécuter aux agents, il est possible que ceux-ci délèguent toute ou partie de la résolution du but, de manière récursive. Cela ne change rien au niveau de l'équipe E. Plutôt que d'une organisation hiérarchique, on peut parler dans le cas présent d'holarchie.

Abstraction de la situation

La prise en compte de la position des agents dans l'environnement complexifie la planification. En effet, certains problèmes que l'agent doit résoudre pour atteindre ses buts sont dus à des contraintes spatiales. Il en découle une autre manière de répartir l'effort de planification entre le chef et les membres de l'équipe. Le chef établit un plan qui ne tient pas compte des gardes de distance, omettant ainsi le caractère situé de la simulation et toutes les contraintes induites par les déplacements. En effet, si on ne tient pas compte des gardes de distance, la planification n'utilise jamais l'interaction de déplacement et ne tient donc pas compte des conditions imposées par les obstacles dans l'environnement. On se retrouve alors dans le cas d'une planification classique. L'*abstraction de la situation* est un cas particulier de la *répartition de la connaissance* dans lequel on ne cache que les gardes au chef. Le but de cette méthode est d'isoler le caractère situé pour qu'il soit géré uniquement par les agents : c'est dans la résolution des contraintes spatiales que les agents trouvent leur autonomie de planification, car les déplacements obligent parfois à élaborer un plan. Ainsi, le plan établi par le chef est valable quelle que soit la configuration spatiale de l'environnement, et les membres de l'équipe sont chargés de résoudre toutes les contraintes spatiales imposées par l'environnement, en fonction de la configuration de celui-ci.

Une fois le plan abstrait calculé par le chef d'équipe, les tâches sont données aux agents de l'équipe qui sont chargés de les exécuter. Or ces agents se réfèrent à leurs propres connaissances concernant l'interaction qu'ils doivent exécuter. Cette connaissance comprend cette fois les gardes de distances. Chaque agent établit alors un plan permettant de résoudre la garde, prenant en compte la topologie de l'environnement et les positions des cibles impliquées dans le plan alors que ces aspects avaient été écartés par le chef. Le plan établi est alors adapté à la géographie de l'environnement dans lequel l'agent se trouve alors que le plan du chef est abstrait par rapport à la situation.

Discussion

Lorsque l'agent chef A utilise une interaction d'un membre B de l'équipe, il est nécessaire qu'il stoppe sa planification : continuer est doublement inutile, d'une part parce que l'agent A ne saura pas exécuter le plan qu'il a calculé, et d'autre part parce que l'agent B va calculer la même chose. Plusieurs solutions s'offrent pour arrêter la planification : une solution très

simple aurait permis d'obtenir le même résultat : il suffit d'arrêter la planification lorsque le chef A utilise une des interactions d'un agent de son équipe. Notre solution est plus élégante sur plusieurs points. L'arrêt de la planification aurait nécessité la modification de l'algorithme de planification pour « surveiller » et déclencher un éventuel arrêt. La manière que nous avons utilisée (masquage des conditions et gardes) permet à la planification de s'arrêter d'elle même, sans modification de l'algorithme. De plus, la solution que nous présentons est justifiée en terme de similarité de fonctionnement avec la réalité que nous souhaitons simuler. En effet, dans les organisations humaines, on rencontre souvent des hiérarchies dans lesquelles le supérieur n'a pas toujours les capacités techniques pour exécuter les tâches de « bas-niveau », contrairement à son subalterne. L'utilisation de cette méthode permet de bien séparer les rôles organisationnels et les rôles exécutifs, et l'on peut deviner rapidement ces rôles en observant les connaissances d'un agent.

4.4.2 Affectation des actions aux agents

Une fois la planification effectuée, le chef doit répartir les actions à exécuter entre les agents de manière à ce qu'un maximum d'actions puissent être effectuées simultanément par les agents afin de réduire le temps nécessaire à l'exécution de la totalité du plan. Il doit aussi gérer dynamiquement les affectations dans l'équipe afin de toujours exécuter un maximum d'actions simultanément.

Affectation initiale

Les actions à distribuer sont toujours situées au niveau des feuilles de l'arbre constituant le plan du chef, par construction. Cette répartition est aisée lorsque l'équipe est constituée d'agents identiques (i.e. ayant les mêmes capacités *peut-effectuer*). En effet, dans ce cas, il suffit d'attribuer de manière gloutonne les buts aux agents, jusqu'à ce qu'il n'y ait plus d'agent libre ou de but à affecter.

Dans le cas contraire (équipe hétérogène), la répartition gloutonne des tâches ne donne pas de résultat satisfaisant. Le problème de répartition des tâches entre les agents est un problème d'affectation que l'on résout habituellement à l'aide de l'algorithme de Ford-Fulkerson [FF56] ou un de ses dérivés optimisés comme l'algorithme de Edmonds Karp [EK72] qui permettent de trouver le flot optimal dans un réseau de transport en recherchant des chaînes augmentantes. Nous définissons le problème d'affectation comme suit :

- soit $\mathcal{A}$ l'ensemble des agents a_i composant l'équipe
- soit σ_i l'ensemble des capacités de l'agent a_i
- soit $\mathcal{T}$ l'ensemble des tâches t_j que l'on peut effectuer dans l'environnement
- soit $\mathcal{B} \subset \mathcal{T}$ l'ensemble des buts b_k des agents

Pour résoudre le problème d'affectation, on peut le ramener à un problème de recherche du flot maximal dans un réseau de transport (figure 4.15). Pour cela, on crée un nœud S (source du réseau de transport), et on y relie autant de nœuds N_{a_i} qu'il y a d'agents a_i dans $\mathcal{A}$. On crée ensuite autant de nœuds N_{b_k} qu'il y a de buts b_k dans $\mathcal{B}$ et on les relie ensuite à un nouveau nœud P (puits du réseau de transport). On crée ensuite un arc entre un nœud N_{a_i} et un nœud N_{b_k} si $b_k \in \sigma_i$, c'est-à-dire si l'agent a_i sait effectuer la tâche b_k. On donne ensuite

à tous les arcs une capacité 1. De cette manière, si un flot de S vers P passe par un agent a_i, l'arc (S, N_{a_i}) sera immédiatement saturé (flot de 1 sur un arc de capacité 1) ce qui interdit de sélectionner cet agent à nouveau. Pour la même raison un seul flot de S vers P pourra passer par chaque nœud N_{b_k}, ce qui interdit d'affecter plusieurs fois un même but.

	b_0	b_1	b_2	b_3
a_0	X			X
a_1	X		X	
a_2	X	X		
a_3			X	X
a_4		X		X

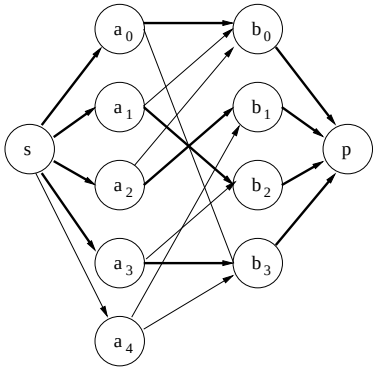


FIG. 4.15 – Transformation d'un problème d'affectation en problème de flot maximal dans un graphe. Dans le tableau, une croix dans la case de coordonnées (a_i, b_k) signifie que $b_k \in \sigma_i$. A chaque croix du tableau correspond un arc central dans le graphe. Les arcs en trait fort représentent le flot trouvé par l'algorithme. Un arc fort entre un nœud N_{a_i} et un nœud N_{b_k} représente l'affectation de la tâche b_k à l'agent a_i. On remarque dans cet exemple qu'il n'est pas possible d'affecter une tâche à tous les agents. Cela peut se produire même si l'on a autant d'agents que de tâches.

Une chaîne augmentante de S vers P calculée par l'algorithme de Ford-Fulkerson peut avancer sur un arc non saturé (ici avec un flot nul) et le saturer, ou reculer sur un arc saturé (ici avec un flot de 1) et le désaturer. La recherche des chaînes augmentantes se fait de manière incrémentale : à chaque fois que l'on trouve une chaîne augmentante, on obtient une affectation supplémentaire. Dans le graphe de flot, les affectations sont représentées par tous les liens saturés entre un nœud N_{a_i} et un nœud N_{b_k}.

