Titre: Title:	Étude comparative de planificateurs appliqués au domaine des jeux vidéo
Auteur: Author:	Jean-François Cartier
Date:	2011
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Cartier, J.-F. (2011). Étude comparative de planificateurs appliqués au domaine des jeux vidéo [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/570/

UNIVERSITÉ DE MONTRÉAL

ÉTUDE COMPARATIVE DE PLANIFICATEURS APPLIQUÉS AU DOMAINE DES JEUX VIDÉO

JEAN-FRANÇOIS CARTIER
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AVRIL 2011

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ÉTUDE COMPARATIVE DE PLANIFICATEURS APPLIQUÉS AU DOMAINE DES JEUX VIDÉO

présenté par : CARTIER, Jean-François
en vue de l'obtention du diplôme de : Maîtrise ès Sciences Appliquées
a été dûment accepté par le jury d'examen constitué de :

M. BOYER, François-Raymond, Ph.D., président.
M. GAGNON, Michel, Ph.D., membre et directeur de recherche.
M. CÔTÉ, Carle, Ph.D., membre et codirecteur de recherche.
M. PAL, Christopher, Ph.D., membre.

REMERCIEMENTS

Je tiens à témoigner ma reconnaissance envers mon directeur de recherche, le professeur Michel Gagnon, pour son soutien, pour ses conseils et pour m'avoir initié à la recherche. Je remercie mon codirecteur, Carle Côté, pour son aide, le temps qu'il m'a accordé et son apport au projet grâce à son expertise dans le domaine des jeux vidéo. Je remercie également le CRSNG, le FQRNT, Ubisoft et la Banque Nationale de m'avoir octroyé les bourses qui m'ont permis de réaliser ce projet. Finalement, je remercie Gabriel Cartier, Matthieu Ouellette-Vachon, Hubert Guérard et Fellipe Monteiro pour leur aide et leurs conseils sans lesquels ce projet n'aurait pas été ce qu'il est aujourd'hui.

RÉSUMÉ

L'utilisation de la planification dans une architecture de prise de décision d'un jeu vidéo est une technique récente et il existe peu de références démontrant son efficacité par rapport aux techniques usuelles. Notre étude consiste en une évaluation expérimentale de deux planificateurs, GOAP et HTN, dans l'élaboration d'agents autonomes dans un jeu de tir. L'objectif du projet de recherche est de déterminer s'il est avantageux ou non d'utiliser ces planificateurs selon les critères suivants : la qualité de l'agent, la qualité de la planification, la jouabilité et certains attributs de qualité non fonctionnels tels que les limitations comportementales, la facilité d'implémentation et la robustesse face aux changements de conception.

Nous avons comparé les deux types de planificateurs à une technique usuelle, la machine à états finis. Les résultats obtenus montrent que les architectures utilisant la planification offrent une qualité d'agent supérieure à la machine à états finis, nécessitent en moyenne moins de temps de calcul, ne nécessitent pas de prévoir toutes les situations auxquelles l'agent fera face et sont plus robustes aux changements de conception. Toutefois, elles résultent en un agent moins réactif et l'implémentation de l'architecture GOAP est une tâche plus complexe que l'implémentation de la machine à états. Finalement, GOAP offre une qualité de l'agent légèrement supérieure à HTN, mais ce dernier est plus facile d'implémentation.

En définitive, sans toutefois être sans inconvénient, les planificateurs possèdent des avantages à être utilisés dans une architecture de prise de décision d'un jeu de tir. Quant au planificateur le plus approprié, le choix devrait être réalisé en fonction des exigences spécifiques du projet.

ABSTRACT

The use of planning in a decision making architecture of a video game is a recent technique and there are few references demonstrating its effectiveness compared to conventional techniques. Our study provides an experimental evaluation of two planners, GOAP and HTN, in the development of autonomous agents in a shooting game. The objective of the project is to determine whether it is advantageous or not to use these planners based on the following criteria: quality of the agent, quality of planning, gameplay and some non-functional quality attributes such as behavioral limitations, ease of implementation and robustness to design changes.

We compared the two types of planners to a conventional technique, the finite state machine. The results show that the planning architectures offer a superior quality of agent than the finite state machine, require less computing time, do not require to anticipate all situations that the agent will face and are more robust to changes in design. However, they result in a less reactive agent and the implementation of the GOAP architecture is a more complex task than implementing the state machine. Finally, GOAP provides a slightly superior agent quality than HTN, but the latter is easier to implement.

Ultimately, though not without downsides, planners have advantages for use in a decision making architecture of a shooter. As for the most appropriate planner, the choice should be made according to specific project requirements.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ANNEXES

LISTE DES SIGLES ET ABRÉVIATIONS

ADL	Action Description Language
CBR	Case-Based Reasoning
CSP	Constraint Satisfaction Problem
FPS	First-Person Shooter
FSM	Finite State Machine
GOAP	Goal-Oriented Action Planning
GOB	Goal-Oriented Behavior
HSP	Heuristic Search Planning
HTN	Hierarchical Task Network
IA	Intelligence Artificielle
NOLF2	No One Lives Forever 2
PDDL	Planning Domain Definition Language
PNJ	Personnage Non-Joueur
POP	Partial-Order Planning
RTS	Real-Time Strategy
STRIPS	STanford Research Institute Problem Solver
TBS	Turn-Based Strategy
TPS	Third-Person Shooter
UT	Unreal Tournament

Caractères usuels

T	Niveau de difficulté approprié de l'agent par rapport à t_k
T_2	Niveau de difficulté approprié de l'agent par rapport à pt
S	Diversité du comportement de l'agent par rapport à t_k
S_2	Diversité du comportement de l'agent par rapport à pt
H_n	Entropie normalisée entre [0,1] des cellues visitées par l'agent dans une partie
$E\{H_n\}$	Espérance de H_n, soit la diversité spatiale de l'agent
I	Jouabilité
N	Nombre total de parties jouées par l'agent
t_k	Temps moyen mis par l'agent pour tuer son adversaire sur N parties
t_{min}	Temps minimal mis par l'agent pour tuer son adversaire en N parties
t_{max}	Temps maximal mis par l'agent pour tuer son adversaire en N parties

pt	Pointage moyen de l'agent sur N parties
pt_{min}	Pointage minimal de l'agent en N parties
pt_{max}	Pointage maximal de l'agent en N parties
V_n	Nombre de visites pour toutes les cellules visitées pour une partie

Caractères grecs

σ	Écart type de t_k sur N parties
σ_{max}	Estimé de la valeur maximale de σ
ϕ	Écart type de pt sur N parties
ϕ_{max}	Estimé de la valeur maximale de ϕ

LISTE DES TRADUCTIONS

analyse de l'environnement	terrain analysis
analyse moyens-fin	means-end analysis
analyseur syntaxique	parser
apprentissage par modification des faiblesses	weakness modification learning
apprentissage par renforcement	reinforcement learning
arbre de comportement	behavior tree
arbre de décision	decision tree
architecture de subsomption	subsumption architecture
code machine	bytecode
concepteur de jeu	game designer
degré d'appartenance	degrees of membership
degré de compatibilité	fitness value
écriture du scripts	scripting
engin de jeu vidéo	game engine
figé dans le code	hard-coded
génération d'histoires	story generation
graphe de planification	planning graph
interface de connexion	socket
jeu de tir	shooter
logiciel intégré	framework
logique floue	fuzzy logic
machine à états basée sur une pile	stack-based finite state machine
machine à états finis	finite state machine
machine à états flous	fuzzy state machine
machine à états hiérarchique	hierarchical state machine
narration interactive	interactive drama or interactive storytelling
personnage non-joueur	non-player character (NPC)
planification de trajectoires	pathplanning
planification en ordre partiel	partial-order planning
planification par recherche avec heuristique	heuristic search planning
planification sociale	social planning
prédiction statistique n-gramme	n-gram statistical prediction
problème de satisfaction de contraintes	constraint satisfaction problem

profilage du joueur	player modeling
recherche de chemins	pathfinding
raisonnement basé sur des cas	case-based reasoning
réseau bayesiens	Bayesian network
réseau de neurones	neural network
réseau hiérarchisé de tâches	hierarchical task network
système basé sur les règles	rule-based system
système de production	production system
trame	frame

CHAPITRE 1

INTRODUCTION

Jadis une niche de marché considérée seulement par les plus curieux dans les années soixante-dix [119], l'industrie du jeu vidéo est dorénavant un secteur économique majeur de notre société. La croissance rapide des composants électroniques qui s'est amorcée il y a quelques années est en grande partie responsable de l'accroissement de cette industrie. En effet, le matériel informatique ne cesse de s'améliorer et son coût diminue, ce qui permet aux fabricants de consoles de jeux vidéo de concevoir des machines très performantes à un prix abordable. Ceci offre aux développeurs la possibilité d'augmenter la complexité et le réalisme de leurs jeux tout en élargissant grandement le nombre de joueurs.

La demande grandissante pour des jeux toujours plus immersifs force les compagnies à créer des personnages les plus réalistes possible. Cette exigence entraîne un accroissement de la recherche et du développement dans le domaine des personnages de jeux vidéo crédibles, car ils constituent un élément clé de l'amélioration continue du réalisme des jeux [102]. Afin que les personnages d'un jeu vidéo soient crédibles, ils doivent posséder deux caractéristiques importantes. Premièrement, il est nécessaire qu'ils possèdent une apparence réaliste et, deuxièmement, ils doivent agir de façon vraisemblable.

L'évolution de l'apparence des personnages s'est effectuée au même rythme que celle des processeurs graphiques, c'est-à-dire à un rythme effréné. Par contre, le développement du comportement des personnages, guidé par l'intelligence artificielle du jeu vidéo, n'a pas été aussi rapide que celui de leur apparence [107]. Afin de réduire l'écart entre l'apparence et le comportement des personnages, la communauté scientifique, autant académique qu'industrielle, s'est récemment intéressée au problème de l'intelligence artificielle dans les jeux vidéo.

Le comportement des personnages non-joueur (PNJ) d'un jeu vidéo est généré par un module de son intelligence artificielle nommé architecture de prise de décision. Plusieurs techniques, possédant chacune leurs avantages et leurs inconvénients, sont utilisées pour implémenter une architecture de prise de décision dans un jeu vidéo : l'écriture de scripts, les machines à états finis, les arbres de décisions, etc. Récemment, les recherches en intelligence artificielle appliquée aux domaines des jeux vidéo se sont concentrées au développement de nouvelles techniques pour créer des personnages plus crédibles. Parmi celles-ci figure l'aug-

mentation de l'autonomie des agents grâce à des méthodes telles que la planification en temps réel [102].

La planification a été utilisée avec succès dans l'architecture de prise de décision dans quelques jeux connus [90, 91, 92]. Malgré tout, il existe peu de références portant sur les avantages et les inconvénients à utiliser cette technique d'intelligence artificielle dans le domaine spécifique du jeu vidéo. De plus, un problème de planification est résolu par différents systèmes appelés planificateurs. Il existe aujourd'hui une multitude de planificateurs et il est difficile de déterminer celui qui offrira les meilleurs résultats à un problème spécifique. Dans cette optique, le présent projet vise à faire une comparaison de la performance de deux différentes architectures de prise de décision utilisant un planificateur, soit une architecture GOAP et une architecture HTN, dans un jeu vidéo de tir. En comparant les performances de ces deux architectures avec une architecture de prise de décision utilisant une machine à états finis, nous sommes en mesure de vérifier s'il est avantageux d'utiliser une architecture de prise de décision utilisant la planification. Le cas échéant, il nous est également possible de déterminer le planificateur qui est le plus approprié.

Le second chapitre présente d'abord la problématique, l'hypothèse guidant la recherche et les objectifs. Le troisième chapitre consiste en une mise en contexte en présentant les techniques d'intelligence artificielle couramment utilisées dans un système de prise de décision dans un jeu vidéo sont présentées. Au quatrième chapitre, une explication approfondie de la planification dans le domaine de l'intelligence artificielle classique est fournie. Par la suite, au cinquième chapitre, une revue des travaux antérieurs portant sur la planification utilisée dans le domaine des jeux vidéo est présentée. Le sixième chapitre explique notre méthodologie et le septième chapitre expose les résultats obtenus. Finalement, le huitième et dernier chapitre présente la conclusion, soit une synthèse des travaux effectués suivi de la description des travaux futurs.

CHAPITRE 2

Problématique

Les problèmes d'intelligence artificielle appliquée au domaine des jeux vidéo sont intéressants puisqu'ils diffèrent souvent des problèmes d'IA classique. En effet, leur espace de recherche est très vaste et l'algorithme doit retourner une solution, qu'elle soit complète ou non, dans un temps limité [88]. Dans bien des cas, il ne suffit pas de trouver une technique qui déterminera la solution optimale, mais il faut plutôt déterminer la technique qui retournera la meilleure solution dans le temps alloué. De plus, il faut toujours garder en tête que la meilleure solution dans le domaine du jeu vidéo n'est pas nécessairement la solution optimale au problème. En effet, les personnages d'un jeu vidéo doivent commettre des erreurs afin que le joueur puisse gagner et le nombre d'erreurs commises doit varier en fonction de la difficulté du jeu. Tout ceci doit être couplé avec un comportement qui paraît malgré tout intelligent pour un être humain, ce qui représente un défi majeur. Afin de relever ce défi, il faut déterminer quelle technique d'IA donnera les meilleurs résultats dans le contexte spécifique d'un jeu et également adapter les nouvelles techniques de l'IA pour les appliquer aux jeux vidéo. Le récent intérêt apporté par la communauté scientifique autant académique qu'industrielle à l'intelligence artificielle dans les jeux vidéo a donc permis l'exploitation de plusieurs techniques prometteuses de l'IA dans les jeux vidéo. Parmi ces techniques, notons les réseaux Bayesiens, la logique floue, les arbres de décision, les réseaux de neurones et la planification [98].

Nous nous intéresserons spécifiquement à la planification, soit « [...] la tâche qui consiste à mettre au point une séquence d'actions permettant d'atteindre un but » [99]. Ce type de problème est résolu par différents systèmes appelés *planificateurs*. La planification appliquée au domaine du jeu vidéo est une technique intéressante puisqu'elle s'adresse à plusieurs sous-domaines de l'intelligence artificielle dans les jeux vidéo, notamment la recherche de chemins [79], la création automatique de niveaux de jeu [95], la détermination de séquences dans les jeux de stratégie [32] et les architectures de prise de décision [89]. L'utilisation de planificateurs lors de l'élaboration d'une architecture de prise de décision comporte plusieurs avantages [89] par rapport aux méthodes courantes telles que les machines à états finis et les langages de script. Par contre, bien que certains jeux implémentant cette technique aient connu un grand succès [92], il existe peu de méthodologies ou de références qui permettent d'établir quel type de planificateur devrait être utilisé lors de l'élaboration d'une architecture

de prise de décision dans un contexte spécifique de jeu vidéo. En effet, il est important de mentionner que, puisque la recherche sur la planification a débuté dans les années soixante-dix [99], il existe aujourd'hui une multitude de planificateurs. Chaque planificateur est caractérisé par plusieurs facteurs qui diminuent considérablement l'aptitude à déterminer à priori ses performances dans un contexte spécifique : l'approche utilisée pour résoudre le problème, la représentation du problème, les types de problèmes visés, la nécessité de retourner une solution optimale ou non, etc. Il est clair que la décision d'utiliser un certain planificateur en fonction du domaine visé n'est pas simple et qu'il est difficile d'estimer ses performances à priori. Dans cette optique, la compétition internationale de planificateurs [73] a été créée en 1998. Cet événement bisannuel compare les résultats de différents planificateurs soumis à plusieurs problèmes de domaines variés dans le but d'amasser des statistiques rendues publiques afin de pouvoir évaluer la performance générale des planificateurs. Malgré tout, en analysant les résultats des différentes compétitions, on remarque que les performances des planificateurs varient généralement grandement en fonction du domaine de planification. À la lumière des résultats, il semble clair que le choix d'un certain planificateur dans un contexte spécifique exige une analyse approfondie quant à la représentation du problème et des tests expérimentaux sur un ensemble de planificateurs candidats afin de déterminer celui qui est le plus performant. Dans cette optique, notre étude consiste en une évaluation expérimentale de deux différents planificateurs, dans le but de déterminer celui qui est le plus approprié pour l'élaboration d'une architecture de prise de décision dans un type particulier de jeu vidéo, soit un jeu de tir.