Réaffectation

A tout moment de la simulation, la liste des agents disponibles et la liste des buts à effectuer est susceptible d'évoluer. Quatre cas de figure peuvent se présenter :

- *nouveau but :* lorsqu'un nouveau but b_l est ajouté, il y a peut-être un agent a_i qui est libre.
- *but supprimé :* quand un but est retiré (parce qu'il est terminé ou qu'il n'est plus nécessaire), l'agent a_i qui était chargé d'exécuter ce but devient libre. Il y a peut-être un but b_l qui n'est pas alloué actuellement.
- *nouvel agent :* lorsqu'un agent a_i est ajouté à l'équipe, ou lorsqu'il vient de terminer une tâche, il y a peut-être un but b_l qui n'est pas alloué actuellement.
- *agent supprimé :* quand un agent a_j qui exécutait le but b_l est retiré de l'équipe (par exemple parce qu'il a été détruit), il y a peut-être un agent a_i qui est libre.

Chacun de ces cas peut donc se rapporter à un problème dans lequel l'agent a_i est libre et le but b_l non affecté. Si $b_l \in \sigma_i$, a_j peut exécuter le but b_l et le problème est résolu simplement. Dans le cas contraire, l'agent a_i peut remplacer un autre agent a_k qui exécute actuellement le but b_m si $b_m \in \sigma_i$ et $b_l \in \sigma_k$. Une succession de remplacements est aussi possible. Il s'agit donc de trouver une nouvelle affectation des agents qui permettent (1) d'allouer un maximum de buts en (2) effectuant le minimum de changements d'affectation. En effet, chaque changement est "coûteux" dans la mesure où l'agent doit se déplacer pour atteindre la cible de son nouveau but, et qu'il n'est pas directement utile à l'avancement du but pendant son déplacement.

La propriété incrémentale de l'algorithme de Ford-Fulkerson nous permet de prendre en compte ces deux points. En effet, il suffit d'ajouter les nœuds (agent ou but) concernés, ou de désaturer le flot correspondant à une affectation qui n'a plus lieu d'être. La recherche d'une chaîne augmentante de S vers P dans le graphe modifié permet de trouver immédiatement une nouvelle affectation. Selon le théorème de Ford-Fulkerson, s'il n'existe pas de chaîne augmentante alors le flot est maximal, et par conséquent on ne peut pas affecter plus d'agents. Toutefois, cet algorithme ne permet pas de garantir que le nombre de changements d'affectation est minimal. Pour cela, on utilise l'algorithme de Edmonds Karp qui recherche des chaînes augmentantes de taille minimale : en effet si la chaîne augmentante est de taille minimale, il y a un nombre minimal d'arcs allant d'un nœud but vers un nœud agent. Lorsqu'un tel arc est présent dans la chaîne augmentante, on diminue le flot porté par cet arc, ce qui revient à enlever l'affectation d'un agent.

Conclusion

Le moteur de comportement que je propose, associé au modèle centré interaction, est prévu pour fonctionner dans trois modes d'interaction. Le premier mode ne fait appel qu'à un seul agent animé : il permet de valider le comportement individuel d'un agent et sa capacité à résoudre un problème à l'aide des interactions données. Dans le second mode, l'intervention d'autres agents animés introduit la non-monotonie de l'environnement. Les informations perçues à un moment donné par un agent peuvent devenir obsolètes à cause du comportement des autres agents animés présents dans l'environnement. Le raisonnement sur la mémoire renforce la crédibilité de la simulation, en amenant l'agent à faire des erreurs à cause de ses croyances. Le module de replanification intervient alors pour réparer localement le plan pour que l'agent ne soit pas bloqué par l'erreur qu'il a commise. Enfin, le troisième mode concerne les équipes : lorsque plusieurs agents sont nécessaires pour résoudre un problème, le chef établit un plan abstrait pour l'équipe en se basant sur les capacités des membres. Ces derniers peuvent alors utiliser leur moteur de comportement pour poursuivre le raisonnement et s'adapter en fonction de la situation à laquelle ils sont confrontés. En fonction des variations de la composition de l'équipe, le chef peut redistribuer les tâches de l'équipe afin de maximiser le nombre d'agents utilisés et d'accélérer ainsi l'exécution du plan. Il est bien sûr possible de mixer le mode de concurrence et le mode de collaboration : les agents sont capables de s'adapter à la présence d'autres agents, que ceux-ci fassent partie de leur équipe ou non.

Le fonctionnement global de nos équipes d'agents autonomes (modèle centré interaction, dualité peut-subir/peut-effectuer, représentation de la connaissance, intégration des déplacements, moteur de comportements individuel et collectif, replanification, répartition et réallocation des tâches) a donné lieu à une publication [DMR05b] et un chapitre de livre [DMR06].

Chapitre 5

La réalisation

5.1 La plate-forme de simulation

Le travail que je viens de présenter a abouti à la création d'un environnement de simulations présenté en figure 5.1. Cet environnement de simulation a deux vocations. La première est de mettre à l'épreuve les différentes propositions de mon travail afin de les valider. La seconde est de fournir des outils pour développer des comportements et les tester, afin par exemple d'aider le game designer dans la création de comportements.

Au démarrage de la plate-forme, on accède à la partie *création* qui permet de construire la simulation. On peut y créer des interactions, des agents et des cartes ou manipuler les éléments déjà présents dans la bibliothèque. On ajoute ensuite les agents sur la carte afin de créer l'environnement, sans oublier de donner un but aux agents animés. On peut alors lancer le simulateur et observer le comportement de l'agent. A tout moment de la simulation, on peut ajouter, supprimer ou déplacer des agents. On peut aussi ajouter ou supprimer des possibilités d'interaction en intervenant sur les *peut-subir* et les *peut-effectuer* des agents. On peut alors observer la réponse des agents, mais aussi connaître leur « état mental » grâce à la visualisation de leur plan et de leur mémoire, ce qui permet d'analyser leur comportement.

Le cœur de la plate-forme (implémentation des propositions...) a été programmé par Jean-Christophe Routier et moi-même, tandis que l'interface graphique permettant la création des simulations a été programmée par Alexandre Vandenabelle et Yoann Vue, étudiants en DESS IAGL dans le cadre de leur projet. La plate-forme réalisée en $Java^{TM}$ est actuellement composée de 476 classes (dont 311 sont consacrées à l'interface graphique) regroupées en 36 paquetages. Elle n'intègre pas encore la gestion des équipes.

L'étude de comportements grâce à cet environnement se déroule en deux phases : la création et l'exécution de la simulation.

5.1.1 Création d'une simulation

Nous avons vu précédemment qu'une simulation était composée de plusieurs éléments : les interactions, les agents et l'environnement. Dans une première phase, il est nécessaire de créer ces différents composants, qui sont ajoutés à une bibliothèque afin de pouvoir être réutilisés.