2.1 Objectif

L'objectif général du présent projet est de vérifier s'il est avantageux d'utiliser une architecture de prise de décision utilisant la planification par rapport à l'utilisation d'une technique usuelle, la machine à états finis, et, le cas échéant, de déterminer le planificateur qui est le plus approprié.

2.2 Question de recherche

La question de recherche à la base de nos travaux est la suivante : quels avantages présente l'utilisation de certains planificateurs pour l'élaboration d'une architecture de prise de décision par rapport aux techniques couramment utilisées ? Pour répondre à cette question, nous tenterons de démontrer les avantages de deux types distincts de planificateurs par rapport à une machine à états finis. Les deux types de planificateurs que nous avons choisis sont parmi les plus couramment utilisées comme architecture de prise de décision dans le domaine des

jeux vidéos [43, 50, 81, 82, 90, 91, 92] :

1. Architecture GOAP, c'est-à-dire une exploration A* dans un espace d'états en utilisant une représentation basée sur STRIPS.
2. Architecture HTN, c'est-à-dire un planificateur qui effectue une décomposition hiérarchique d'un problème de planification en utilisant un réseau de tâches.

2.3 Hypothèse

L'hypothèse principale de notre recherche est la suivante : le planificateur HTN est le plus approprié pour une architecture de prise de décision d'un jeu vidéo d'action. En effet, ce type de planificateur semble le plus équilibré par rapport aux critères d'un jeu vidéo d'action. En comparaison avec GOAP, les planificateurs HTN peuvent effectuer une replanification partielle, c'est-à-dire qu'ils permettent de ne replanifier que la partie du plan qui le rend invalide. Ceci laisse croire que le temps de planification global du planificateur HTN sera moins élevé que GOAP et, étant donné que la replanification est plus rapide, l'agent pourra replanifier plus souvent, lui inculquant ainsi un caractère plus réactif. Par contre, puisque GOAP doit replanifier le plan en entier lorsque celui-ci est invalidé, l'agent pourrait avoir un comportement plus varié. Puisque le planificateur HTN présente des qualités qui semblent lui conférer, à priori, une bonne performance globale sans nécessairement dominer sur un aspect spécifique, nous croyons qu'il sera le plus adapté aux requis d'une architecture de prise de décision d'un jeu vidéo d'action.

CHAPITRE 3

Intelligence artificielle dans les jeux vidéo

Le présent chapitre a comme but premier de présenter les différentes techniques d'intelligence artificielle actuellement utilisées dans les jeux vidéo. Tout d'abord, nous donnons une brève explication de deux types de jeux vidéo populaires, les jeux de tir et les jeux de stratégie, et nous présentons les techniques d'IA employées dans les architectures de prise de décision des jeux vidéo. Bien que l'intelligence artificielle soit utilisée pour résoudre plusieurs autres problèmes dans les jeux vidéo, tels que le mouvement et la navigation des personnages, la recherche de chemins et l'analyse de l'environnement, notre projet ne porte que sur la prise de décision. Suivra une explication détaillée de la planification.

3.1 Types de jeux vidéo populaires

Dans ce mémoire, nous ferons références à deux types précis de jeux vidéo qui se distinguent par leur popularité et l'intérêt qu'ils représentent pour l'application de techniques de l'intelligence artificielle. Il s'agit des jeux de tir et des jeux de stratégie.

3.1.1 Jeux de tir

Les jeux de tir sont un sous-genre des jeux d'action. Le point central de ce type de jeu est le combat invoquant généralement des armes de tir. Un jeu de tir est généralement constitué de plusieurs missions qui doivent être achevées en éliminant plusieurs types d'ennemis. Les jeux de tir comprennent souvent des éléments à amasser tels que des munitions, de nouvelles armes, des armures, des points de vie, etc. Lorsque la perspective est une perspective de première personne, c'est-à-dire lorsque le joueur voit l'action au travers des yeux du protagoniste, on parlera alors de jeu de tir à la première personne ou « First-Person Shooter » (FPS) [114]. Dans le cas où le joueur peut voir complètement le personnage qu'il contrôle, on parle alors de jeu de tir à la troisième personne ou « Third-Person Shooter » (TPS) [117]. Le populaire Quake III Arena (1999) [55] est un exemple de jeu de tir à la première personne.

3.1.2 Jeux de stratégie

Une bonne définition des jeux de stratégie provient de [116] : « Les jeux de stratégie sont un genre de jeux vidéo qui mettent l'accent sur une réflexion et une planification habile

afin de parvenir à la victoire. Plus précisément, un joueur doit planifier une série d'actions contre un ou plusieurs adversaires et la réduction des forces de l'ennemi est habituellement un objectif. La victoire est obtenue par le biais d'une planification supérieure et l'élément de hasard possède un rôle moins important. Dans la plupart des jeux vidéo de stratégie, le joueur possède une vue globale du jeu et contrôle ses unités grâce à une série d'ordres. Ainsi, la plupart des jeux de stratégie comportent des éléments de guerre à divers degrés et disposent d'une combinaison de facteurs tactiques et stratégiques. En plus du combat, ces jeux interpellent souvent la capacité du joueur à explorer et à gérer une économie. »

Dans les jeux de stratégie, chaque joueur doit typiquement amasser des ressources, construire une armée et finalement attaquer les ennemis pour les détruire. Une partie peut être décomposée en deux phases : la phase de développement et la phase de combat. Chaque joueur commence une partie avec quelques unités et quelques bâtiments. Il doit alors créer une armée avant d'attaquer ses ennemis. Il s'agit de la phase de développement ou phase de production. Les actions possibles lors de cette phase sont typiquement :

- Envoyer une unité spécialisée collecter des ressources.
- Construire un bâtiment.
- Produire une unité.
- Améliorer les unités.

Cette phase est cruciale, car toute la partie dépend de la méthode utilisée et la rapidité pour la compléter. Nous verrons plus loin que plusieurs travaux antérieurs se concentrent seulement sur cette phase. Lorsqu'il sent que son armée est assez forte, le joueur peut mobiliser ses troupes pour attaquer ses ennemis. Il s'agit de la phase de combat.

Il existe deux sous-genres importants aux jeux de stratégie. Lorsque les actions du joueur sont effectuées en temps réel, c'est-à-dire que le jeu se poursuit pendant qu'il prend ses décisions, on parle alors de jeu de stratégie en temps réel ou « Real-time Strategy » (RTS) [115]. Lorsque le joueur possède un certain temps de réflexion pour prendre ses décisions pendant lequel le jeu est arrêté, on parle alors de jeu de stratégie basé sur les tours ou « Turn-based Strategy » (TBS) [118]. StarCraft (1998) [13] est un exemple typique de jeu de stratégie.

3.2 Techniques d'intelligence artificielle utilisées pour effectuer la prise de décision

L'architecture de prise de décision est le module de l'intelligence artificielle d'un PNJ qui génère son aspect comportemental. Il existe plusieurs techniques pour effectuer la prise de décision dans un jeu vidéo et il est possible d'implémenter une architecture de prise de décision de différentes façons. Néanmoins, elles fonctionnent toutes globalement de la même manière. La figure 3.1, tirée de [79], présente un exemple de schéma d'une architecture de prise de décision.

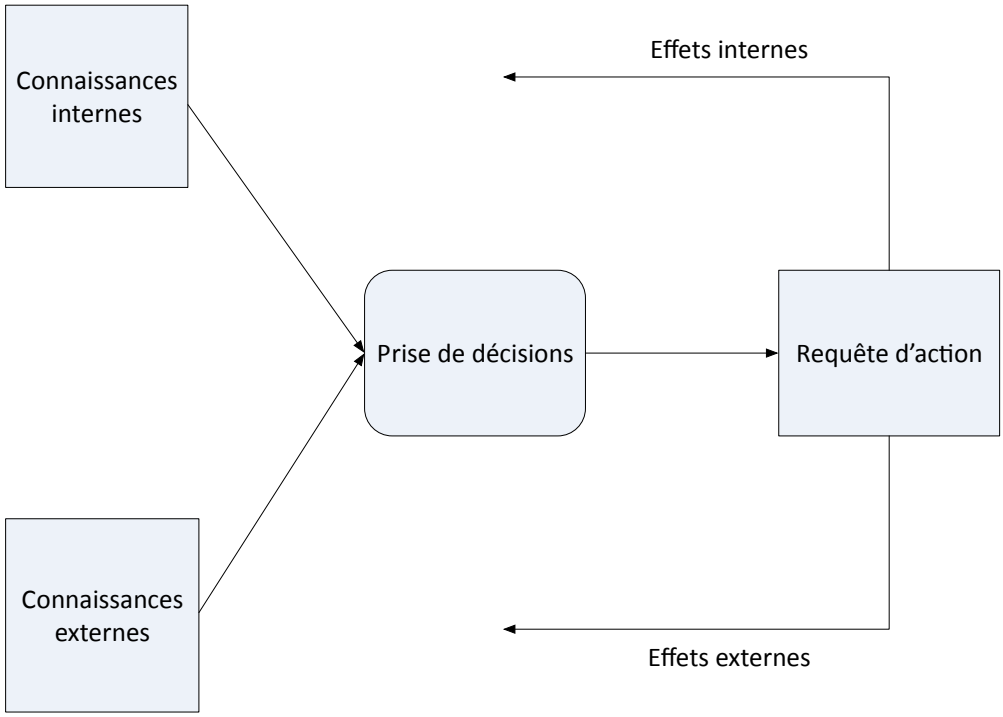


Figure 3.1 Exemple de schéma d'une architecture de prise de décision.

Le personnage possède des informations qu'il utilise pour choisir la prochaine action qu'il accomplira. Une architecture de prise de décision peut donc être modélisée par un système composé d'une seule entrée, les informations possédées par le personnage, et d'une seule sortie, l'action à accomplir. Les informations possédées par le personnage peuvent être divisées en deux sections : les connaissances internes et les connaissances externes. Les connaissances internes représentent les informations que le personnage possède sur son état interne : ses points de vie, son équipement, ses buts, ses actions passées, etc. Les connaissances externes représentent les informations que le personnage possède sur son environnement : la position des autres personnages, les objets qui l'entourent, etc. La sortie d'un système de prise de

décision, l'action à accomplir, possède elle aussi une composante interne et une composante externe. La composante externe de l'action à accomplir comprend tous les effets sur l'état externe du personnage : se déplacer dans une pièce, éliminer un adversaire, activer un interrupteur, etc. La composante interne de l'action à accomplir est moins évidente dans un jeu vidéo, mais peut s'avérer très importante pour certaines techniques de prise de décision. Celle-ci englobe tous les changements sur l'état interne du personnage : un changement des buts du personnage, un changement sur son modèle du monde, etc. Certaines actions ne possèdent qu'une composante externe, d'autres qu'une composante interne, mais la plupart ont les deux composantes.

Nous nous intéresserons au bloc « prise de décision » de la figure 3.1, c'est-à-dire la technique d'intelligence artificielle employée pour choisir l'action à accomplir en fonction des connaissances du personnage. Actuellement, plusieurs techniques différentes sont utilisées et celles-ci sont brièvement présentées ci-dessous. Il est important de mentionner que plus d'une de ces techniques peuvent être couplées dans un système de prise de décision. De plus, certaines des méthodes présentées ci-dessous ne représentent qu'une partie de l'algorithme de prise de décision et doivent être utilisées avec d'autres techniques pour compléter le système de prise de décision. Les méthodes basées sur la planification, qui sont au coeur de notre recherche, ne sont pas présentées ici. Elles font plutôt l'objet de la section suivante.

3.2.1 Écriture de scripts

L'écriture de scripts est une technique qui se définit par le fait de séparer la logique de l'intelligence artificielle du jeu lui-même. En effet, au début des années quatre-vingt-dix, les jeux devenaient d'une complexité telle qu'il était nécessaire de séparer le design des comportements des personnages du moteur du jeu. Dans ce cas, le comportement des personnages est implémenté dans des scripts créés par les artistes et les concepteurs de jeux plutôt que par les développeurs.

Il existe cinq niveaux d'écriture de scripts [97] :

- Niveau 0 : Tous les comportements sont figés dans le code dans le langage source (C/C++).
- Niveau 1 : Des données dans des fichiers fournissent des statistiques et la position des personnages et des objets.
- Niveau 2 : Seules les scènes où le joueur ne peut interagir sont scriptées.
- Niveau 3 : Une logique simple implémentée automatiquement par des outils ou dans des scripts.

  - Niveau 4 : Une logique complexe implémentée dans des scripts et qui utilise des fonctions implémentées dans l'engin en C/C++.
  - Niveau 5 : Tous les comportements sont entièrement scriptés dans un langage différent du langage source.

Les langages de scripts utilisés sont généralement Lua et Python, mais peuvent également être, entre autres, Tcl, Java, JavaScript, Ruby, Scheme ou un langage créé par la compagnie développant le jeu. Les scripts sont compilés en code machine avant l'exécution du jeu ou ils sont interprétés lors de l'exécution. L'écriture de scripts est une technique qui est encore largement utilisée. Ses avantages sont les suivants [79, 97] :

- Les comportements des personnages peuvent être modifiés et testés sans recompiler tout le jeu.
- Les habilités des concepteurs de jeux peuvent être exploitées sans l'aide de programmeurs.
- Les comportements sont à la portée des joueurs qui peuvent les modifier ou en créer de nouveaux. La modification de jeux génère des revenus importants et c'est pourquoi la plupart des grands titres possèdent une forme de langage de scripts.

Les désavantages de l'écriture de scripts sont les suivants [79, 97] :

- Difficile à déboguer et à gérer.
- Des gens qui ne connaissent pas la programmation sont forcés de programmer.
- Créer et supporter un langage de scripting et ses outils complémentaires requiert beaucoup d'efforts et de temps.

3.2.2 Machines à états finis

Souvent, les personnages d'un jeu possèdent un ensemble limité d'actions. Ils effectuent une action donnée jusqu'à ce qu'un certain événement survienne et qu'ils doivent y réagir. Ce type de comportement s'implémente parfaitement avec une machine à états finis.

Une machine à états finis (FSM) est composée d'un ensemble fini d'états et de transitions. Chaque transition possède un ensemble de conditions qui, lorsque vérifiées, permettent de transiter d'un état vers un autre. À tout moment, la machine ne peut être que dans un seul état actif. Dans cet état, elle vérifie constamment les conditions de toutes les transitions dans l'état. Dès que toutes les conditions d'une transition sont vérifiées, celle-ci est effectuée et la machine change d'état. Dans le cas d'un jeu vidéo, les états représentent les comportements adoptés par le personnage. La figure 3.2, tirée de [79], présente un exemple de machine à états finis simple pour un personnage.

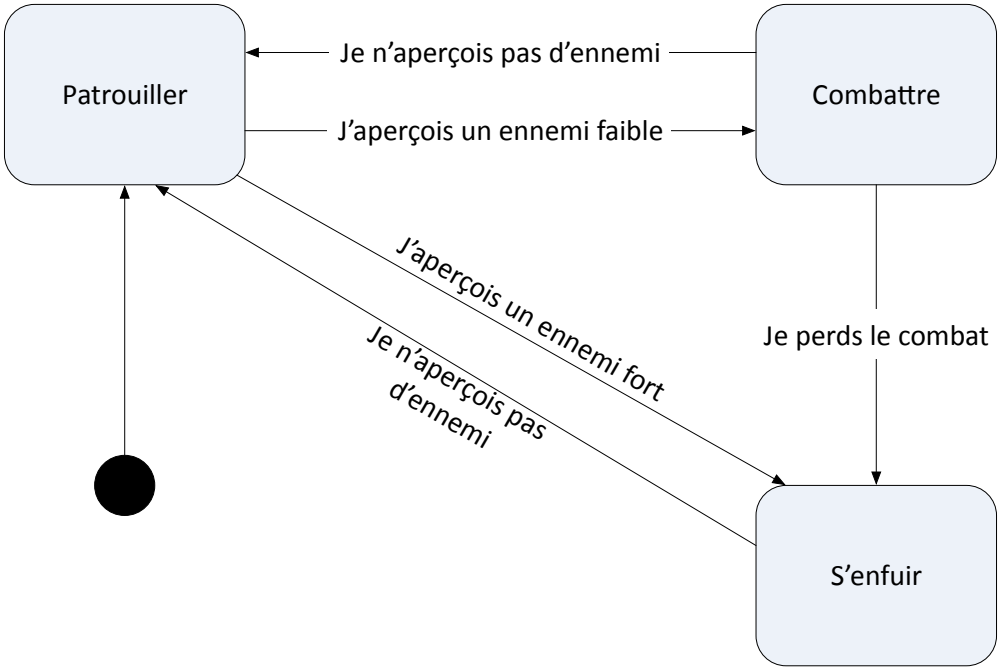


Figure 3.2 Exemple de machine à états finis simple.

Les machines à états finis et l'écriture de scripts représentent les deux techniques les plus utilisées pour implémenter les systèmes de prise de décision. Bien que les machines à états finis soient simples à écrire, elles sont extrêmement difficiles à maintenir. En effet, les machines à états dans les jeux comportent souvent des milliers d'états et leur code peut être complexe et difficile à comprendre. De plus, lorsque le comportement d'un personnage doit être modifié, il est possible qu'une grosse partie de la machine à états soit à refaire. Également, il est important de mentionner que le comportement de chaque personnage doit être codé individuellement lorsque les FSM sont utilisées. Ceci représente une lourde tâche et nécessite beaucoup de temps, d'autant plus que tout le code du jeu peut avoir à être recompilé lorsque le comportement d'un seul des personnages est modifié [79].

Les environnements très riches des jeux de nouvelle génération font en sorte que les comportements d'un personnage peuvent être difficiles à représenter dans une seule FSM. Il est alors possible d'utiliser des machines à états hiérarchiques, c'est-à-dire des machines à états qui contiennent des états qui sont en fait d'autres FSM [79]. De plus, il est également possible de diviser les machines à états finis en couches. Un tel système s'appelle architecture de subsomption. Les couches de bas niveau s'occupent des comportements rudimentaires tels que l'évitement des obstacles et les couches de haut niveau s'occupent des comportements plus élaborés tels que la détermination et la poursuite des buts. Puisque les niveaux inférieurs ont

une priorité plus élevée, le système demeure robuste et il s'assure que les exigences des niveaux inférieurs sont satisfaites avant de laisser les couches supérieures influencer le comportement du personnage [97]. Finalement, il existe également une variante des machines à états finis qui possède une certaine mémoire des états passés. Ce type de machines se nomme machines à états basées sur une pile, car elles conservent les états passés dans une pile. Dans un jeu, cette technique est importante lorsqu'un personnage veut reprendre l'action qu'il était en train d'effectuer avant d'avoir été interrompu pendant un instant [97].

3.2.3 Arbre de comportement

Un arbre de comportement tente de combiner les avantages d'une machine à états finis hiérarchique (présentée à la section 3.2.2), c'est-à-dire la facilité d'implémentation et un comportement réactif, et d'un planificateur HTN (présenté à la section 4.2.5), c'est-à-dire une représentation intuitive et un comportement autonome et orienté sur les buts [8]. Celui-ci peut être vu comme une machine à états finis hiérarchique à laquelle on aurait retiré les transitions entre les états. Une fois les transitions disparues, il ne s'agit plus vraiment d'états, mais plutôt de comportements, cest-à-dire une collection d'actions qui s'exécutent et se terminent, d'où le nom « arbre de comportement » [6].

La représentation d'un arbre de comportement est très semblable à une représentation HTN. Il s'agit d'un arbre composé de tâches qui se divisent récursivement en sous-tâches. Les tâches du premier niveau peuvent être associées aux sous-machines dans une machine à états finis hiérarchique. Les sous-tâches des niveaux inférieurs peuvent être associées aux états alors que les feuilles fournissent une interface entre l'intelligence artificielle et le moteur du jeu vidéo. Le comportement de l'agent est déterminé par les branches de l'arbre. En effet, les branches possèdent des conditions qui doivent être vérifiées pour qu'elles puissent être empruntées. Afin de générer la complexité dans le comportement de l'agent, des tâches composites sont utilisées. Celles-ci comprennent les séquences [5], les sélecteurs [4], les parallèles [2] et les décorateurs [7]. Une séquence définit une série de noeuds qui doivent être parcourus en ordre. Un sélecteur permet de choisir un seul noeud à parcourir parmi un ensemble de noeuds. Un parallèle permet d'exécuter plusieurs noeuds en même temps. Finalement, un décorateur permet d'ajouter une décoration à un noeud, c'est-à-dire un comportement supplémentaire qui sera effectué. A. J. Champandard [3] explique comment un arbre de comportement est parcouru. La figure 3.3 présente les tâches composites.

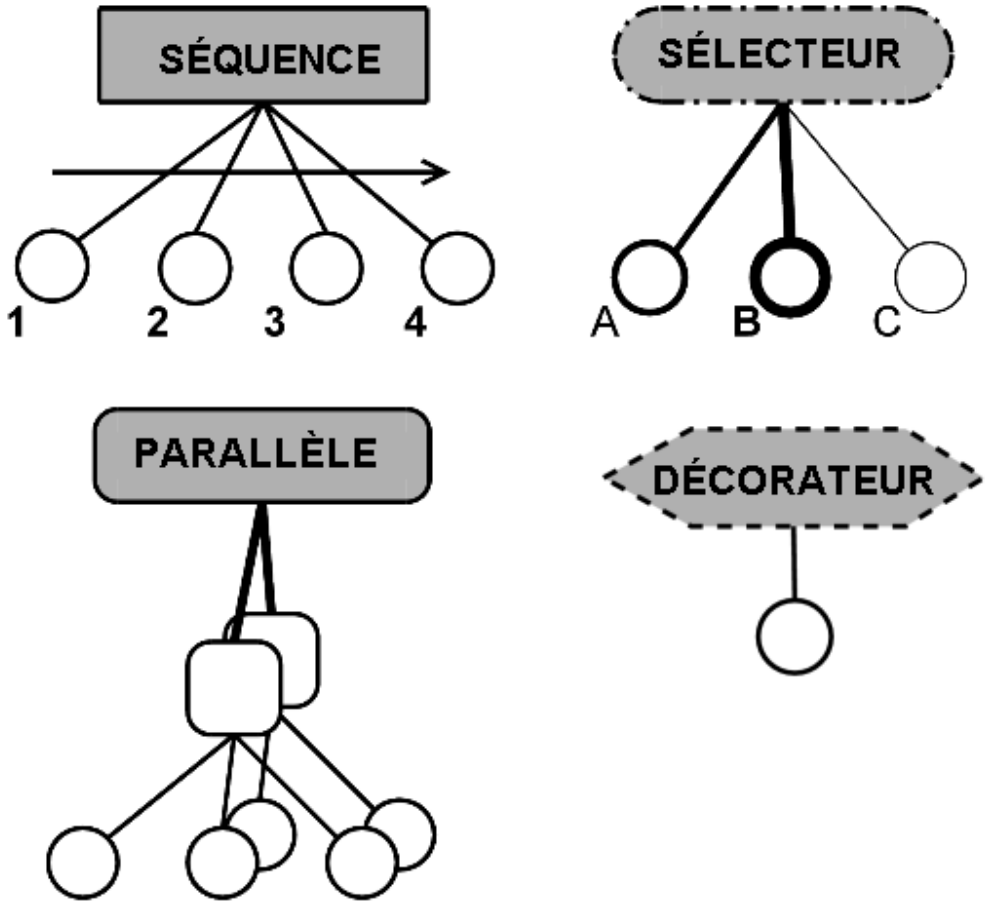


Figure 3.3 Tâches composites d'un arbre de comportement.

Pour un exemple appliqué à un jeu vidéo, D. Isla [61] décrit l'arbre de comportement qui a été implémenté dans le jeu Halo 2 (2004) [78].

3.2.4 Arbres de décision

Un arbre de décision est composé de points de décision, représentés par les nœuds, et d'actions, représentées par les feuilles. En partant de la décision initiale, la racine de l'arbre, l'arbre est parcouru selon un algorithme qui prend une décision à chaque nœud jusqu'à ce que le processus de prise de décision n'ait plus de décision à prendre, c'est-à-dire jusqu'à l'arrivée à une feuille. Lorsque l'algorithme arrive à une feuille, l'action qu'elle représente est immédiatement effectuée. Dans un contexte de jeu vidéo, chaque choix est effectué en fonction des connaissances du personnage. Puisque les arbres de décision son souvent utilisés comme mécanismes simples et rapides de prise de décision, les personnages utilisent normalement une représentation globale de l'état du jeu plutôt qu'une représentation individuelle pour

prendre leurs décisions tout au long du parcourt de l'arbre [79]. La figure 3.4, tirée de [79], présente un arbre de décision.

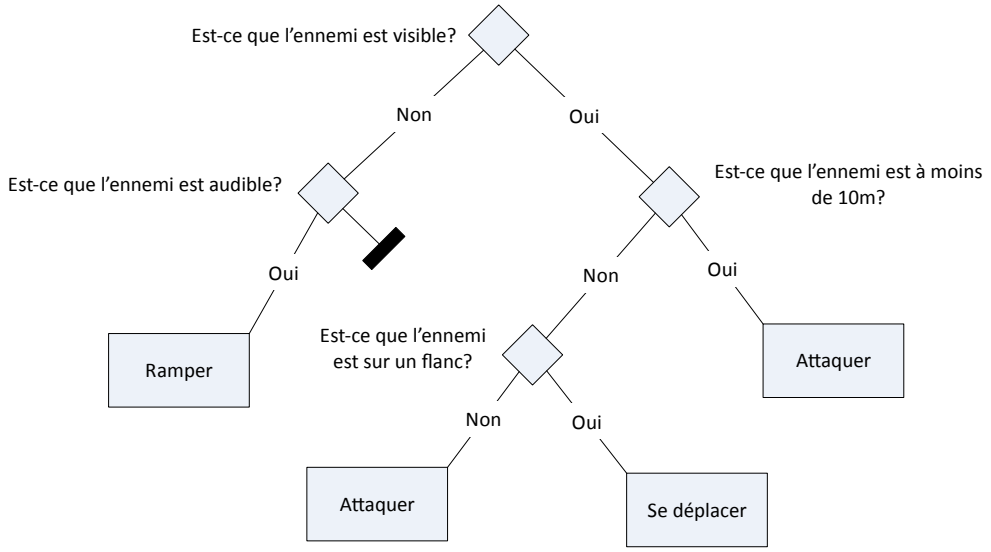


Figure 3.4 Exemple d'arbre de décision.

Les arbres de décision peuvent également être utilisés pour faire de l'apprentissage, ce qui est important dans les jeux puisqu'il existe des algorithmes très efficaces pour construire des arbres de décision en temps réel. Cette technique a notamment été utilisée dans le jeu Black & White (2001) [71] dans lequel une créature apprend grâce à un système de réactions positives et négatives de la part du joueur. Bien que Black & White ait démontré le pouvoir de l'apprentissage par arbres de décision dans un jeu vidéo, cette technique reste ignorée par le reste de l'industrie [98].

3.2.5 Systèmes basés sur les règles

Un système basé sur les règles, aussi appelé système expert ou système de production, est une architecture capturant le savoir d'un expert sous forme de règles. Le système se décompose en trois parties : une base de données contenant les connaissances utiles à l'intelligence artificielle, un ensemble de règles de la forme SI—ALORS et un système d'inférence. Pour chaque règle, la base de données est interrogée pour savoir si la condition SI est satisfaite. Le système d'inférence détermine quelle règle devrait être enclenchée, résolvant tout conflit entre plusieurs règles pouvant être enclenchées simultanément. Les systèmes basés sur les règles sont rarement utilisés, principalement parce que le même comportement peut être plus simplement généré en utilisant une machine à états finis ou un arbre de décision [79]. Néanmoins, ils sont

grandement utilisés dans les jeux de sports pour lesquels le personnage doit posséder une bonne quantité d'informations provenant d'un expert pour jouer correctement [98].

3.2.6 La logique floue

Les décisions basées sur la logique sont sans ambiguïté : la valeur des conditions et les décisions sont des éléments de l'ensemble {VRAI, FAUX}, et la ligne qui les divise n'est jamais questionnée. La logique floue est un ensemble de techniques mathématiques conçues pour faire face à ces zones grises.

En logique traditionnelle, la notion de prédicat est utilisée. Un prédicat représente un fait ou une caractéristique d'un objet. Par exemple, le terme prédicatif `affame (joueur1)` pourrait représenter le fait que le personnage désigné `joueur1` soit affamé. Par contre, il n'y a aucune notion de degré de famine : le personnage est ou n'est pas affamé. La logique floue développe la notion de prédicat en lui attribuant une valeur. Par exemple, un personnage peut être affamé avec une valeur de 0.5. De ce fait, il est possible qu'un personnage ne soit pas seulement affamé ou non, mais bien peu affamé, partiellement affamé, très affamé, etc. Les valeurs des prédicats varient généralement entre 0 et 1 et sont appelées degrés d'appartenance. Une valeur de 1 représente une appartenance complète alors qu'une valeur de 0 représente que le prédicat ne s'applique pas.

La logique floue peut être utilisée de plusieurs façons dans un système de prise de décision. Par exemple, elle peut être utilisée dans un système qui serait normalement composé de logique traditionnelle avec les opérateurs AND, OR et NOT. Elle peut également être utilisée pour déterminer quand les transitions dans une machine à états devraient être effectuées. Dans ce cas, les machines à états sont nommées machines à états flous. La logique floue peut aussi être utilisée dans un système basé sur les règles [79]. En d'autres termes, les possibilités d'utilisation de la logique floue dans une architecture de prise de décision sont très nombreuses.

3.2.7 Architecture de tableau noir

Une architecture de tableau noir n'est pas une technique de prise de décision en soi, mais plutôt un mécanisme pour coordonner les actions de différents agents de prise de décision. Les agents de prise de décision peuvent utiliser n'importe quelle technique pour effectuer leur tâche individuelle.

Le but d'une architecture de tableau noir est de résoudre un problème complexe de haut niveau en l'affichant sur un tableau. À tour de rôle, des agents intelligents experts dans un certain domaine regarderont alors le tableau et indiqueront l'intérêt qu'ils portent envers le problème. Un arbitre donnera par la suite le contrôle à un expert qui effectuera un travail, modifiera possiblement le tableau et proposera une solution partielle ou complète. Une fois sa solution donnée, l'arbitre donne le contrôle à un autre expert qui fera les mêmes étapes et ainsi de suite jusqu'à l'obtention d'une solution complète satisfaisante au problème de haut niveau. La figure 3.5, tirée de [79], présente une architecture de tableau noir pour un jeu vidéo.