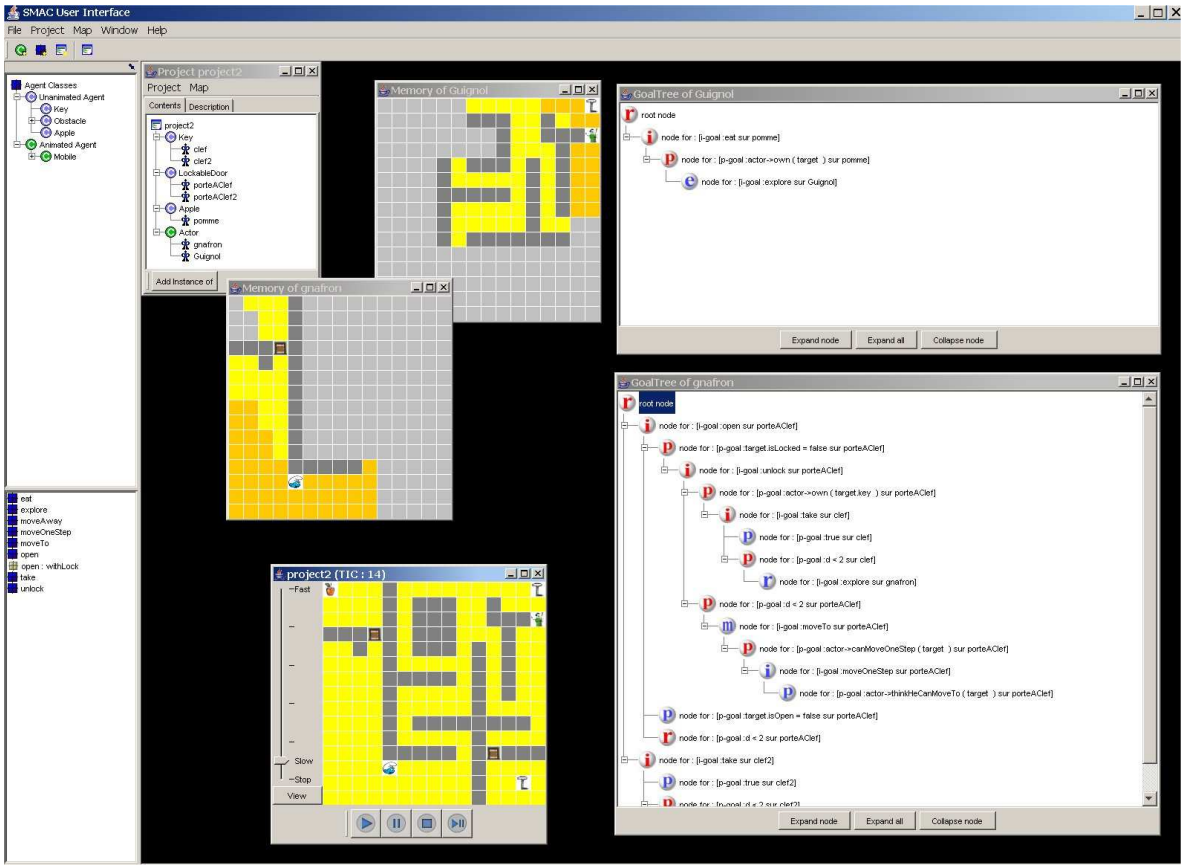


FIG. 5.1 – L'interface de simulation permet de créer des interactions, des agents et des environnements puis de lancer une simulation. On peut alors observer l'environnement, les mémoires des agents ainsi que leurs plans afin d'analyser leur comportement.

Création d'une interaction

La création d'une interaction est très simple : elle consiste à en fournir le code en renseignant les différents champs (figure 5.2). Conformément au modèle, le code des interactions n'est pas lié à des agents mais à un acteur et à une cible désignés respectivement par *actor* et *target*. Un onglet permet de donner une description de l'interaction.

Création d'un agent

Au sein de la plate-forme, les agents sont organisés en classes afin d'en faciliter la conception. Un mécanisme objet permet d'hériter des caractéristiques d'une superclasse afin de ne pas redéfinir certaines caractéristiques. A la base, deux classes d'agents sont disponibles : *Animated* et *Unanimated* correspondant respectivement aux agents animés et agents inanimés du modèle. Les premiers sont dotés du moteur de comportement décrit en 4.3. Une sous-classe de la classe *Animated* est la classe *Mobile* qui désigne les agents animés capables de se déplacer. Il est possible de définir des nouvelles sous-classes d'agents et de leur ajouter des propriétés, en particulier les *subjected interactions* et les *performed interactions* (figure 5.3) correspondant

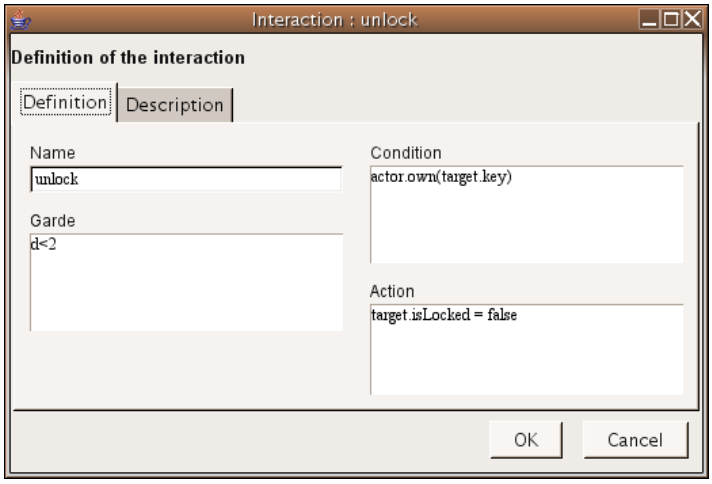


FIG. 5.2 – Création d'une interaction.

aux *peut-subir* et *peut-effectuer* du modèle. Les interactions sont choisies dans la bibliothèque. Il existe une classe d'agent particulière : la classe *Obstacle*. Celle-ci étend la classe *Unanimated* en y ajoutant deux propriétés particulières : *inCondition* et *outCondition*. Ces deux propriétés permettent de spécifier les conditions qui seront portées respectivement par les arcs entrants et sortants de la place sur laquelle sera placée un agent de cette classe. Par exemple, pour simuler une porte, on crée un agent *door* héritant de la classe *Obstacle* dont la *inCondition* est $target.isOpen = true$.

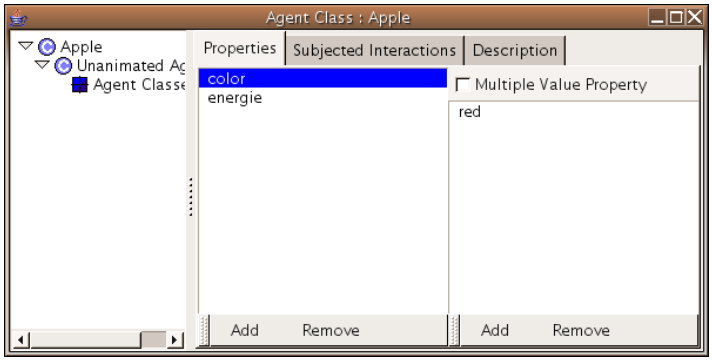


FIG. 5.3 – Création d'une classe d'agent.

Création d'un environnement

Dans la plate-forme, les environnements sont de type « matrice 2D » : on dispose d'une grille dans laquelle on peut placer des murs afin de représenter des bâtiments, des pièces ou des chemins dans un labyrinthe. Le graphe sous-jacent est automatiquement créé.

Création d'un projet

La création d'un projet est la phase ultime précédant le lancement de la simulation. Un projet regroupe un environnement et des agents. Pour commencer, on crée des instances d'agents (figure 5.4) à partir des classes de la bibliothèque. On peut alors personnaliser le comportement de l'agent en lui ajoutant ou en lui retirant des propriétés. On peut en particulier ajouter ou retirer des interactions à la liste de ses *peut-subir* et de ses *peut-effectuer* dans le cas d'un agent animé. L'onglet *goal* permet de donner un ou plusieurs buts aux agents animés.

Une fois les instances d'agent créées, on crée ou on charge un environnement et on y place les agents. Lorsque l'on ajoute un agent de la classe Obstacle sur une place, les conditions portées par les arcs allant de et vers la place sont mis à jour respectivement en fonction de la *outCondition* et *inCondition* de l'agent.

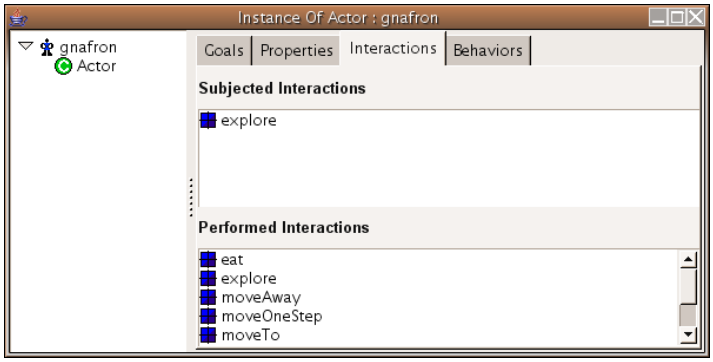


FIG. 5.4 – Création d'une instance d'agent.

La simulation va pouvoir commencer...

5.1.2 Lancement d'une simulation

Une grande partie de l'intérêt de la plate-forme se situe bien évidemment ici. Nous pouvons confronter des agents à des situations pour visualiser leur comportement, et éventuellement intervenir dans la simulation pour la perturber et observer la réponse de l'agent. Dans le simulateur présenté en figure 5.1, plusieurs fenêtres de visualisation sont présentes :

- *fenêtre environnement* : elle permet d'afficher l'*environnement réel* à un instant donné. Il s'agit de l'environnement que l'on a créé et dans lequel on a placé les agents, mis à jour en fonction de l'évolution de la simulation. C'est dans cette fenêtre que l'on peut observer le déroulement de la simulation. Pour rappel, les agents, disposant d'un halo de vision, n'ont pas connaissance de tout ce qui se trouve dans cette fenêtre, destinée au créateur de la simulation.
- *fenêtre mémoire* : il en existe autant que d'agents animés présents dans la simulation. Cette fenêtre montre l'*environnement tel qu'un agent pense qu'il est*. Des parties grisées matérialisent les zones inconnues de l'environnement. Ces zones sont présentes généralement au début de la simulation lorsqu'un agent n'a pas encore exploré tout l'envi-

ronnement. Les parties orangées matérialisent le halo de vision de l'agent à un instant donné : on peut ainsi « voir avec les yeux de l'agent » et comprendre quelles sont les informations qui entraînent une modification de son plan. Enfin, les parties jaunes sont les parties de l'environnement qui ont été explorées au moins une fois par l'agent, mais qui ne sont pas actuellement dans son halo de vision. En comparant avec l'environnement réel, on peut comprendre pourquoi l'agent s'obstine à essayer d'atteindre une position de l'environnement qui ne contient rien d'intéressant : à cet endroit dans sa mémoire il y a peut-être l'objet convoité !