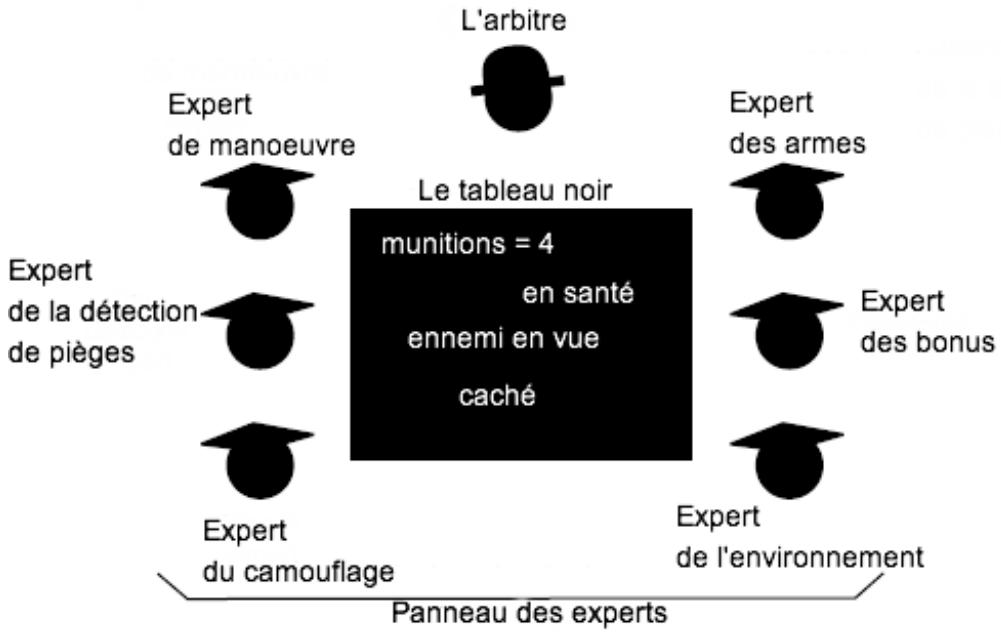


Figure 3.5 Exemple d'architecture de tableau noir.

Alternativement, une architecture de tableau noir peut être utilisée seulement comme un espace partagé de communication entre les différents personnages du jeu pour effectuer la coordination entre ceux-ci. Par exemple, si plusieurs personnages veulent attaquer un certain endroit, il est intéressant qu'ils connaissent les points d'attaques des autres pour attaquer de sources différentes, ce qui cause une attaque plus efficace [98]. Pour plus d'informations, D. Isla et B. Blumberg [62] fournissent une excellente explication des architectures de tableau noir.

3.2.8 Réseaux Bayesiens

Les réseaux Bayesiens permettent à un agent intelligent d'effectuer un raisonnement complexe en utilisant l'inférence probabiliste lorsqu'il fait face à une certaine incertitude de la même manière qu'un humain le ferait. Une bonne explication de réseaux Bayesiens est fournie par S. Russell et P. Norvig [99].

Une application intéressante des réseaux Bayesiens est la modélisation de ce qu'un personnage doit croire sur le joueur humain en fonction des informations qu'il possède. Par exemple, un personnage peut tenter d'inférer la présence ou non du joueur en utilisant ses connaissances sur l'environnement tel que la présence de certains sons, de certaines traces laissées par le joueur, etc. Ceci permet d'éviter que le personnage triche et permet au joueur de duper son ennemi en lui effectuant des actions trompeuses telles que lancer un objet pour générer un son dans une certaine direction [98].

3.2.9 Réseaux de neurones

Les réseaux de neurones sont des fonctions non linéaires complexes possédant une ou plusieurs entrées et une ou plusieurs sortie. Ils sont composés d'une seule couche, dans lequel cas ils sont appelés perceptrons, ou de plusieurs couches. Les réseaux de neurones peuvent également être entrainés avec différents ensembles d'entrées pour produire une certaine fonction. Une description détaillée des réseaux de neurones est fournie par S. Russell et P. Norvig [99].

Dans un jeu vidéo, les réseaux de neurones peuvent être utilisés pour contrôler certains personnages et effectuer leur apprentissage. Par exemple, dans le jeu Black & White, chaque désir d'une créature était représenté par un perceptron. Par exemple, un perceptron était utilisé pour représenter le désir de manger (la faim). En utilisant trois entrées, le perceptron détermine si la créature a faim. Si la créature mange, son perceptron est entrainé par des réactions positives ou négatives de la part du joueur [98]. Néanmoins, il y a relativement peu d'applications des réseaux de neurones dans les systèmes de prise de décision des jeux vidéo [98].

3.2.10 Prédiction statistique n-gramme

Un n-gramme est une technique statistique utilisée pour prédire la prochaine valeur dans une séquence. Lorsqu'une prévision est requise, la séquence est parcourue à reculons à la recherche de toutes les séquences identiques aux $n-1$ valeurs les plus récentes où n est

généralement 2 (digramme) ou 3 (trigramme). Lorsque les événements passés sont conservés dans une séquence qui est construite dans le temps, les événements futurs peuvent être prédits.

Il existe des exemples d'application de la prédiction statistique n-gramme dans une architecture de prise de décision pour les jeux de combat. Dans ce cas, une séquence d'une taille déterminée des coups passés effectués par le joueur est conservée par l'agent intelligent et est mise à jour dans le temps. Le prochain coup du joueur peut alors être prédit sur demande par l'agent en calculant des statistiques en temps réel [98].

3.2.11 Profilage du joueur

Le profilage du joueur est une technique qui permet de créer un profil des caractéristiques du joueur en temps réel dans le but d'adapter le niveau de jeu à ses forces et ses faiblesses. Pendant qu'il joue, des statistiques sur le joueur sont amassées et son profil est constamment mis à jour. Par exemple, si le profil du joueur démontre qu'il possède une faiblesse à utiliser une certaine arme, alors les personnages du jeu pourraient être appelés à exploiter cette faiblesse dans des niveaux de difficulté supérieurs du jeu [98].

3.2.12 Apprentissage machine

Plusieurs formes d'apprentissage peuvent être utilisées dans un système de prise de décision. Les deux techniques les plus employées sont l'apprentissage par renforcement et l'apprentissage par modification des faiblesses.

L'apprentissage par renforcement est une technique d'apprentissage qui permet à l'ordinateur de découvrir ses propres solutions à des problèmes complexes par essais et erreurs. En lui fournissant des récompenses et des punitions au bon moment, une intelligence artificielle basée sur l'apprentissage par renforcement peut apprendre à résoudre une variété de problèmes difficiles. Cette technique est particulièrement utile lorsque les effets des actions des personnages sont incertains [98].

L'apprentissage par modification des faiblesses a pour but d'empêcher qu'un personnage perde contre un joueur de la même façon plus d'une fois. Pour ce faire, l'état du jeu précédent l'échec du personnage ainsi que les actions effectuées par le personnage à ce moment sont enregistrés. Lorsque cet état se présente à nouveau, le personnage pourra alors modifier quelque peu ses actions pour que le même scénario ne se répète pas. Cette technique est particulièrement utile dans les jeux de sports, mais elle peut s'appliquer à presque tous les

genres [98].

CHAPITRE 4

La planification

La planification consiste à déterminer une séquence d'actions permettant de passer d'un état initial à un état désiré (le but). L'étude de la planification, qui a commencé dès les débuts de la recherche en intelligence artificielle, « [...] a conduit à de nombreux outils utiles pour les applications du monde réel et a permis de mieux comprendre l'organisation du comportement et le sens du raisonnement sur les actions » [9]. Un problème de planification est résolu par un système nommé *planificateur*. L'environnement d'un problème de planification peut être entièrement observable, déterministe et discret. Il s'agit alors de planification classique. Bien qu'il existe des planificateurs conçus pour résoudre des problèmes aux environnements stochastiques, nous ne nous intéresserons qu'à la planification classique.

4.1 La représentation d'un problème de planification

Afin de résoudre un problème de planification, le planificateur reçoit en entrée la représentation du problème, c'est-à-dire la représentation des états, des actions et des buts. Un langage fort utilisé pour créer une représentation propositionnelle, c'est-à-dire en utilisant uniquement la logique propositionnelle, d'un problème de planification est le langage STRIPS [34]. Le langage STRIPS peut également être utilisé pour représenter des problèmes en utilisant la logique du premier ordre. En STRIPS, les états sont décrits sous la forme d'une conjonction de littéraux positifs. Les littéraux peuvent être des littéraux propositionnels ou des littéraux du premier ordre sans variables ni fonction. De plus, l'hypothèse du monde clos est utilisée : toute condition qui n'est pas spécifiée dans l'état est supposée fausse [99]. Un but est satisfait lorsqu'on atteint un état qui contient tous les littéraux mentionnés dans celui-ci. Finalement, les actions sont représentées par leur nom, une liste de préconditions et une liste d'effets. La liste de préconditions est « une conjonction de littéraux positifs indiquant ce qui doit être vrai dans un état avant que l'action puisse être exécutée » [99]. La liste d'effets est une conjonction de littéraux positifs ou négatifs indiquant comment l'état doit être modifié lorsque l'action est exécutée : les littéraux positifs doivent être ajoutés à l'état s'ils ne s'y trouvent pas déjà alors que les littéraux négatifs indiquent les littéraux de l'état qui doivent être supprimés.

Afin d'augmenter l'expressivité du langage STRIPS et de pouvoir l'appliquer à plus de

domaines du monde réel, le langage ADL [93] fut créé. ADL permet de relaxer certaines restrictions de STRIPS en apportant plusieurs modifications au langage STRIPS, dont les suivantes :

- Les états peuvent contenir des littéraux négatifs.
- Les effets peuvent être conditionnels, c'est-à-dire que ceux-ci s'appliquent seulement si certaines conditions spécifiées sont satisfaites.
- Les buts peuvent contenir des disjonctions.
- L'hypothèse du monde ouvert est utilisée : toute condition qui n'est pas spécifiée dans l'état est supposée inconnue.

Les langages de représentation d'un problème de planification tels que STRIPS et ADL ont été standardisés dans un formalisme nommé PDDL [75] dont la dernière version est PDDL3 [39]. PDDL permet non seulement de créer des représentations propositionnelles en utilisant STRIPS ou ADL, mais également d'autres types de représentations tels que les représentations numériques et les représentations temporelles, et ce, dans une syntaxe standard. Les représentations numériques incluent des quantités numériques dans la définition du problème telles que des coûts associés aux actions. Les représentations temporelles permettent d'inclure la notion de temps au problème : par exemple, les actions peuvent avoir une certaine durée, certaines actions peuvent être effectuées en parallèle, la condition X sera vraie au temps Y, etc. Finalement, PDDL contient également un sous-langage pour représenter les problèmes des réseaux hiérarchiques de tâches qui seront expliqués à la section 4.2.5.

Afin de mieux comprendre la représentation d'un problème de planification, représentons le problème du monde des blocs, un exemple classique de planification, en STRIPS. Ce problème est constitué de trois cubes posés sur une table contentant quatre compartiments. Les blocs peuvent s'empiler les uns sur les autres et on ne peut placer directement qu'un seul bloc sur un autre bloc. Un seul bloc peut être pris à la fois, donc pour pouvoir déplacer un bloc, celui-ci doit être libre, c'est-à-dire qu'aucun autre bloc ne doit reposer sur lui. Le but sera toujours de construire une certaine pile de blocs à partir d'un certain état initial. Par exemple, nous pourrions vouloir résoudre le problème représenté à la figure 4.1.

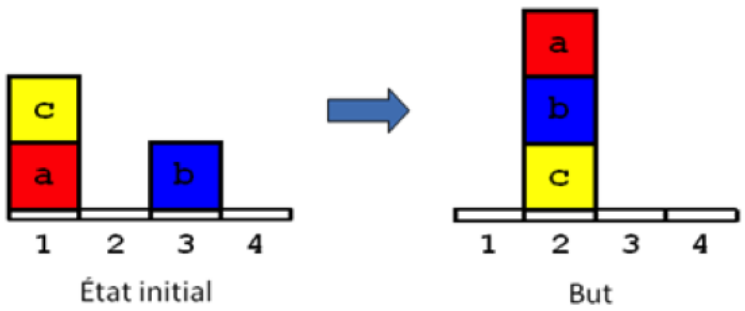


Figure 4.1 Exemple de problème de planification lié au monde des blocs.

Les littéraux du premier ordre utilisés pour représenter le problème sont présentés au tableau 4.1.

Tableau 4.1 Littéraux utilisés pour représenter le problème du monde des blocs.

Littéral	**Signification**
sur(X,Y)	Le bloc X se trouve sur l'endroit Y. Y peut être un bloc ou un emplacement sur la table.
libre(X)	L'objet X est libre, c'est-à-dire qu'il n'y a pas de bloc sur celui-ci.
bloc(X)	X est un bloc.
place(X)	X est un emplacement sur la table.
diff(X,Y)	X est différent de Y.

Les trois blocs sont les constantes a, b, c et les quatre emplacements sont les constantes p1, p2, p3, p4. Comme mentionné ci-haut, le problème est représenté par un état initial, un but et un ensemble d'actions.

État initial : `sur(a,p1) ∧ sur(c,a) ∧ sur(b,p3) ∧ libre(c) ∧ libre(p2) ∧ libre(b) ∧ libre(p4)`

But : `sur(c,p2) ∧ sur(b,c) ∧ sur(a,b) ∧ libre(p1) ∧ libre(p3) ∧ libre(p4) ∧ libre(a)`

Action : `deplacerSur(Bloc1,Orig,Dest)`

PRÉCOND : `bloc(Bloc1) ∧ sur(Bloc1,Orig) ∧ libre(Bloc1) ∧ libre(Dest) ∧ diff(Bloc1,Dest) ∧ diff(Orig,Dest)`

EFFETS : `sur(Bloc1,Dest) ∧ libre(Orig) ∧ ¬libre(Dest) ∧ ¬sur(Bloc1,Orig)`

On remarque que le problème ne comporte qu'une seule action, soit celle de déplacer un bloc (Bloc1) d'un endroit (Origine) vers un autre endroit (Destination).

4.2 Résolution d'un problème de planification

Il existe plusieurs techniques pour résoudre un problème de planification, dont les principales sont les suivantes :

- Exploration dans un espace d'états
- Planification en ordre partiel
- Utilisation de graphes de planification
- Traduction du problème en un problème de satisfaction de contraintes (CSP)
- Planification avec réseaux hiérarchisés de tâches
- Utilisation de techniques d'apprentissage machine

4.2.1 Exploration dans un espace d'états

L'approche la plus simple à la résolution d'un problème de planification est la recherche dans un espace d'états, également appelée planification totalement ordonnée. Puisque les actions comportent à la fois des préconditions et des effets, la recherche peut s'effectuer vers l'avant à partir de l'état initial ou vers l'arrière à partir du but [99]. La recherche dans un espace d'états est généralement guidée par une heuristique. Dans ce cas, ce type de planification est souvent appelé planification par recherche avec heuristique (HSP). La différence majeure entre les différents planificateurs utilisant la recherche par exploration dans un espace d'états comme méthode de résolution est l'heuristique utilisée et la manière de calculer cette heuristique. Il existe une multitude de planificateurs exploitant l'exploration dans un espace d'états pour résoudre un problème de planification dont Prodigy [15, 111], TLPlan [11], HPlan-P [12], FF [51], HSP [16, 17] et Fast Downward [47]. Finalement, il est à noter que lorsque l'espace des états d'un problème de planification est fini, par exemple en utilisant une représentation en STRIPS sans symboles de fonction, tout algorithme d'exploration qui est complet dans un graphe, tel que A*, sera également un algorithme de planification complet [99].

4.2.2 Planification en ordre partiel

L'exploration dans un espace d'états est aussi appelée planification totalement ordonnée puisqu'elle n'exploite que des actions linéaires, c'est-à-dire des actions directement connectées

à l'état initial dans le cas d'une recherche vers l'avant et des actions directement connectées au but dans le cas d'une recherche vers l'arrière. Ce type de recherche ne tire pas profit de la décomposition du problème [99]. La planification en ordre partiel (POP) travaille plutôt à résoudre plusieurs sous-problèmes importants de façon indépendante sans se soucier si une certaine action d'un sous-problème doit intervenir avant ou après une certaine action d'un autre sous-problème. Un plan partiellement ordonné est alors construit, c'est-à-dire un plan qui spécifie toutes les actions qui doivent être effectuées sans toutefois l'ordre exact dans lequel ces actions doivent être effectuées. Une fois le plan partiellement ordonné construit, un algorithme effectue sa linéarisation, c'est-à-dire place les actions dans le bon ordre, afin de créer le ou les plans totalement ordonnés. La planification en ordre partiel peut être implémentée comme une exploration dans l'espace des plans partiellement ordonnés où les actions ne sont pas des actions dans le monde, mais plutôt des actions sur les plans comme ajouter une étape au plan, imposer un ordre qui place une action avant une autre, etc. [99]. Deux exemples de planificateurs POP connus sont UCPOP [94] et SNLP [74].

4.2.3 Utilisation de graphes de planification

Les graphes de planification sont des structures de données construites à partir de la représentation d'un problème de planification. Ceux-ci contiennent plusieurs niveaux qui représentent les littéraux qui peuvent être vrais et les actions qui pourraient être effectuées à un certain instant. Un tel graphe a la propriété de pouvoir rapidement propager les informations utiles pour restreindre la recherche à mesure que celui-ci est construit [10]. Les informations contenues dans un graphe de planification peuvent être utilisées pour calculer des heuristiques d'exploration efficaces comme pour le planificateur AltAlt [86] ou encore pour extraire directement un plan solution comme pour les planificateurs Graphplan [14], STAN [72], IPP [70] et LPG [40, 41].

4.2.4 Traduction du problème en un CSP

Il est également possible de transformer un problème de planification en problème de satisfaction de contraintes (CSP) et d'appliquer un algorithme de satisfiabilité tel que WalkSAT ou l'algorithme DPLL pour trouver une solution. Des exemples de planificateurs utilisant cette méthode sont Satplan [64, 66], Blackbox [65] et CPlan [110].

4.2.5 Planification avec réseaux hiérarchisés de tâches

La planification avec réseaux hiérarchisés de tâches, ou la planification HTN, applique une décomposition hiérarchique à un problème de planification. Pour ce faire, un réseau

de tâches initial représentant les tâches de très haut niveau qui doivent être effectuées est d'abord fourni au planificateur. Chaque tâche est soit primitive ou composée. Une tâche primitive (aussi appelée action) peut être effectuée directement par l'agent, tandis qu'une tâche composée est un réseau d'autres tâches, c'est-à-dire un ensemble partiellement ordonné d'actions de plus bas niveau. Afin de générer un plan, le réseau de tâches initial est raffiné en décomposant les tâches composées jusqu'à ce que le plan ne contienne que des tâches primitives.

La représentation d'un problème HTN contient donc trois éléments [30, 31] :

- Un réseau de tâches initial d représentant le problème qui doit être résolu. Chaque tâche possède un nom et une liste d'arguments qui peuvent être variables ou constants. Le réseau de tâches contient également des contraintes qui indiquent par exemple dans quelle mesure les variables peuvent être liées (des préconditions), dans quel ordre les différentes tâches doivent être effectuées, etc.
- Un ensemble d'opérateurs Op indiquant les effets de chaque tâche primitive.
- Un ensemble de méthodes Me indiquant comment effectuer les tâches composées. Chaque méthode est une paire $m = (t, d)$ où t est une tâche et d un réseau de tâches. Ceci indique qu'une façon d'exécuter la tâche t est d'effectuer les tâches contenues dans d, ce qui peut être accompli en satisfaisant toutes ses contraintes.

Il est à noter que le terme *opérateur* peut également être utilisé pour idenfitier une tâche primitive et que le terme *méthode* peut également être utilisé pour identifier une tâche composée. Il est possible de représenter un problème de planification HTN avec PDDL.

Le principal avantage d'une représentation HTN face à STRIPS est que celle-ci est plus expressive [30, 31] et qu'elle permet de décrire les problèmes de façon plus naturelle [50]. De plus, la planification HTN permet de réduire l'espace de recherche d'exponentiel à linéaire sous certaines conditions [69]. Finalement, une caractéristique intéressante de la planification HTN est la replanification partielle. Contrairement à la planification totalement ordonnée, si un plan devient invalide durant la planification ou l'exécution, on peut se limiter à ne remplacer par un nouveau réseau de tâches que la partie du plan causant l'invalidité, plutôt que d'effectuer une replanification complète. Cette caractéristique puissante permet à la planification HTN d'être bien adaptée aux environnements dynamiques tels que ceux rencontrés dans les jeux vidéo [112].

Des planificateurs HTN connus sont : UMCP [31, 48], Nonlin [104], SIPE-2 [120], O-Plan [24], SHOP [85] et son successeur SHOP2 [84].

Afin de mieux comprendre le fonctionnement de la planification HTN, voyons un exemple tiré de [84]. Soit le problème de transport de colis par camion représenté à la figure 4.2.

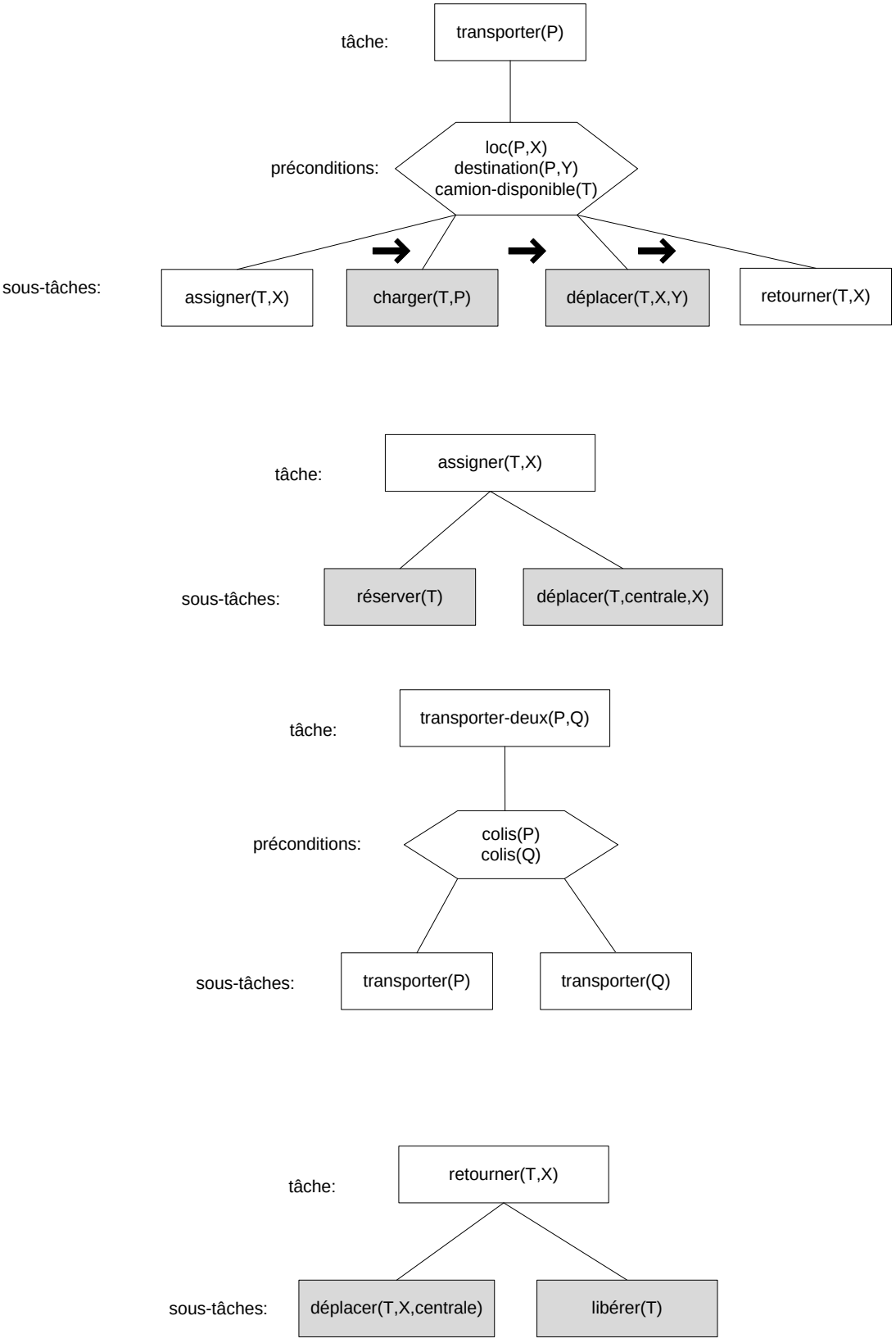


Figure 4.2 Réseau de tâches pour représenter un problème de transport de colis.

La figure 4.2 présente les méthodes pour transporter un colis `P`, transporter deux colis `P` et `Q`, assigner un camion `T` à une destination `X` et retourner un camion `T` d'une origine `X` à la centrale. Les lettres majuscules représentent les variables alors que ce qui est en minuscules sont les constantes. Les flèches représentent l'ordre dans lequel les sous-tâches doivent être effectuées. Les sous-tâches en gris sont des opérateurs : `charger(T,P)` charge le colis `P` dans le camion `T`, `déplacer(T,X,Y)` déplace le camion `T` de l'origine `X` à la destination `Y`, `réserver(T)` réserve un camion en effaçant `camion-disponible(T)` et `libérer(T)` libère un camion en ajoutant `camion-disponible(T)`. La figure 4.3 présente un plan pour accomplir la tâche de haut niveau `transporter-deux(p1,p2)` avec l'état initial suivant : {`colis(p1), loc(p1,i1), destination(p1,i3) camion-disponible(t1), loc(t1,centrale), colis(p2), loc(p2,i2), destination(p2,i4), camion-disponible(t2), loc(t2,centrale)`}. Il est possible de remarquer sur la figure 4.3 que les actions résultant de sous-tâches distinctes peuvent être intercalées, c'est-à-dire que « chaque décomposition ramène une action de haut niveau à un ensemble partiellement ordonné d'actions de plus bas niveau » [99]. Lorsque le plan ne contient que des tâches primitives, celui-ci est ordonné en utilisant les préconditions des tâches et des contraintes spécifiques à l'ordonnancement nommées contraintes d'ordre.

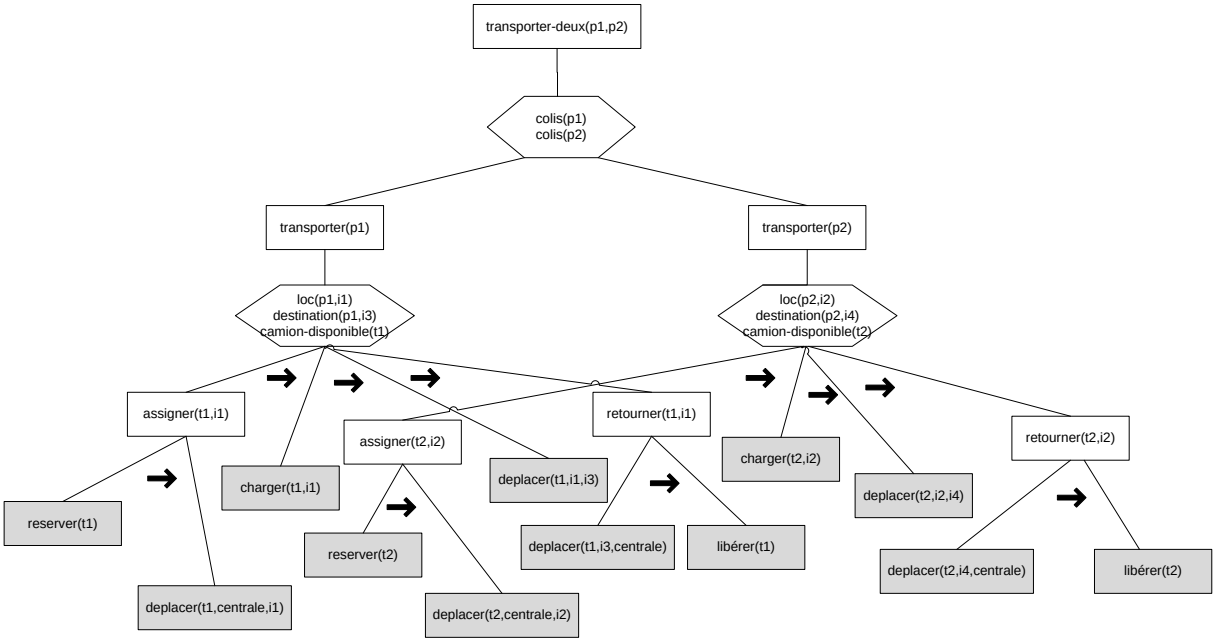


Figure 4.3 Solution au problème de transport de colis.

4.2.6 Utilisation de techniques d'apprentissage machine

La planification peut également être effectuée en utilisant des techniques d'apprentissage machine. Certaines techniques peuvent être employées pour résoudre un problème de planification et une technique importante parmi celles-ci est la planification basée sur les cas.

Pour toutes les techniques de planification présentées jusqu'à présent, la planification peut être vue comme un processus de raffinement de plans jusqu'à l'obtention du plan final. La planification basée sur les cas (ou CBP), technique dérivée du raisonnement basé sur les cas, exploite une méthode différente : plutôt que de chercher à raffiner un plan, d'anciens cas sont adaptés pour résoudre de nouveaux problèmes [23]. Pour ce faire, plusieurs cas sont amassés et enregistrés durant une phase d'entraînement. Un cas est représenté par un problème de planification ainsi que le plan qui fut utilisé pour le résoudre. Suite à cette phase d'entraînement, lorsque le planificateur basé sur les cas est confronté à un problème, il tentera de

déterminer le cas le plus semblable, puis il récupérera ce plan et l'adaptera pour trouver une solution valide. Ce qui distingue les différents planificateurs basés sur les cas est la méthode utilisée pour récupérer les cas, c'est-à-dire calculer la ressemblance entre des cas, et la technique d'adaptation des plans. Les planificateurs basés sur les cas comprennent CHEF [45], CABALA [27] et RepairSHOP [113].

La planification basée sur les cas peut également utiliser des macros. Dans ce cas, il ne s'agit pas de plans complets qui sont récupérés, mais plutôt des macros, c'est-à-dire des séquences d'opérateurs. Ces macros peuvent par la suite être utilisées pour guider une recherche par heuristique dans le but d'accélérer le processus de planification. Des exemples de planificateurs utilisant les macros sont Macro-FF [18] et Macro-AltAlt [83].