- *fenêtre arbre de buts* : il en existe autant qu'il y a d'agents animés. Cette fenêtre affiche le plan d'un agent à un moment donné. En visualisant le plan, on comprend toutes les étapes intermédiaires du raisonnement de l'agent et on peut faire le lien entre le but de l'agent et l'action qu'il est en train d'entreprendre.
- *fenêtre propriétés* : elle permet d'afficher l'état des propriétés d'un agent inanimé. Comme la modification d'une de ces propriétés est souvent l'un des buts de l'agent, on peut vérifier que ce but est atteint, ou en train de l'être.

Les différentes fenêtres sont mises à jour à chaque pas de la simulation afin de pouvoir visualiser l'*état mental* de l'agent.

La fenêtre *environnement* donne la possibilité de contrôler le déroulement de la simulation. On peut la démarrer, l'arrêter, la réinitialiser ou la faire fonctionner en mode « pas à pas ». Un curseur permet en outre de contrôler la vitesse de déroulement de la simulation.

5.2 Réutilisation des composants

Le modèle centré interaction a pour objectif de permettre la réutilisation des composants d'une simulation dans une autre. Nous allons donner dans cette section un exemple de réutilisation : pour cela, nous allons construire une première simulation dont nous réutiliserons une grande partie des composants pour construire la seconde. Nous verrons que les composants peuvent être utilisés par le moteur de comportement dans des contextes différents : le code d'une interaction est indépendant des agents et de la simulation dans laquelle il est utilisé.

5.2.1 Simulation 1

Voici la liste des agents en présence et la liste des interactions qu'ils peuvent subir ou effectuer :

interactions / agents (classe)	peut-subir	peut-effectuer
Gargamel (AnimatedAgent)		take go open eat drink
porte (Door)	open	
pomme (Apple)	take eat	

Voici le code des principales interactions :

nom	take	open
condition		target.isOpen = false
garde	dist(actor, target)<1	dist(actor, target)<1
action	add(actor.inventory, target) remove(target)	target.isOpen = true

nom	eat
condition	actor.own(target)
garde	
action	destroy(target) sub(actor.inventory, target)

Le placement des agents dans l'environnement est indiqué par la figure 5.5.

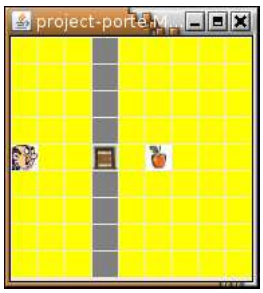


FIG. 5.5 – Environnement de la simulation 1

Le but de l'agent *Gargamel* est *eat* sur la cible *apple*. Il utilise ses connaissances sur les agents et les interactions pour établir le plan suivant :

```
eat apple
|-actor.own(apple)
  |-take apple
    |-dist(actor,apple)<1
      |-go apple
        |-door.isOpen=true
          |-open door
            |-dist(actor,door)<1
              |-go door
```

Ce plan donne lieu à l'exécution suivante : *go door*, *open door*, *go apple*, *take apple* et *eat apple*.

5.2.2 Simulation 2

On construit une seconde simulation en réutilisant les composants suivants :
– l'agent *Gargamel*
– la carte de l'environnement
– les interactions précédemment définies
On ajoute un agent *cola* à la simulation. Cet agent peut subir les interactions suivantes : *take*,*open* et *drink*. L'interaction *drink* dont le code est donnée ci-dessous est ajoutée à la simulation :

nom	drink
condition	actor.own(target) target.isOpen = true
garde	
action	destroy(target) sub(actor.inventory, target)

L'environnement de départ est donné en figure 5.6. Cette fois, le but de l'agent *Gargamel* est *drink* sur la cible *cola*.

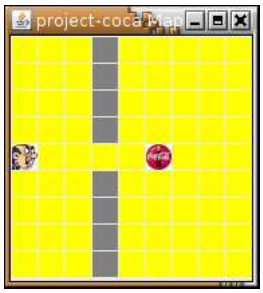


FIG. 5.6 – Environnement de la simulation 2

L'agent établit alors ce plan :

```
drink cola
|-actor.own(cola)
| |-take cola
|   |-dist(actor,cola)<1
|     |-go cola
|-cola.isOpen = true
  |-open cola
    |-dist(actor,cola)<1
```

Pour exécuter ce plan, l'agent effectue séquentiellement les interactions suivantes : *go cola*, *take cola*, *open cola* et *drink cola*.

Nous constatons que l'interaction *open* a été utilisée par l'agent dans un contexte différent de la première simulation : il s'agissait tout à l'heure d'ouvrir un agent qui constituait un

obstacle au déplacement de *Gargamel* tandis que dans cette seconde simulation, l'interaction *open* est utilisée pour permettre à l'agent de consommer une boisson. On peut construire une troisième simulation faisant intervenir à la fois l'agent *porte* et l'agent *cola* : l'interaction *open* sera utilisée deux fois par l'agent pour deux motifs différents.

Le modèle centré interaction et la représentation des connaissances séparant les interactions et les agents permettent de réutiliser les composants : entre ces deux simulations, l'agent *Gargamel* est resté inchangé, l'environnement également, ainsi que le code des interactions. Les seules différences sont l'ajout d'une interaction *drink* et d'un agent *cola* pouvant subir cette interaction. Le moteur de comportement de l'agent *Gargamel* n'a pas changé entre ces deux simulations, mais il a intégré les nouvelles données pour produire le plan.

5.3 Une simulation détaillée pas à pas

Cette section donne un exemple détaillé pas à pas de simulation utilisant deux agents évoluant en concurrence.

5.3.1 Situation initiale

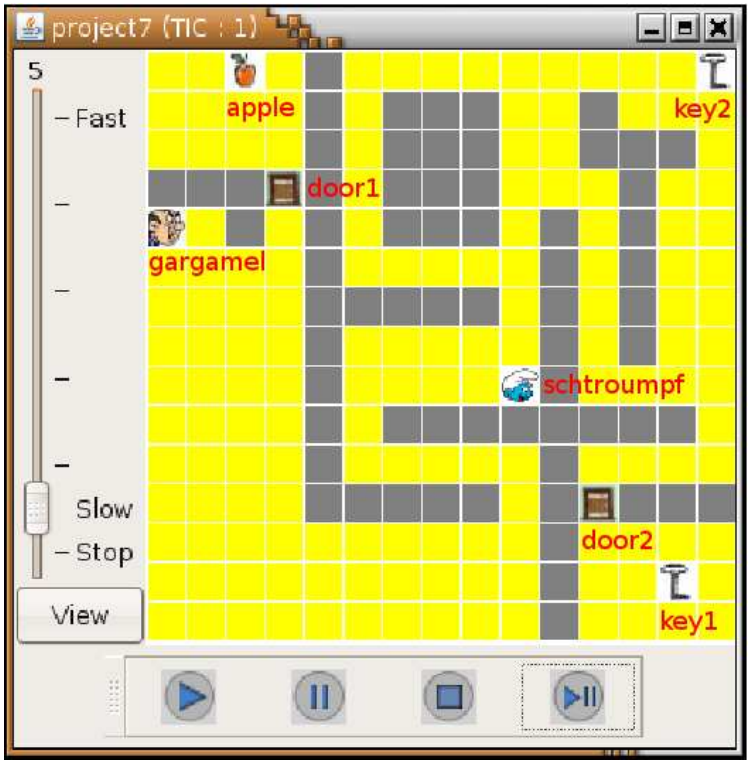


FIG. 5.7 – Environnement initial

La situation de la figure 5.7 met en scène deux agents (*gargamel* et *schtroumpf*) ayant tous deux pour but de manger la pomme *apple*. Celle-ci se trouve en haut à gauche de l'environnement, dans une pièce fermée par la porte *door1* dont la clef *key1* est dans la pièce en bas à droite. Cette pièce est condamnée par une porte *door2* dont la clef *key2* est en haut à droite de l'environnement. Au début de la simulation, les agents ne connaissent pas l'environnement (leur mémoire est vide). Il établissent le plan de la figure 5.8.

```
eat apple
|-actor.own(apple)
  |-explore
```

FIG. 5.8 – Plan 1 : plan initial des deux agents

Après quelques pas de simulation (figure 5.9), l'agent *gargamel* se trouve devant la porte *door1*. On peut voir en jaune la partie de l'environnement découverte par l'agent, en orange la partie qu'il voit actuellement, et en gris clair les zones qui ne sont pas encore explorées. L'agent *gargamel* découvre la pomme *apple*[23] et établit le plan de la figure 5.10. Il s'agit d'une replanification partielle : c'est la nouvelle information *new agent apple* qui a « réagi » avec le nœud *actor.own(apple)* et redéclenché la planification du sous-arbre.