Finalement, d'autres techniques d'apprentissage peuvent être utilisées pour résoudre un problème de planification, notamment l'apprentissage de politiques [26, 37, 38], c'est-à-dire une liste de règles générales qui spécifient quelles actions peuvent être exécutées sous quelles conditions, qui agit comme heuristique afin de guider la recherche.

4.3 Caractéristiques d'un planificateur

Bien que la méthode utilisée afin de résoudre un problème de planification soit une caractéristique majeure d'un planificateur, il existe d'autres particularités qui les différencient. Les plus importantes sont les suivantes :

- Optimalité versus sous-optimalité : certains planificateurs retournent toujours la solution optimale, c'est-à-dire le plan qui permet de passer de l'état initial à l'état final en un nombre minimal d'actions dans le cas où toutes les actions ont le même coût ou n'ont pas de coût associé, ou en utilisant la suite d'actions la moins coûteuse possible dans le cas où les actions ont des coûts différents. Ces planificateurs se nomment planificateurs optimaux. D'autres planificateurs retournent plutôt une solution permettant de passer de l'état initial à l'état final sans se soucier de l'optimalité de cette dernière. Ce type de solution est suffisant pour plusieurs domaines d'application réels et les planificateurs qui le produisent sont appelés planificateurs sous-optimaux.
- Représentation du problème à résoudre : tel que mentionné à la section 4.1, il y a plusieurs façons de représenter un problème de planification : représentation propositionnelle, représentation numérique, représentation temporelle, etc. Le ou les types de représentation supportés par un planificateur sont propres à celui-ci. Par exemple, certains planificateurs ne peuvent résoudre que des problèmes propositionnels, alors

que d'autres peuvent résoudre des problèmes numériques et temporels seulement, etc. De plus, bien que les planificateurs discutés dans cette section soient uniquement des planificateurs classiques, il ne faut pas oublier qu'un problème de planification peut également être non-déterministe ou stochastique. Il existe également une gamme de planificateurs adaptés à ce style de problème.

- Type de langage de représentation utilisé : bien que la plupart des planificateurs sont capables de résoudre un problème représenté en PDDL, certains d'entre eux ne supportent pas toutes les particularités de celui-ci. Par exemple, certains planificateurs acceptent les problèmes représentés en STRIPS, mais refusent les problèmes représentés en ADL. D'autres acceptent une représentation ADL, mais ne supportent pas les effets conditionnels. De plus, puisque le langage PDDL est parfois mis à jour, ce ne sont pas tous les planificateurs qui supportent les dernières fonctionnalités.

CHAPITRE 5

Planification dans les jeux vidéo

La planification a commencé à être explorée par les développeurs de jeux vidéo au début des années deux mille. Depuis ce temps, les compagnies et la communauté scientifique des divertissements numériques lui ont découvert plusieurs applications et elle est maintenant utilisée à plusieurs fins dans les jeux vidéo. Suite à une revue exhaustive des travaux antérieurs, il nous a été possible de créer une taxonomie de l'utilisation de la planification dans les jeux vidéo. Celle-ci est présentée à la figure 5.1.

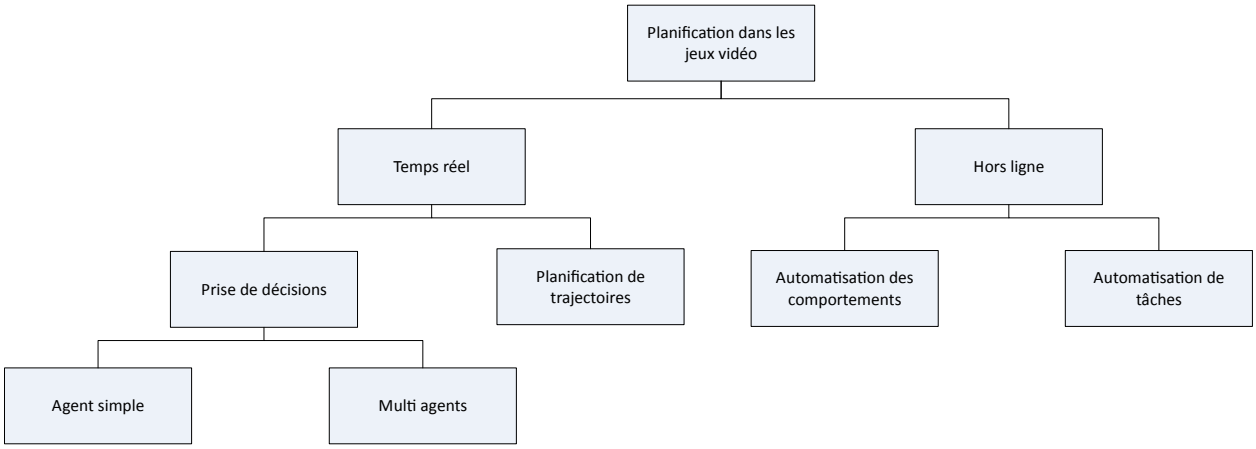


Figure 5.1 Taxonomie de l'utilisation de la planification dans les jeux vidéo.

La planification en temps réel signifie que celle-ci est effectuée pendant l'exécution du jeu. En général, les plans sont invalidés après un certain temps de jeu et la planification doit être effectuée de nouveau pour trouver un nouveau plan. Il s'agit alors de replanification. La planification en ligne comporte la prise de décision et la planification de trajectoires. La prise de décision régit le comportement des agents intelligents d'un jeu tandis que la planification de trajectoires tente de déterminer le chemin optimal entre un point de départ et un point d'arrivée.

Lorsque la planification n'est pas effectuée lors du déroulement du jeu, on parle alors de planification hors ligne. La planification hors ligne est principalement utilisée pour effectuer l'automatisation des comportements et l'automatisation de tâches. Comme il a été mentionné dans la section 3.2, les techniques d'IA les plus utilisées dans les architectures de prise de

décision des jeux vidéo indiquent d'avance le comportement que doivent adopter les personnages non-joueurs (PNJ) pour toutes les situations possibles auxquelles ils seront confrontés. Bien que cette méthode entraîne certains problèmes, elle comporte également ses avantages et est désirable dans certains cas. Il est possible d'implémenter l'automatisation des comportements en utilisant la planification hors ligne. Pour ce faire, avant le déroulement du jeu, des plans sont formulés et testés pour les personnages dans différentes situations. Lors du déroulement d'une partie, les PNJ effectueront la séquence d'actions dictée par le meilleur plan dans une situation donnée sans effectuer de replanification, automatisant ainsi leur comportement. Également, bien souvent, les développeurs et les concepteurs de jeux sont confrontés à des corvées ardues et répétitives telles que la création de scripts, de machines à états finis, de dialogues, etc. Dans ce cas, la planification hors ligne peut être utilisée pour automatiser ces tâches, réduisant ainsi considérablement la quantité de travail des développeurs et des concepteurs et assurant une validité du travail produit.

Puisque le présent projet porte sur la prise de décision, l'accent sera mis sur les travaux antérieurs portant sur cet aspect de l'intelligence artificielle d'un jeu vidéo. Néanmoins, les autres travaux sur la planification dans les jeux vidéo seront tout de même présentés, mais de façon plus brève, à titre de références pour le lecteur intéressé par ces domaines et pour connaître les tendances de la recherche dans ces secteurs. Pour présenter les travaux antérieurs de manière cohérente avec la section 4, ceux-ci seront divisés selon le type de planificateur utilisé.

5.1 Planification par exploration dans un espace d'états

La planification par exploration dans un espace d'états est certainement le type de planification le plus utilisé dans les jeux vidéo. En effet, on la retrouve dans toutes les applications de la planification présentées sur la figure 5.1, soit la prise de décision, la planification de trajectoire, l'automatisation de comportements et l'automatisation de tâches.

L'utilisation la plus marquée d'un planificateur effectuant une exploration dans un espace d'états dans une architecture de prise de décision est probablement l'intégration de l'algorithme A* dans une architecture basée sur les buts nommée « Goal-Oriented Action Planning » (GOAP) par J. Orkin. GOAP est une extension d'une manière d'implémenter une architecture de prise de décision basée sur les buts nommée « Goal-Oriented Behavior » (GOB) [79]. GOB est un terme général couvrant toutes les techniques d'implémentation d'architecture de prise de décision qui prennent en compte les buts et les intentions des personnages. Dans

une architecture GOB, l'objectif est de satisfaire les désirs internes des agents. Les techniques utilisées pour atteindre cet objectif peuvent être n'importe laquelle des techniques présentées à la section 3.2. Lorsque la planification est utilisée, on parle alors de « Goal-Oriented Action Planning ». J. O'Brien [87] fournit une excellente description de d'une architecture GOAP.

Toute architecture GOB comprend deux caractéristiques centrales : les buts et les actions. Chaque personnage possède un ou plusieurs buts ou motifs, dont un nombre quelconque sont actifs. Chaque but possède une valeur d'insistance représentée par un nombre. Un but avec une valeur d'insistance plus élevée influencera de manière plus importante le comportement du personnage. Le personnage agit donc de manière à satisfaire ses buts en réduisant leur valeur d'insistance. Une valeur d'insistance égale à zéro signifie que le but est complètement satisfait. En plus des buts, chaque personnage possède une série d'actions qu'il peut effectuer. Les actions disponibles dépendent de l'état courant du jeu. Chaque action modifie la valeur d'insistance de certains buts et le choix de l'action est fait en fonction de ses effets. Dans le cas de GOAP, une nouvelle caractéristique s'ajoute au modèle : le plan. Le plan représente la séquence d'actions valides permettant de passer de l'état courant du jeu vers un état final où les buts du personnage sont satisfaits. Celui-ci est déterminé par un planificateur. Plusieurs types de planificateurs peuvent être utilisés dans une architecture GOAP.

J. Orkin [89] explique l'architecture GOB qui a été utilisée pour le jeu No One Lives Forever 2 : A Spy in H.A.R.M.'s Way (NOLF2, 2002) [80], un jeu de tir à la première personne, et les problèmes d'une telle architecture qui peuvent être résolus par la planification GOAP. Dans NOLF2, l'architecture de prise de décision est de type GOB, mais n'utilise pas de planificateur. Chaque personnage possède environ 25 buts et, contrairement à une architecture GOB classique, un seul but peut être actif à la fois. Par contre, la plus grande différence entre les buts d'une architecture GOAP et ceux de NOLF2 est que ces derniers comportent tous un plan intégré. Lorsqu'un but est actif chez un personnage, celui-ci effectue une séquence d'actions prédéterminées. Le plan peut contenir des branches conditionnelles, mais ces conditions sont déjà déterminées au moment où la séquence d'actions du but s'enclenche. Ceci entraîne un certain nombre de problèmes qui peuvent être résolus en utilisant une architecture GOAP. En premier lieu, il est difficile d'imaginer toutes les situations possibles pour des plans prédéterminés et un grand nombre de situations créent des plans qui deviennent rapidement ingérables. Deuxièmement, les plans créés à la main peuvent contenir des anomalies, alors que ceux créés par un planificateur sont assurés d'être sans aberrations. Finalement, des changements de conception minimes peuvent entraîner des changements majeurs au code de l'intelligence artificielle. Par exemple, chaque plan prédéterminé doit être vérifié et pos-

siblement refait dans le cas où il y a un changement dans les actions possibles des personnages.

Non seulement une architecture GOAP résout les problèmes entraînés par les plans figés dans le code de NOLF2, mais elle comporte également plusieurs avantages par rapport aux autres techniques d'IA utilisées dans une architecture de prise de décision telles que les machines à états finis ou les systèmes basés sur les règles. Ces avantages sont les suivants [89, 90, 91, 92] :

- Une architecture GOAP est modulaire, c'est-à-dire que les buts et les actions sont découplés. Ce découplage facilite le partage de comportements et permet aux développeurs de décider quels buts et actions seront disponibles pour un PNJ en particulier ou pour tous les PNJ du jeu. Inversement, il permet de décider quels buts et actions doivent être exclus du comportement d'un certain PNJ, ce qui évite la prolifération de drapeaux pour vérifier certaines conditions (telles que `PeutNager`, `PeutVoler`, `EstEnVoiture`, etc.) typiquement retrouvées dans l'IA des jeux vidéo.
- Afin que le planificateur détermine la suite d'actions à effectuer pour atteindre un certain but, tous les buts et toutes les actions du système doivent partager une certaine structure particulière. La similarité entre les différents modules de l'IA dans ce cas réduit l'effet des styles de programmation différents des développeurs et facilite la collaboration entre ceux-ci.
- Un problème majeur des techniques traditionnelles est que les concepteurs de jeux doivent prendre des décisions par rapport à l'IA du jeu puisque les conditions des règles dépendent du type de PNJ qu'ils créent. L'architecture GOAP modulaire crée une séparation entre l'implémentation et les données du jeu. Les ingénieurs implémentent les actions et les buts et décident de leurs préconditions et de leurs effets. Les concepteurs de jeux utilisent par les suites des fichiers de données pour spécifier quels buts et quelles actions sont disponibles pour un certain type de PNJ. Ceci leur permet de penser à ce que les PNJ peuvent faire, plutôt que de se soucier de quand et comment ils doivent le faire.
- Une architecture GOAP ne requiert pas l'écriture de règles pour chaque situation à laquelle un PNJ peut faire face. Des solutions raisonnables sont déterminées par le PNJ grâce aux préconditions et aux effets des actions placées en ordre afin de satisfaire un but même dans les situations auxquelles les développeurs n'avaient pas pensé. Ceci présente un avantage fort intéressant étant donné la complexité grandissante des mondes dans lesquels les PNJ doivent évoluer. De plus, il est important de mentionner que cette caractéristique d'une architecture GOAP donne aussi lieu à la replanification, ce qui permet à un PNJ de trouver une nouvelle séquence d'actions pour atteindre un but

lorsque celui-ci est complété, lorsqu'il devient invalide ou lorsqu'un autre but devient plus important.

Une architecture GOAP a été mise en pratique en 2006 dans le FPS à succès F.E.A.R. [1]. J. Orkin [90, 91, 92] explique son architecture de prise de décision. L'architecture est composée d'une machine à états finis ne comportant que deux états (techniquement trois, car un état est doublé pour cause de compatibilité avec les structures de données), `GoTo` et `Animate`, et d'un planificateur A*. En y songeant bien, un agent intelligent d'un jeu ne fait que se déplacer et jouer des animations. La seule difficulté réside dans les transitions entre les différents états et les paramètres de ceux-ci. À quelle destination devrait aller le personnage ? Où se trouve la destination ? Quelle animation devrait être jouée ? Devrait-elle être jouée une seule fois ou en boucle ? La logique servant à répondre à ces questions est découplée de la machine à états dans F.E.A.R. et elle se trouve dans une architecture GOAP.

Dans F.E.A.R., le système d'IA devait permettre le combat en temps réel de 10 PNJ au maximum. Cette exigence en termes de performance signifie que le système doit déterminer un plan rapidement et que la fréquence de replanification doit être faible. Afin de s'assurer que ces objectifs soient atteints, plusieurs techniques ont été utilisées. Premièrement, l'état courant du monde est représenté par un tableau de symboles de taille fixe. Les symboles représentent des propriétés du monde. Chaque PNJ possède sa vision symbolique du monde propre à lui-même. L'état courant du monde possède des symboles identifiant des propriétés telles que la position du PNJ, les armes qu'il possède, le nombre de munitions qu'il possède, ses cibles et l'énergie de celles-ci. En second lieu, les actions et les buts sont assignés aux différents types de PNJ par les concepteurs de jeux. Les ingénieurs se chargent de les implémenter. Pour ce faire, le formalisme STRIPS est utilisé. Chaque but spécifie certaines valeurs aux symboles de l'état du monde et les effets de chaque action sont spécifiés comme un sous-ensemble des symboles de l'état du monde qu'elle modifie. Le planificateur recherche alors des actions qui ont les effets désirés pour satisfaire un but donné. Les actions peuvent aussi posséder des préconditions, qui sont également spécifiées comme un sous-ensemble des symboles de l'état du monde. Afin d'accélérer la recherche, les actions sont placées dans une table de hachage selon les symboles du monde qu'elles affectent. Une heuristique de recherche visant à réduire le nombre d'actions nécessaire pour satisfaire les symboles non satisfaits est également utilisée. La planification est effectuée de nouveau seulement lorsque le plan courant est invalidé ou lorsqu'un des buts principaux du PNJ change. La fréquence de replanification est donc différente pour chaque PNJ, mais elle est toujours bien moindre qu'à chaque image générée. J. Orkin [91] fournit une description plus détaillée de l'implémentation de l'architecture.