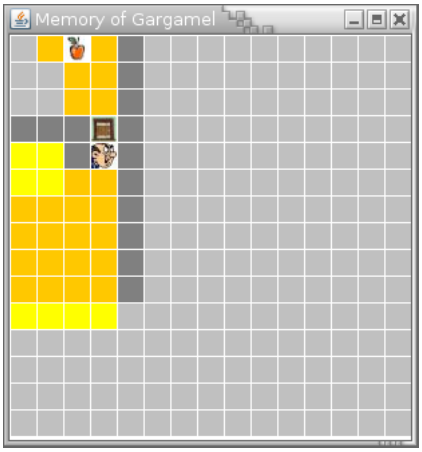


FIG. 5.9 – L'agent *gargamel* aperçoit la pomme *apple*.

5.3.2 Plan et interaction spécifique

Une fois que les agents ont tous les deux exploré la totalité de l'environnement, leur plan est celui de la figure 5.11. Ce plan, réalisé par des replanifications partielles successives à la découverte de chaque nouvel agent, exprime le raisonnement suivant : pour prendre la pomme *apple*, il faut pouvoir l'atteindre. Pour cela, il faut franchir la porte *door1* qui doit être ouverte.

[23] Nous considérons en effet que les agents sont transparents, à la différence des murs. Il est donc possible de connaître ce qui se trouve derrière une porte.

```
eat apple
|-actor.own(apple)
  |-take apple
    |-dist(actor,apple)<1
      |-go apple
        |-door1.isOpen=true
          |-open door1
            |-door1.isLocked=false
            | |-unlock door1
            |   |-actor.own(key1)
            |   | |-explore
            |   |-dist(actor,door1)<1
            |     |-go door1
            |-dist(actor,door1)<1
              |-go door1
```

FIG. 5.10 – Plan 2 : l'agent *gargamel* a découvert la pomme *apple*

La porte *door1* est une *LockableDoor* : elle possède une interaction *open* spécifique qui ajoute la condition *target.isLocked* = *false*. Pour être ouverte, une *LockableDoor* ne doit pas être verrouillée. L'agent utilise le code spécifique de cette interaction : il doit déverrouiller *door1* grâce à l'interaction *unlock*, dont une condition stipule qu'il doit posséder *target.key*, c'est-à-dire *key1*. Pour se procurer cette clef, il doit franchir la porte *door2* qui est quant à elle verrouillée grâce à la clef *key2*, qu'il peut atteindre.

Ce plan consiste en fait à exécuter séquentiellement ces actions : aller à *key2*, prendre *key2*, aller à *door2*, déverrouiller *door2*, ouvrir *door2*, aller à *key1*, prendre *key1*, aller à *door1*, déverrouiller *door1*, ouvrir *door1*, aller à *apple*, prendre *apple*, manger *apple*.

5.3.3 Déplacement/Suppression d'objets

Quelques pas de simulation plus tard, l'agent *gargamel* a ramassé la clef *key2*. La figure 5.12 présente les différentes vues. Dans la vue du haut, on observe l'état actuel de l'environnement : la clef *key2* a disparu car l'agent *gargamel* vient de la ramasser. On retrouve naturellement la même chose dans la mémoire de l'agent *gargamel* à gauche, tout au moins dans son halo de vision, représenté en orange. En revanche, il se méprend sur la position de l'agent *schtroumpf* : dans sa mémoire, il est à la dernière position connue, lorsqu'il était encore dans son halo de vision. La vue de droite présente la mémoire de l'agent *schtroumpf* : non seulement il se méprend sur la position de l'agent *gargamel* qu'il pense être à la dernière position à laquelle il l'a vu, mais surtout, pour lui la clef *key2* est encore présente dans l'environnement !

La simulation continue pour les deux agents. De son côté, l'agent gargamel continue d'exécuter son plan initial, qui réduit au fur et a mesure que les prémisses résolues deviennent vraies (figures 5.13) et 5.14.

Pendant ce temps, l'agent *schtroumpf* a tenté de continuer l'exécution du plan de la figure 5.11. Arrivé dans la situation de la figure 5.15, il s'aperçoit enfin que la clef *key2* a disparu

```
eat apple
|-actor.own(apple)
  |-take apple
    |-dist(actor,apple)<1
      |-go apple
        |-door1.isOpen=true
          |-open door1
            |-door1.isLocked=false
            | |-unlock door1
            |   |-actor.own(key1)
            |   | |-take key1
            |   |   |-dist(actor,key1)<1
            |   |     |-go key1
            |   |       |-door2.isOpen=true
            |   |         |-open door2
            |   |           |-door2.isLocked=false
            |   |           | |-unlock door2
            |   |           |   |-actor.own(key2)
            |   |           |   | |-take key2
            |   |           |   |   |-dist(actor,key2)<1
            |   |           |   |     |-go key2
            |   |           |   |-dist(actor,door2)<1
            |   |           |     |-go door2
            |   |           |-dist(actor,door2)<1
            |   |             |-go door2
            |   |-dist(actor,door1)<1
            |     |-go door1
            |-dist(actor,door1)<1
              |-go door1
```

FIG. 5.11 – Plan 3 : les agents ont exploré tout l'environnement

de l'environnement, ce qui l'empêche de terminer l'exécution de son plan. Il s'est bien fait schtroumpfer.

Conclusion

Cet environnement nous a permis d'éprouver certaines propositions du présent travail. Tout d'abord, nous avons ainsi pu tester conjointement la représentation des connaissances et le moteur de comportement. Ce dernier parvient à manipuler les connaissances pour calculer un plan, lequel est à son tour exécutable. Ainsi nous avons par exemple pu juger de la pertinence de l'intégration des déplacements dans la planification. Nous avons également pu montrer la réutilisabilité des composants entre les simulations. Cet outil peut maintenant permettre de tester différents comportements : on peut effectuer des changements dans l'environnement, ajouter ou retirer des possibilités d'interactions aux agents *pendant le déroulement de la simulation* et observer l'influence des changements sur le comportement des agents.

FIG. 5.12 – L'environnement (en haut), la mémoire *gargamel* (à gauche) et la mémoire de *schtroumpf* (à droite).

```
eat apple
|-actor.own(apple)
  |-take apple
    |-dist(actor,apple)<1
      |-go apple
        |-door1.isOpen=true
          |-open door1
            |-door1.isLocked=false
            | |-unlock door1
            |   |-actor.own(key1)
            |   | |-take key1
            |   |   |-dist(actor,key1)<1
            |   |     |-go key1
            |   |       |-door2.isOpen=true
            |   |         |-open door2
            |   |           |-door2.isLocked=false
            |   |           | |-unlock door2
            |   |           |   |-dist(actor,door2)<1
            |   |           |     |-go door2
            |   |           |-dist(actor,door2)<1
            |   |             |-go door2
            |   |-dist(actor,door1)<1
            |     |-go door1
            |-dist(actor,door1)<1
              |-go door1
```

```
eat apple
|-actor.own(apple)
  |-take apple
    |-dist(actor,apple)<1
      |-go apple
        |-door1.isOpen=true
          |-open door1
            |-door1.isLocked=false
            | |-unlock door1
            |   |-actor.own(key1)
            |   | |-take key1
            |   |   |-dist(actor,key1)<1
            |   |     |-go key1
            |   |-dist(actor,door1)<1
            |     |-go door1
            |-dist(actor,door1)<1
              |-go door1
```

FIG. 5.13 – Plans 4 et 5 : En haut, l'agent *gargamel* a ramassé la clef *key2*. En bas, il a déverrouillé puis ouvert la porte *door2*.

```
eat apple
|-actor.own(apple)
  |-take apple
    |-dist(actor,apple)<1
      |-go apple
```

FIG. 5.14 – Plan 6 : L'agent *Gargamel* a ramassé la clef *key1*, déverrouillée puis ouvert la porte *door1* et va maintenant pouvoir s'emparer de la pomme *apple* afin de la manger.

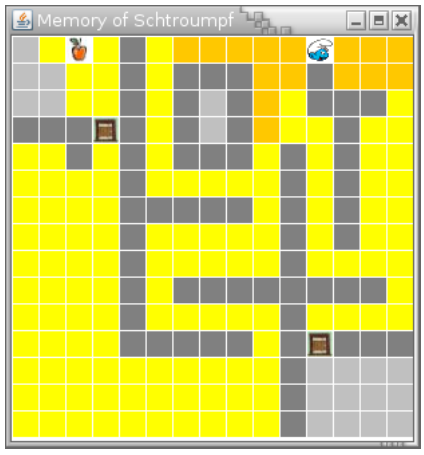


FIG. 5.15 – L'agent *schtroumpf* s'aperçoit de la disparition de la clef *key2*.

Conclusion Générale

« Demain sera le premier jour du reste de notre existence »
T. Pratchett & Neil Gaiman, De bons Présages

La simulation de comportements couvre un domaine de connaissance très large : de la représentation de connaissances à la planification, en passant la prise en compte des aspects sociaux, les challenges sont nombreux. Il existe des propositions de modèles basés sur les systèmes multi-agents mais ceux-ci souffrent de la difficulté de réutilisation des comportements de par la dissolution de l'expression des interactions dans les agents. De par l'utilisation de scripts créés ad hoc, les jeux vidéos, pourtant très avides de comportements intelligents, ne disposent pas de bibliothèques de comportements réutilisables. Dans beaucoup de simulations, la prise en compte des déplacements repoussée au moment de l'exécution du plan écarte délibérément certaines informations utiles et nuit à la vraisemblance du comportement.

Ma proposition de représentation de la connaissance basée sur le modèle centré interaction et la dualité *peut-subir/peut-effectuer* permet d'extraire et d'abstraire le code des interactions de manière à pouvoir réutiliser séparément des composants (agents, interactions, environnements) d'une simulation dans d'autres. Le moteur de comportement des agents cognitifs anticipant la gestion des déplacements au moment de la planification permet d'améliorer la rationalité et donc la crédibilité des comportements. Grâce à l'utilisation d'une mémoire et d'un module de replanification partielle, un agent peut évoluer dans un environnement dynamique : il est capable d'adapter localement le plan qu'il est en train d'exécuter face à des modifications de l'environnement, ce qui rend possible l'introduction d'autres agents. Ceci permet non seulement d'effectuer des simulations d'agents en concurrence, mais constitue de plus un préalable indispensable au fonctionnement des agents en équipe. Au sein de cette dernière, la répartition des connaissances permet d'utiliser le même moteur de comportement pour construire le plan d'équipe de manière semi-centralisée, ce qui préserve l'autonomie de décision des agents. L'affectation dynamique des tâches au sein de l'équipe permet de maximiser le nombre de tâches exécutées simultanément, en particulier lors de l'ajout ou du retrait d'un ou de plusieurs membres à l'équipe. La plate-forme développée a permis de valider la plupart de ces propositions.

Ce travail comporte toutefois quelques limites qu'il convient de souligner. Tout d'abord, ce travail ne gère pas certains aspects tels que le temps explicite ou l'environnement social. Le moteur de comportement des agents peut être confronté à des dead locks face à des situations identifiées. On pourra tenter de détecter ces situations afin de les éviter dans un premier

temps, puis de les gérer. L'intégration des déplacements dans la planification, si elle permet d'améliorer la rationalité des agents, est encore sujette à des imperfections. Ainsi, lors du calcul d'un plan par un agent, les déplacements sont calculés avec pour point d'origine la position actuelle de l'agent. Cette hypothèse est incorrecte à certaines étapes du plan car l'agent se sera déjà déplacé avant d'entreprendre ces actions. Ceci peut engendrer des déplacements erratiques dans certains cas particuliers. Pour considérer correctement le point d'origine des déplacements lors du calcul du plan, il faudrait connaître l'ordre d'exécution des actions alors même que celles-ci ne sont pas toutes connues puisque l'on est en train de les calculer. La plupart du temps, les effets sont heureusement masqués par la replanification qui reconsidère le point d'origine lorsque l'agent se déplace en exécutant le plan. A court terme, la plate-forme pourrait être enrichie en implantant la gestion des équipes. On pourra aussi implanter la méthodologie IODA afin de faciliter la création coordonnée des agents et des interactions.

A plus long terme, le modèle pourra être élargi sur plusieurs points. L'introduction d'un coût pour chaque action permettrait d'effectuer un choix dans les nœuds *OU* de l'arbre représentant le plan, en fonction du coût estimé de chaque sous-plan. Ce coût pourra évidemment dépendre de la personnalité de l'agent. On pourra ajouter l'obsolescence des informations de la mémoire en fonction de leur nature : l'exactitude de la position d'un agent animé est beaucoup plus éphémère que celle d'un agent inanimé ! On pourra pousser plus loin cette idée en ajoutant la possibilité pour un agent d'oublier les informations trop anciennes. Une gestion explicite du temps permettrait d'envisager la synchronisation afin de résoudre certains problèmes nécessitant l'intervention simultanée de plusieurs agents.

La gestion d'équipe semi-centralisée que je propose, si elle permet de donner aux agents une autonomie de décision, possède deux inconvénients. Le premier réside dans l'absence d'un mécanisme de remontée d'information des agents vers le chef. Il peut s'ensuivre le blocage de l'équipe toute entière en cas de panne de l'un des agents, qui ne peut pas atteindre le but qui lui a été délégué. Le second point concerne le chef : il constitue un point névralgique important car sa disparition est fatale pour l'équipe. Il constitue actuellement une cible de choix dans un contexte de compétition. Une gestion d'équipe totalement décentralisée serait souhaitable afin de pallier ce problème.

Bibliographie

[AIS06] AISEEK, 2006 : `http://www.aiseek.com/`.

[Ba85] Brownston et al.'s : Programming expert systems in OPS5. 1985.

[BCM96] S. Boussetta, D. Cohen et P. Moraitis : An agent model for distributed planning. *In Proc. French Conference on Multi-Agent Systems*, pages 22–35, 1996.

[BDM98] B. Beaufils, J.-P. Delahaye et P. Mathieu : Complete classes of strategies for the classical iterated prisoner's dilemma. *In Evolutionnary Programming VII proceedings LNCS 1447*, 1998.

[BF81] Avron Barr et Edward Feigenbaum : *The Handbook Of Artificial Intelligence*, volume I, chapitre II. Pitman, 1981.

[BF95] Avrim Blum et Merrick Furst : Fast planning through planning graph analysis. *In Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1636–1642, 1995.

[BGIZ02] Daniel S. Bernstein, Robert Givan, Neil Immerman et Shlomo Zilberstein : The complexity of decentralized control of Markov decision processes. *Math. Oper. Res.*, 27(4):819–840, 2002.

[BLM02] François Bousquet, Christophe Le Page et Jean-Pierre Müller : Modélisation et simulation multi-agent. *In Actes des deuxièmes assises nationales du GdR I3*, pages 173–182, 2002.

[Bro86] R. Brooks : A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.

[BTT06] Olivier Bonnet-Torrès et Catherine Tessier : De l'équipe aux individus : planification et replanification. *In Actes des JFSMA06*, pages 197–210, 2006.

[CGGG02] D. Capera, J-P. Georgé, M-P Gleizes et P. Glize : Un défi pour les SMA : explorer l'émergence. *In Systèmes muti-agents et systèmes complexes. Actes des JFIADSMA'02*, pages 159–162, 2002.

[Che06] Pierre Chevaillier : *Les systèmes multi-agents pour les environnements virtuels de formation.* Habilitation à Diriger les Recherches, Université de Bretagne Occidentale, Brest, 2006.

[DBBM07] Julien Derveeuw, Bruno Beaufils, Olivier Brandouy et Philippe Mathieu : L'apport des systèmes multi-agents à la modélisation des marchés financiers. *Revue Française d'Intelligence Artificielle*, 2007.