H. Chan et al. [20] présentent une autre utilisation intéressante de la planification par exploration dans un espace d'états dans une architecture de prise de décision. En effet, l'article propose une méthode innovatrice afin de planifier la phase de production des jeux de type Real-Time Strategy (RTS) à l'aide d'un planificateur par exploration dans un espace d'états. L'architecture proposée est constituée d'un planificateur, d'un ordonnanceur avec heuristique et d'un algorithme de recherche en arbre de type « Meilleur d'abord » avec heuristique. Le planificateur trouve d'abord un plan pour arriver à un certain but donné et l'ordonnanceur modifie ensuite le plan pour que le parallélisme soit le plus efficace possible. Ces étapes sont répétées à chaque nœud d'un arbre de recherche dans l'espace des buts intermédiaires du planificateur.

Puisque le problème de planification de trajectoires peut être représenté par un graphe, une exploration dans un espace d'états se prête bien à ce type de problème. Spécifiquement, l'algorithme A* et ses variantes sont le plus souvent utilisés pour réaliser cette tâche [19, 79, 92]. La plupart des recherches tentent donc d'améliorer A* ou de créer des variantes plus efficaces pour la planification de tâches. Par exemple, C. Hernandez et P. Meseguer [49] proposent une variante de A* pour laquelle le nombre d'actions planifiées à chaque itération dépend de la qualité des valeurs de l'heuristique déterminées. M. R. Jansen et N. R. Sturtevant [63] utilisent également une variante de A* pour effectuer de la planification de trajectoires coopérative efficace entre plusieurs agents d'un RTS. Finalement, S. Fernandez et al. [33] présentent une approche intéressante au problème de planification de trajectoires basée sur l'intégration de la planification de trajectoires et de la planification de tâches. Dans tous les cas, il est possible de remarquer que la recherche au niveau de la planification de trajectoires gravite essentiellement autour de l'algorithme A*.

Tel que mentionné, la planification par exploration dans un espace d'états est également utilisée dans l'automatisation des comportements. T. Fayard [32] présente une application intéressante de l'automatisation des comportements. Le but de l'étude est d'utiliser un planificateur pour équilibrer les races d'un RTS, StarCraft [13].

Finalement, la planification par exploration dans un espace d'états est également utilisée dans l'automatisation de tâches telles que la narration interactive et la génération automatique de niveaux de jeu. L'automatisation de tâche par la planification hors ligne possède une multitude d'applications dans le domaine du jeu vidéo. Une des applications les plus importantes est certainement la génération d'histoires au niveau de la narration interactive,

aussi appelée drame interactif. Dans les jeux vidéo traditionnels, le joueur suit généralement une histoire séquentielle choisie au préalable par les concepteurs de jeux. Il est possible que quelques choix s'offrant au joueur modifient le cours de l'histoire, mais ces versions alternatives sont toutes fixes et ont été pensées préalablement. Dans un drame interactif, le joueur peut activement participer à l'histoire et modifier son cours par ses actions. Chaque nouvelle partie peut donc donner lieu à une histoire unique. Par contre, il a été démontré que générer assez de scénarios afin de créer un environnement hautement interactif permettant au joueur de se sentir engagé dans l'histoire est souvent impraticable pour des auteurs humains [101]. De ce fait, la planification est souvent utilisée pour créer automatiquement une multitude de branches d'histoires possibles et valides. H.-M. Chang et V.-W. Soo [21] s'intéressent à la narration interactive en tentant de résoudre le problème de la génération d'histoires non structurées et non cohérentes.

Malgré l'intérêt grandissant pour les techniques de narration interactive, leur relation avec les jeux vidéo traditionnels reste encore à être explorée. Plusieurs concepteurs de jeux ont exprimé des inquiétudes par rapport à l'utilisation de ces méthodes dans les jeux vidéo traditionnels, principalement à cause du manque de contrôle qu'ils auraient sur le contenu généré dynamiquement. D. Pizzi et al. [95] proposent donc une utilisation nouvelle des techniques de la narration interactive : la création automatique de niveaux de jeu. Précisément, celle-ci consiste en la génération de toutes les solutions possibles d'un niveau de jeu donné. Les solutions, qui sont générées grâce à la planification, permettent de déterminer la jouabilité finale d'un niveau et peuvent être visualisées sous forme de bandes dessinées avec lesquelles les concepteurs de jeux sont habitués de travailler.

5.2 Planification en ordre partiel

La planification en ordre partiel est principalement utilisée dans le domaine des jeux vidéo pour automatiser des tâches. En effet, tel que démontré par M. Si, S. C. Marsella et M. O. Riedl [101], elle peut être appliquée au domaine de la narration interactive. L'article présente un logiciel intégré qui utilise un système multi-agents pour contrôler les personnages d'une histoire. La planification en ordre partiel permet également de générer automatiquement des résumés visuels de sessions de jeux. Y.-G. Cheong et al. [22] proposent un système nommé ViGLS (**V**isualization of **G**ame **L**og **S**ummaries) qui génère des résumés visuels de sessions de jeux à partir des traces du jeu (game logs) en utilisant la planification en ordre partiel.

5.3 Planification avec réseaux hiérarchisés de tâches

L'utilisation de la planification avec réseaux hiérarchisés de tâches dans le domaine du divertissement numérique se fait principalement au niveau de la prise de décision et de l'automatisation de tâches.

Tout d'abord, P. Gorniak et I. Davis [43] et H. Munoz-Avila et H. Hoang [82] démontrent que la planification HTN peut être utilisée pour coordonner une petite équipe d'agents. H. Hoang, S. Lee-Urban et H. Munoz-Avila [50] et H. Munoz-Avila et T. Fisher [81] utilisent la même technique pour coordonner une équipe d'agents du FPS populaire Unreal Tournament [29], appelés « Bots » pour ce jeu précis.

L'architecture client-serveur présente dans Unreal Tournament (UT) possède un environnement de programmation événementielle. Selon ce type de programmation, les PNJ (appelés Bots dans UT) réagissent aux changements dans l'environnement. Un des problèmes de la programmation événementielle est la difficulté de formuler une stratégie globale de haut niveau tandis que chaque Bot individuel doit réagir à un environnement toujours changeant. L'équipe propose d'utiliser un planificateur HTN afin que les Bots soient capables de réagir à un environnement très dynamique tout en pouvant participer à une stratégie globale de haut niveau.

Le but est de créer une stratégie de groupe plutôt que des comportements indépendants pour chaque Bot. Pour ce faire, l'équipe a choisi d'utiliser le planificateur SHOP [85] pour leur implémentation du Bot. Le planificateur détermine la stratégie globale que doivent adopter tous les membres de l'équipe. Le plan résultant consiste en une série d'actions que doivent effectuer les Bots. Pour effectuer ces actions, l'équipe a décidé d'utiliser une implémentation événementielle en Java des Bots déjà existante. Cette implémentation contrôle normalement les Bots à l'aide d'une intelligence artificielle basée sur les machines à états finis. Cette implémentation a été modifiée afin qu'elle puisse effectuer les tâches primitives assignées par les plans déterminés par le planificateur HTN. Le comportement individuel de chaque Bot est donc exécuté grâce à une machine à états finis. De ce fait, grâce au planificateur, les Bots sont capables de suivre une stratégie d'équipe globale de haut niveau tout en réagissant aux événements dynamiques non déterministes grâce à l'implémentation événementielle de leurs actions.

Également, la planification avec réseaux hiérarchisés de tâches peut être utilisée pour générer automatiquement des machines à états de dialogues. En effet, une autre application

de la planification afin d'automatiser les tâches se trouve au niveau de la création de dialogues dans les jeux vidéo. Ceux-ci sont actuellement créés par des équipes d'écrivains et de concepteurs de jeux qui écrivent manuellement toutes les lignes de texte et les machines à états finis spécifiant la structure des dialogues. Afin d'automatiser cette tâche, C. R. Strong et M. Mateas [103] proposent un système qui permet de créer automatiquement les machines à états de dialogues en utilisant un planificateur HTN pour créer des structures de dialogues puisqu'une conversation se décompose bien hiérarchiquement.

Finalement, la planification avec réseaux hiérarchisés de tâches peut être utilisée pour générer automatiquement des scripts. En effet, la planification en temps réel est parfois un processus trop exigeant en termes de temps de calcul pour certains jeux vidéo. Dans cette optique, J.-P. Kelly, A. Botea et S. Koenig [67, 68] proposent l'utilisation de la planification HTN hors ligne afin de générer automatiquement des scripts. Tel qu'énoncé à la section 3.2.1, le problème avec les scripts est qu'ils deviennent rapidement longs et complexes lorsque des scénarios élaborés, comme ceux qu'offrent les jeux vidéo d'aujourd'hui, sont modélisés. Afin de générer automatiquement les scripts, un expert conçoit un ou plusieurs réseaux hiérarchiques de tâches spécifiques à un jeu donné et un ou plusieurs problèmes. Les problèmes sont représentés par des informations sur les personnages du jeu, tel que les tâches qu'ils peuvent accomplir ainsi que leurs états initiaux et leurs buts. Ensuite, un planificateur, dans ce cas JSHOP2 [56], une implémentation Java du planificateur SHOP2 [84], est utilisé afin de résoudre chaque problème. Un générateur de scripts traduit les solutions du planificateur (c'est-à-dire les plans) de son langage (par exemple PDDL) à un langage de scripts accepté par le jeu. Finalement, les nouveaux scripts générés sont ajoutés au jeu.

5.4 Planification utilisant des techniques d'apprentissage machine

La planification utilisant des techniques d'apprentissage machine est principalement utilisée sous forme de planification basée sur les cas dans les architectures de prise de décision. S. Ontanon [88] et A. Trusty, S. Ontanon et A. Ram [108] expliquent une utilisation de la planification basée sur les cas pour effectuer la prise de décision dans un RTS. L'approche utilisée par l'équipe est intéressante puisqu'elle comporte une phase hors ligne et une phase en temps réel.

La première phase se fait hors ligne et elle se nomme « Behavior acquisition ». Lors de cette phase, un plan initial qui guidera la recherche est créé par un expert. Pour ce faire, il joue au dit RTS lui-même et toutes ses actions sont enregistrées dans une trace. Par la suite,

pour chacune des actions de la trace, l'expert indique le but qu'il suivait en effectuant cette action. Grâce à ces annotations, une série de cas est générée. La deuxième phase se nomme « Execution ». Cette phase détermine d'abord quel plan formulé par l'expert, acquis lors de la phase précédente, correspond le plus à l'état courant du jeu. Ce processus utilise un algorithme du plus proche voisin standard avec une métrique de similarité basée sur les éléments but et comportement du triplet situation/but/comportement. Par la suite, le système tente de modifier le cas de l'expert récupéré pour trouver le meilleur plan valide dans l'état courant. A. Trusty, S. Ontanon et A. Ram [108] a amélioré cette dernière phase approche nommée Stochastic Plan Optimization.

La planification basée sur les cas est également utilisée par A. Sanchez-Ruiz [100] pour un jeu de stratégie de type « turn-based ». Dans ce cas, des connaissances ontologiques sont utilisées pour représenter l'environnement de jeu, ce qui est plus naturel pour les développeurs et permet d'accélérer le temps de planification. L'étude propose donc de maintenir une bibliothèque de stratégies (plans) connues dans le jeu. La nouveauté est que cette bibliothèque comprend des informations ontologiques sur les objets utilisés dans les stratégies passées ainsi que les conditions sous lesquelles ces stratégies ont été effectuées. La méthode pour déterminer les anciens plans pertinents est basée sur l'ontologie, contrairement aux algorithmes habituellement utilisés tels que l'algorithme du plus proche voisin.

Le système fonctionne comme suit. Premièrement, des nouvelles stratégies sont générées par un planificateur HTN, Repair-SHOP, un planificateur basé sur les cas basé sur le planificateur SHOP [85] qu'ils ont conçu. Pour ce faire, il demande à un module spécialisé le plan passé le plus pertinent se trouvant dans la base de connaissances pour atteindre les buts dans l'état courant. Par la suite, les anciens plans récupérés sont ordonnés en utilisant des fonctions de similarités numériques plus précises et le plan le plus similaire est envoyé au planificateur pour son adaptation. Finalement, un autre module communique avec le jeu afin de gérer l'exécution du plan et de déterminer quand un nouveau plan doit être calculé, adapté et stocké.

Le processus de récupération du plan le plus similaire utilisé dans l'étude est plus complexe que les techniques utilisées habituellement dans les systèmes de planification basés sur les cas. Par contre, les auteurs croient qu'en utilisant des techniques complexes pour récupérer les plans similaires, ces derniers seront meilleurs et plus faciles à adapter pour le planificateur, réduisant ainsi le temps de planification et améliorant la qualité du plan final.

CHAPITRE 6

Méthodologie

Dans ce chapitre, nous décrivons l'expérimentation qui a été réalisée afin de démontrer l'avantage des planificateurs sur l'approche classique des machines à états finis. Pour ce faire, il a fallu développer un moteur de jeu vidéo. Dans un premier temps, nous détaillons les métriques utilisées pour évaluer les planificateurs. Par la suite, nous décrivons nos implémentations de notre moteur de jeu vidéo et des planificateurs.

6.1 Détermination des métriques d'évaluation

Nous avons déterminé que les différentes architectures de prise de décisions devraient être évaluées selon trois aspects dans le cadre de ce travail :

1. Qualité de la planification
2. Qualité de l'agent
3. Jouabilité

6.1.1 Qualité de la planification

Les performances d'un planificateur sont généralement évaluées grâce à trois métriques : le temps CPU nécessaire pour déterminer une solution, le nombre de problèmes résolus dans un ensemble de problèmes de différents domaines et la qualité des plans produits [52, 53, 57, 58, 59, 60, 96]. Le nombre de problèmes résolus dans un ensemble de problèmes de différents domaines ne s'applique pas à notre projet, car le domaine visé est toujours le même. La qualité du plan est évaluée en calculant le nombre d'actions dans le plan ou le coût total du plan dans le cas où les actions ont un coût. Puisque, dans notre cas, le formalisme permettant de représenter le problème diffère quelque peu entre les deux planificateurs et que les coûts sont fonction des actions que l'on désire prioriser (tel que décrit dans la section 6.3.2), nous avons décidé de ne considérer que le nombre d'actions dans le plan. De plus, puisque, indépendamment du formalisme utilisé pour faire la planification, chaque plan doit être traduit en actions de plus bas niveau faisant partie du moteur de jeu, nous avons décidé d'ajouter le nombre de ces actions comme métrique. Nous appelons ces actions *actions atomiques*. Finalement, puisque notre environnement est dynamique, nous avons décidé

d'ajouter le nombre de planifications et le nombre de replanifications aux métriques afin de tester la robustesse des planificateurs. Il est important de mentionner que nous qualifions de replanification une planification qui est effectuée suite à l'**échec** du plan courant. Modifier le plan parce que le but de l'agent a changé n'est pas considéré comme une replanification, mais plutôt une planification supplémentaire. En résumé, les cinq métriques que nous utiliserons pour évaluer la qualité de la planification sont :

1. Le temps CPU moyen pour trouver une solution.
2. Le nombre moyen d'actions contenues dans les plans générés.
3. Le nombre moyen d'actions atomiques contenues dans les plans générés.
4. Le nombre moyen de planifications par partie.
5. Le nombre moyen de replanifications par partie.