[DFJN97] J. E. Doran, S. Franklin, N. R. Jennings et T. J. Norman : On cooperation in multi-agent systems. *The Knowledge Engineering Review*, 12(3):309–314, 1997.

[DMR04] D. Devigne, P. Mathieu et J.-C. Routier : Planning for spatially situated agents in simulations. *In IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'04)*, pages 385–388. IEEE Press., 2004.

[DMR05a] D. Devigne, P. Mathieu et J.-C. Routier : Interaction-based approach for game agents. *In proceedings of ECMS'05*, pages 705–714, 2005.

[DMR05b] Damien Devigne, Philippe Mathieu et Jean-Christophe Routier : Simulation de comportements centrée interaction. *In Journées Francophones sur les Systèmes Multi-Agents (JFSMA'05) , Calais, France*, pages 47–50. Hermès-Lavoisier, 23-25 novembre 2005.

[DMR05c] Damien Devigne, Philippe Mathieu et Jean-Christophe Routier : Team of cognitive agents with leader : how to let them acquire autonomy. *In Proceedings of 2005 IEEE Symposium on Computational Intelligence and Games*, pages 256–262, 2005.

[DMR06] Damien Devigne, Philippe Mathieu et Jean-Christophe Routier : *Intelligence artificielle et jeux*, chapitre 10. Hermes-Lavoisier, november 2006.

[DMR07] Tony Dujardin, Philippe Mathieu et Jean-Christophe Routier : Une sélection d'actions prenant en compte opportunisme et personnalité. 2007. publication en cours.

[EH96] A. El Fallah-Seghrouchni et S. Haddad : A recursive model for distributed planning. *In Proc. of ICMAS'96*. AAAI Press, 1996.

[EK72] J. Edmonds et R.M. Karp : Theoretical improvement in algorithmic efficiency for network flows problems. *Journal of the Assoc. of Comput. Mach.*, 19(2):248–264, 1972.

[Fer95a] Jacques Ferber : *Les systèmes multi-agents*. InterEdition, Paris, 1995.

[Fer95b] Jacques Ferber : *Les systèmes multi-agents : Vers une intelligence collective*. Masson, 1995.

[FF56] L.R. Ford et D.R. Fulkerson : Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[FN71] R. Fikes et N. J. Nilsson : Strips : A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Gar70] Martin Gardner : Mathematical games. the fantastic combinations of John Conway's new solitaire game « life ». *Scientific American*, (223):120–123, Octobre 1970.

[GGG03] Jean-Pierre Georgé, Marie-Pierre Gleizes et Pierre Glize : Conception de systèmes adaptatifs à fonctionnalité émergente : la théorie AMAS. *Revue d'Intelligence Artificielle RSTI série RIA*, 17(4):591–626, 2003.

[Hay84] Brian Hayes : Computer recreations : The cellular automaton offers a model of the world and a world unto itself. *Scientific American*, March 1984.

[Hig02] Dan Higgins : *AI Game Programming Wisdom*, volume I, chapitre 3.2, pages 114–121. Charles River Media Inc., 2002.

[HL05] Bryan Horling et Victor Lesser : A survey of multi-agent organizational paradigms. *The Knowledge Engineering Review*, 19(4):281–316, 2005.

[HML04] Bryan Horling, Roger Mailler et Victor Lesser : A Case Study of Organizational Effects in a Distributed Sensor Network. *In Proceedings of the AAAI-04 Workshop on Agent Organizations : Theory and Practice*, pages 23–30, San Jose, California, July 2004. AAAI Press, California.

[Jen00] N. R. Jennings : Agent methodology for software engineering. *Communication of ACM*, 2000.

[KNH97] Jana Keohler, Bernhard Nebel et Jörg Hoffman : Extending plannning graphs to an ADL subset. Rapport technique, Institute of computer Science - Albert Ludwigs University - 79110 Freiburg, Germany, 1997. `http://www.informatik.uni-freiburg.de/~koehler/ipp.html`.

[KNHD97] Jana Keohler, Bernhard Nebel, Jörg Hoffman et Yannis Dimopoulos : Extending planning graphs to an ADL subset. *In Proceeding of ECP-97*, pages 273–285. Springer LNAI 1348, 1997.

[KS92] Henry A. Kautz et Bart Selman : Planning as satisfiability. *In Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, 1992.

[KS99] Henry Kautz et Bart Selman : Unifying SAT-based and graph-based planning. *In Proceedings of IJCAI-99*, 1999.

[KTS+97] H. Kitano, M. Tambe, P. Stone, M. Veloso, S. Coradeschi, E. Osawa, H. Matsubara, I. Noda et M. Asada : The RoboCup synthetic agent challenge,97. *In International Joint Conference on Artificial Intelligence (IJCAI97)*, 1997.

[Kyn06] Kynogon SA, 2006 : `http://www.kynogon.com`.

[Lai01] John E. Laird : It knows what you're going to do : adding anticipation to a quakebot. *In* Jörg P. Müller, Elisabeth Andre, Sandip Sen et Claude Frasson, éditeurs : *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 385–392, Montreal, Canada, 2001. ACM Press.

[LF03] D. Long et M. Fox : The 3rd international planning competition : Results and analysis. *Journal of Artificial Intelligence Research*, 20:1–59, 2003.

[LVL00] J. Laird et M. Van Lent : *Human Level AI's Killer Application : Interactive Computer Games*. AAAI Press, 2000.

[MAS06] MASA-SCI, 2006 : `http://www.masa-sci.com/directia.htm`.

[McD98] D. McDermott : PDDL — the planning domain definition language, 1998.

[Min86] Marvin Minsky : *The society of mind*. 1986.

[MP05] Philippe Mathieu et Sébastien Picault : Towards an interaction-based design of behaviors. *In European workshop on Multi-Agent systems (EUMAS'05)*, décembre 2005.

[MP06] Philippe Mathieu et Sébastien Picault : Vers une représentation des comportements centrée interactions. *In Actes du 15e congrès francophone AFRIF-AFIA Reconnaissance des Formes et Intelligence Artificielle (RFIA'2006)*, 2006.

[MP07] Philippe Mathieu et Sébastien Picault : Ioda : l'approche centrée interactions pour les agents situés, 2007. A paraître.

[NAI+03] D. Nau, T.-C. Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu et F. Yaman : SHOP2 : An HTN planning system. *JAIR*, 20:379–404, 2003.

[Nar04] A. Nareyek : Artificial intelligence in computer games - state of the art and future directions. *ACM Queue*, 1(10):58–65, 2004.

[NCLMA99] Dana S. Nau, Yue Cao, Amnon Lotem et Hector Munoz-Avila : SHOP : Simple hierarchical ordered planner. *In IJCAI'99 : Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 968–975, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[NK93] B. Nebel et J. Koehler : Plan modification versus plan generation : A complexity-theoretic perspective. *In Proceedings of of the 13th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1436–1441, 1993.

[NTCS99] S. Ni, Y. Tseng, Y. Chen et Y. Sheu : The broadcast storm problem in a mobile ad hoc network. *In Proceedings of the ACM International Conference on Mobile Computing and Networking (MobiCom)*, Seattle, USA, August 1999.

[Ped89] Edwin Pednault : ADL : Exploring the middle ground between strips and the situation calculus. *In* R. Brachman, H. J. Levesque et R. Reiter, éditeurs : *Proceedings of the first International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332. Kaufmann, Morgan, 1989.

[PF05] Damien Pellier et Humbert Fiorino : Coordinated exploration of unknown labyrinthine environments applied to the pusruite-evasion problem. *In International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS)*, pages 895–902, 2005.

[PW92] J. S. Penberthy et D. Weld : UCPOP : A sound, complete, partial-order planner for ADL. *In Third International Conference on Knowledge Representation and Reasoning (KR-92)*, October 1992.

[Que02] Ronan Querrec : *Les Systèmes Multi-Agents pour les Environnements Virtuels de Formation. Application à la sécurité civile.* Thèse de doctorat, Université de Bretagne Occidentale, Brest, 2002.

[RD00] Alexandre Moretto Ribeiro et Yves Demazeau : *Un Modèle d'Interaction Dynamique pour les Systèmes Multi-Agents (A Dynamic Interaction Model for the Multi-Agent Systems.* Thèse de doctorat, Université de Grenoble 1, Saint-Martin-d'Hères, FRANCE, 2000.

[Rey87] Craig W. Reynolds : Flocks, herds, and schools : A distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987.

[RN02] Stuart J. Russell et Peter Norvig : *Artificial Intelligence : A Modern Approach (International Edition).* Pearson US Imports & PHIPEs, November 2002.

[Rob] Robocup : `http://www.robocup.org/`.

[SCK00] Fabiano Silva, Marcos A. Castilho et Luis Allan Künzle : Petriplan : A new algorithm for plan generation (preliminary report). *In IBERAMIA-SBIA*, pages 86–95, 2000.

[Spi06] SpirOps, 2006 : `http://www.spirops.com/`.

[SPW+04] Evren Sirin, Bijan Parsia, Dan Wu, James Hendler et Dana Nau : HTN planning for web service composition using SHOP2. *Web Semantics : Science, Services and Agents on the World Wide Web*, 1(4):377–396, October 2004.

[Ste94] Tony Stentz : Original D*. *In proceedings of ICRA'94*, 1994.

[Sto06] Stottler Henke Associates Inc., 2006 : `http://www.simbionic.com`.

[Sur02] Smith Surasmith : *AI Game Programming Wisdom*, volume I, chapitre 4.2. Charles River Media Inc., 2002.

[TBC06] Vincent Thomas, Christine Bourjot et Vincent Chevrier : Heuristique pour l'apprentissage automatique décentralisé d'interactions dans des systèmes multi-agents réactifs. *In actes des RFIA 2006*, 2006.

[Toz02] Paul Tozour : *AI Game Programming Wisdom*, volume I, chapitre 10.6, pages 541–547. Carles River Media Inc., 2002.

[Toz04] Paul Tozour : *AI Game Programming Wisdom*, volume II, chapitre 5.2, pages 303–306. Carles River Media Inc., 2004.

[TSP] TSP : `http://www.tsp.gatech.edu/`.

[Tur50] Alan Turing : Computing machinery and intelligence. *Mind*, LIX(236):433–460, 1950.

[WAS98] Daniel S. Weld, Corin R. Anderson et David E. Smith : Extending Graphplan to handle uncertainty & sensing actions. *In Proceedings of AAAI '98*, 1998.

[Wel99] Daniel S. Weld : Recent advances in AI planning. *AI Magazine*, 20(2):93–123, 1999.

[Whi07] Dustin White : Clarifications and extensions to tactical waypoint graph algorithms for video games. *In ACM-SE 45 : Proceedings of the 45th annual southeast regional conference*, pages 316–320, New York, NY, USA, 2007. ACM Press.

Publications

1 Articles dans les actes d'une conférence avec comité de relecture

1.1 Planning for spatially situated agents

Abstract - *Most often, agent-based situated simulations use reactive agents. While the use of this paradigm offers a rather natural way to build simple behaviours, it is not so easy to define complex behaviours, and these are often built ad hoc and requires precise adjustements. Moreover reactive model is not well fitted to describe deliberate group behaviours since it is more intended to emergence. For these reasons and in order to obtain more explicative models, we are working on proactive agent-based situated simulations. We aim at proposing a generic cognitive agent model. Such agents must build plan to achieve some goals and thus to propose a behaviour. In this paper we show why and how classical planning works must be adapted to the particular context of spatially situated agents in simulations.*

IAT'04 Beijing (China) september 20 - 24 2004

```
@InProceedings{DMR_IAT04,
author = {Devigne, D. and Mathieu, P. and Routier, J.-C.},
title = {Planning for Spatially Situated Agents},
booktitle = {IEEE/WIC/ACM International Conference on Intelligent Agent
    Technology (IAT'04)},
pages = {385-388},
year = {2004},
publisher = {IEEE Press.}}
```

1.2 Teams of cognitive agents

Abstract - *Most often situated multi-agent simulations, of which platform-games are an example, uses reactive agents. This approach has limitations as soon as complex behaviours are desired. For these reasons we propose an approach using cognitive agents. They have knowledge, objectives and are able to build plans in order to achieve their goals and then execute them. In this paper we particulary address the problem of teams of cognitive agents. We chose to build teams directed by a leader. One major problem is the building of the team plan and in particular one difficulty is to find a mean in order to let autonomy to the team members. This*

can be done if the leader builds abstract plans. We present in this article a solution to this problem.

CIG'05 April 4-6 2005 Essex University, Colchester, Essex

```
@inproceedings{DMR_CIG05,
author = {Damien Devigne and Philippe Mathieu and Jean-Christophe Routier},
title = {Team of cognitive agents with leader : how to let them acquire
    autonomy},
booktitle = {Proceedings of 2005 IEEE Symposium on Computational Intelligence
    and Games},
pages = {256--262},
year = {2005}}
```

1.3 An interaction-based approach for game agents

Abstract - *Most often, agents in simulations are based on reactive models. Such systems do not plan actions for the agents and are limited to express complex realistic behaviour. We propose here an agent model for spatially situated simulations, like computer games are. The involved agents are cognitive ones : they are able to build plans and to adapt them according to the dynamics of the simulation. Our main goal is to obtain believable behaviours for the agents in simulations. Thus we propose a generic model for cognitive situated agent in simulations of which video games are a typical example. The proposed ideas promote reusability from one simulations to another and software engineering concepts concerns have driven our work. The two main problems can easily (and without surprise) be identified : how to represent knowledge ? and how to build plan using this knowledge. To solve the first we propose to describe the laws that manage the simulated world in term of interactions that can be performed by some agents and suffered by others. Concerning the second problem, we propose a planning algorithm that is based on the interactions and take into account the facts that agent are situated and that plans must be executed in a situated environment that is in permanent evolution. Thus the plans are actually incrementally built through partial replanning.*

ESM'05 June 1 - 4, 2005 Riga, Latvia

```
@inproceedings{DMR_ECMS05,
author = "Damien Devigne and Philippe Mathieu and Jean-Christophe Routier",
title = "Interaction-Based Approach For Games Agents",
booktitle = "Proceedings of the 19th Europeen Conference on Modelling and
    Simulation, Simulation in Wider Europe - ECMS 2005",
pages = {705-714}
year = "2005"}
```

1.4 Gestion d'équipes et autonomie des agents

Résumé - *Dans cet article, nous présentons une solution permettant de gérer des équipes d'agents hétérogènes situés dirigés par un chef. Celui-ci est chargé d'établir le plan d'équipe et de distribuer les tâches à effectuer aux agents en fonction de leurs spécificités. Dans une*

approche traditionnelle, le chef planifie tout. Or, il est important de laisser de l'autonomie de décision aux agents afin de leur permettre de s'adapter à la configuration de leur environnement qui n'est pas forcément connue par le chef lors de la planification. Notre solution consiste à créer des plans d'équipes abstraits où tout n'est pas résolu par le chef, ce qui laisse aux agents une réelle autonomie de décision. Nous proposons ensuite une technique d'allocation des tâches calculées aux agents. Notre approche permet à une équipe d'agents hétérogènes de réaliser des tâches qui requièrent les capacités de plusieurs agents. Ces agents agissent de manière autonome en fonction des buts donnés par le chef et de la configuration de l'environnement.

JFSMA05 à Calais du 23 au 25 novembre 2005

```
@InProceedings{DMR_JFSMA05,
author = {Devigne, Damien and Mathieu, Philippe and Routier, Jean-Christophe},
title = "{Simulation de comportements centrée interaction}",
year = {2005},
pages = {47--50},
publisher = {Hermès-Lavoisier },
booktitle = {Journées Francophones sur les Systèmes Multi-Agents (JFSMA'05),
    Calais, France},
month = {23-25 novembre}}
```

2 Chapitre de livre avec comité de relecture

2.1 Simulation de comportements centrée interaction

Résumé - *Nous présentons ici une solution permettant de gérer des équipes d'agents situés dirigées par un chef chargé d'établir le plan de son équipe. Le chef calcule le plan d'équipe en fonction de ses connaissances, et distribue les actions à effectuer aux autres agents de l'équipe. Pour élaborer ce plan, il est possible de mettre en oeuvre plusieurs stratégies permettant de laisser plus ou moins d'autonomie de comportement aux agents. Cette autonomie permet aux agents de s'adapter aux contraintes imposées par l'environnement. Nous montrons dans ce chapitre comment notre approche permet à une équipe hétérogène d'agents de réaliser des tâches complexes qui nécessitent l'utilisation des capacités de plusieurs agents.*

```
@InBook{DMR_IAetJeux06,
author = {Devigne, Damien and Mathieu, Philippe and Routier, Jean-Christophe},
editor = {Cazenave, Tristan},
title = {Intelligence artificielle et jeux},
chapter = {10},
publisher = {Hermes-Lavoisier},
year = {2006},
month = {november},
}
```