Également, il est important de mentionner que, dans le cas du planificateur HTN, de par la nature du planificateur, le nombre de replanifications se divise en deux métriques : le nombre de replanifications totales et le nombre de replanifications partielles.

6.1.2 Qualité de l'agent

Bien que certains utilisent des métriques précises qui sont fonction du type de jeu dans lequel l'agent évolue (par exemple le nombre de collisions avec les murs ou les objets, la distance parcourue par l'agent, le nombre d'items amassés par l'agent, le nombre d'ennemis éliminés, etc.) [77, 76], la plupart des auteurs utilisent le nombre de victoires [44, 28, 25] et/ou le pointage [44, 108] (parfois calculé en fonction du nombre de victoires [46]) pour déterminer la qualité de leur agent. Puisque le but de notre jeu est de remporter le plus grand nombre de parties et que celui-ci comporte un pointage, nous avons décidé d'utiliser ces deux métriques pour évaluer la performance au niveau de la qualité de notre agent.

6.1.3 Jouabilité

Afin de calculer la jouabilité, G.N. Yannakakis et J. Hallam [121] proposent trois métriques générales s'appliquant aux jeux de type « prédateur/proie ». Puisque notre jeu s'apparente à cette catégorie, nous avons décidé de qualifier la jouabilité à l'aide de ces métriques. Par contre, puisque, dans notre jeu, la victoire est en fonction du nombre de points plutôt que cédée automatiquement à celui qui élimine son adversaire, nous avons ajouté deux métriques calculées à l'aide du pointage et dérivées de celles présentées par G.N. Yannakakis et J. Hallam [121]. Les métriques utilisées pour qualifier la jouabilité sont basées sur trois critères :

1. Niveau de difficulté approprié, c'est-à-dire lorsque le jeu n'est ni trop facile, ni trop difficile.

2. Diversité du comportement de l'agent, c'est-à-dire lorsqu'il y a diversité au niveau de comportement de l'agent entre chaque partie.
3. Diversité spatiale de l'agent, c'est-à-dire lorsque le comportement de l'agent est dynamique plutôt que statique. La diversité spatiale est caractérisée par un agent qui bouge continuellement dans le monde et le couvre uniformément. Ce comportement donne l'impression au joueur que l'agent possède un plan stratégique intelligent, ce qui ravive son intérêt envers le jeu.

Afin de calculer la valeur de ces trois critères, les métriques suivantes sont utilisées [121] :

1. Niveau de difficulté approprié :

$$T = \left[1 - \frac{E\{t_k\}}{t_{max}}\right]^{p_1} \tag{6.1}$$

où $E\{t_k\}$ est le temps moyen mis par l'agent pour éliminer son adversaire sur N parties, où N est le nombre total de parties jouées par l'agent. Dans notre cas, il s'agit du temps que met un joueur pour éliminer son adversaire lorsqu'il se trouve en mode attaque. Dans le cas de la machine à états finis, il s'agit du temps pour lequel l'agent se trouve dans la sous-machine *Combat* (voir section 6.3.2). Dans le cas de l'architecture GOAP, il s'agit du temps pour lequel l'agent possède *ENEMY_DEAD* comme but (voir section 6.3.2). Finalement, pour l'architecture HTN, il s'agit du temps pour lequel l'agent possède *Kill enemy* comme tête (voir section 6.3.2). t_{max} est le temps maximal mis par l'agent pour éliminer son adversaire en N parties et p_1 est un coefficient de pondération. En choisissant une valeur de p_1 inférieure à 1, T croît même lorsque la différence entre le temps moyen et le temps maximal mis par l'agent pour éliminer son adversaire est petite, faisant en sorte qu'un agent varié est préféré à un éliminateur presque optimal. Pour cette métrique, une valeur près de 1 est préférée. La valeur de p_1 utilisée par G.N. Yannakakis et J. Hallam [121] et que nous utiliserons est $p_1 = 0.5$.

Afin de déterminer si le niveau de difficulté est approprié, nous avons ajouté une métrique modifiant l'équation 6.1 pour que celle-ci soit plutôt basée sur le pointage. L'autre métrique que nous utiliserons donc pour ce point est :

$$T_2 = \left[1 - \frac{E\{pt\}}{pt_{max}}\right]^{p_2} \tag{6.2}$$

où pt est le pointage moyen de l'agent sur N parties, pt_{max} est le pointage maximal de l'agent en N parties et p_2 est un coefficient de pondération. Pour les mêmes raisons que la métrique précédente, nous avons choisi d'utiliser $p_2 = 0.5$.

2. Diversité du comportement de l'agent :

$$S = \left(\frac{\sigma}{\sigma_{max}}\right)^{p_3} \tag{6.3}$$

où

$$\sigma_{max} = \frac{1}{2}\sqrt{\frac{N}{N-1}}(t_{max} - t_{min})$$

et σ est l'écart type de t_k sur N parties, σ_{max} est un estimé de la valeur maximale de σ utilisé pour normaliser la métrique à une valeur entre 0 et 1, t_{max} est le temps maximal mis par l'agent pour éliminer son adversaire en N parties, t_{min} est le temps minimal mis par l'agent pour éliminer son adversaire en N parties et p_3 est un coefficient de pondération. La valeur de S augmente proportionnellement à l'écart type du temps mis par l'agent pour éliminer son adversaire en N parties. En utilisant l'équation 6.3 telle quelle, un agent qui prend des temps diversifiés pour éliminer son adversaire sera préféré, c'est pourquoi la valeur de p_3 utilisée par G.N. Yannakakis et J. Hallam [121] et que nous utiliserons est $p_3 = 1$.

Tout comme pour la première métrique, nous avons ajouté une métrique modifiant l'équation 6.3 pour que celle-ci soit plutôt basée sur le pointage. Ainsi, nous calculons la diversité du comportement de l'agent à l'aide de S et de la métrique suivante :

$$S_2 = \left(\frac{\phi}{\phi_{max}}\right)^{p_4} \tag{6.4}$$

où

$$\phi_{max} = \frac{1}{2}\sqrt{\frac{N}{N-1}}(pt_{max} - pt_{min})$$

et ϕ est l'écart type de pt sur N parties, ϕ_{max} est un estimé de la valeur maximale de ϕ utilisé pour normaliser la métrique à une valeur entre 0 et 1, pt_{max} est le pointage maximal de l'agent en N parties, pt_{min} est le pointage minimal de l'agent en N parties et p_4 est un coefficient de pondération. Pour les mêmes raisons que la métrique précédente, nous avons choisi d'utiliser $p_4 = 1$.

3. Diversité spatiale de l'agent :

$$H_n = \left[-\frac{1}{\log V_n}\sum_i \frac{v_{in}}{V_n}\log\left(\frac{v_{in}}{V_n}\right)\right]^{p_5} \tag{6.5}$$

où V_n est le nombre de visites pour toutes les cellules visitées pour une partie, c'est-à-dire $V_n = \sum_i v_{in}$ où v_{in} est le nombre de fois que la cellule i a été visitée dans la partie n,

et p_5 est un coefficient de pondération. H_n calcule l'entropie normalisée entre $[0, 1]$ des cellules visitées par l'agent dans une partie, ce qui qualifie la complétude et l'uniformité avec laquelle l'agent couvre le monde. En utilisant des valeurs de p_5 supérieures à 1, la distinction entre de faibles et de fortes valeurs d'entropie est accentuée. La valeur de p_5 utilisée par G.N. Yannakakis et J. Hallam [121] et que nous utiliserons donc est $p_5 = 4$.

Étant donné les valeurs d'entropie normalisée H_n pour N parties, la valeur d'intérêt pour calculer la diversité spatiale de l'agent est la moyenne des entropies normalisées $E\{H_n\}$ pour N parties

Les trois métriques peuvent être combinées linéairement afin de créer une seule métrique d'intérêt présentée à l'équation 6.6.

$$I = \frac{\alpha T + \beta T_2 + \gamma S + \delta S_2 + \epsilon E\{H_n\}}{\alpha + \beta + \gamma + \delta + \epsilon} \tag{6.6}$$

Les valeurs de α, β, γ, δ et ϵ dépendent du style d'agent recherché en fonction de l'intérêt porté à chaque métrique. Tout d'abord, nous jugeons que les métriques basées sur le pointage sont plus importantes que les métriques basées sur le temps mis par l'agent pour éliminer son adversaire. Par la suite, nous avons décidé qu'un comportement diversifié de l'agent est le plus important, suivi de la diversité spatiale de l'agent puisqu'il doit souvent se déplacer pour récolter des objets et finalement d'un niveau de difficulté approprié. Ainsi, nous avons choisi de poser $\alpha = 1$, $\beta = 3$, $\gamma = 2$, $\delta = 5$ et $\epsilon = 4$.

6.1.4 Résumé des métriques quantitatives

Le tableau 6.1 présente un sommaire des métriques quantitatives.

Tableau 6.1 Sommaire des métriques quantitatives.

Métrique	**Catégorie**
Le temps CPU moyen pour trouver une solution	Qualité de la planification
Le nombre moyen d'actions contenues dans les plans générés	Qualité de la planification
Le nombre moyen d'actions atomiques contenues dans les plans générés	Qualité de la planification
Le nombre moyen de planifications par partie	Qualité de la planification
Le nombre moyen de replanifications par partie	Qualité de la planification
Le nombre de victoires	Qualité de l'agent
Le pointage	Qualité de l'agent
T, le niveau de difficulté approprié en fonction de t_k	Jouabilité
T_2, le niveau de difficulté approprié en fonction du pointage	Jouabilité
S, la diversité du comportement de l'agent en fonction de t_k	Jouabilité
S_2, la diversité du comportement de l'agent en fonction du pointage	Jouabilité
H_n, la diversité spatiale de l'agent	Jouabilité
I, la jouabilité, c'est-à-dire une combinaison linéaire des cinq métriques précédentes	Jouabilité

6.1.5 Métriques qualitatives

Finalement, bien que nous n'ayons pas défini de métriques qualitatives comme telles, nous jugions important de comparer les différentes architectures sur leurs aspects qualitatifs. Grâce à nos observations et nos expériences personnelles d'implémentation, nous avons été en mesure de comparer les différentes architectures par rapport aux quatre métriques suivantes :

1. Les limitations comportementales des agents
2. La facilité d'implémentation
3. La robustesse face aux changements de conception

Limitations comportementales des agents

Chacune des architectures possède des avantages les unes par rapport aux autres, mais également des inconvénients. Ces inconvénients peuvent entraîner des limitations comportementales, c'est-à-dire l'impossibilité ou la difficulté à atteindre le comportement désiré d'un agent avec une certaine architecture alors qu'il est aisé de le faire avec une autre.

Facilité d'implémentation

Nous qualifions la facilité d'implémentation en termes d'efforts mis pour concevoir et implémenter une architecture de prise de décision. Ces efforts peuvent se mesurer en temps passé à la conception et l'implémentation de l'architecture, la complexité des tâches de conception et d'implémentation et les difficultés à surmonter lors de l'élaboration de ces tâches.

Robustesse face aux changements de conception

Au long de son cycle de développement, notre projet a subi plusieurs changements de conception. La robustesse d'une architecture de prise de décision face à ces changements se qualifie par le nombre de modifications qu'il est nécessaire d'y apporter afin qu'elle réponde aux nouvelles exigences de conception.

6.2 Choix des planificateurs étudiés

Il existe une multitude de planificateurs répondant aux exigences des jeux vidéo. Pour ce projet, nous avions décidé d'implémenter deux architectures de prise de décision utilisant un planificateur : un choix s'imposait donc.

La plupart des planificateurs récents ont participé à une compétition nommée « International Planning Competition » [57, 58, 59, 60]. Cette compétition, qui a lieu tous les deux ans depuis 2002, évalue la performance de plusieurs planificateurs en leur fournissant différents types de problèmes à résoudre. Grâce aux résultats et aux articles présentant les planificateurs proposés à chaque compétition, nous avons pu répertorier une foule de planificateurs applicables au domaine du jeu vidéo, c'est-à-dire utilisant peu de ressources CPU et de mémoire, ne retournant pas nécessairement la solution optimale, mais plutôt une solution valide le plus rapidement possible, etc. Cependant, la revue de littérature de la planification appliquée au domaine des jeux vidéo démontre que trois types de planificateurs seulement sont majoritairement utilisés dans les architectures de prise de décision. Il s'agit de planificateurs GOAP, de planificateurs HTN et de planificateurs basés sur les cas. Puisque la planification basée

sur les cas exige beaucoup plus de temps au niveau de l'implémentation (tel que démontré par A.Sanchez-Ruiz [100]), notre choix final s'est donc porté sur une architecture GOAP et une architecture HTN.

6.3 Implémentation

6.3.1 Développement d'une plateforme de test

La première étape de la phase de développement consistait à determiner le moteur de jeu vidéo que nous allions utiliser. Une multitude de moteurs de jeu vidéo à code source ouvert est actuellement disponible. Nous avons analysé plusieurs moteurs afin de déterminer celui qui serait utilisé lors de la phase de développement. Les critères de sélection furent la facilité à modifier l'intelligence artificielle des personnages, la richesse de l'environnement, la ressemblance du moteur avec le type de jeu visé (jeu d'action), les ressources disponibles (documentation, niveaux déjà créés, etc.) et la portabilité. Suite à une revue des travaux antérieurs portant sur l'intelligence artificielle dans les jeux vidéo, les moteurs retenus pour évaluation furent F.E.A.R. SDK [35], Quake III [54], OpenNERO [42], Source SDK [109], Unreal Tournament [36], Wargus [106], No One Lives Forever 2 [105]. Suite à l'analyse des différents moteurs, il a été décidé que le moteur de Quake III était celui qui était le plus approprié à notre projet. Par contre, après une exploration approfondie du moteur, nous nous sommes rapidement aperçu qu'il était trop limité pour pouvoir générer des plans intéressants. Puisque la compréhension et la modification d'un moteur de jeu vidéo exigent beaucoup de temps, nous avons décidé de créer notre propre moteur afin de gagner en flexibilité.

La plateforme de test est codée en Java aux fins de portabilité. Le jeu est une variante d'un jeu de tir, présenté à la section 3.1.1, en deux dimensions. Afin de simplifier la conception des différentes architectures de prise de décision, notre jeu, contrairement à la plupart des jeux de tir, s'effectue en un contre un. Le but des joueurs est de faire le plus de points possible. Pour ce faire, le joueur possède deux choix : amasser des biens ou éliminer son adversaire. Chaque bien possède un pointage qui lui est associé et éliminer l'adversaire donne un certain nombre de points fixe. Lorsque le joueur s'élimine accidentellement, un certain nombre de points (élevé) lui sont déduits. Il existe donc deux façons de terminer une partie : tous les objets ont été amassés ou un des deux joueurs meurt. Il est important de noter qu'éliminer l'adversaire ne mène pas toujours à la victoire. Si un joueur élimine son adversaire, mais que le nombre de points associé à cette action ne lui permet pas d'avoir plus de points que ledit adversaire, alors la partie se termine et le joueur perd. Si un joueur perçoit son ennemi et sait qu'il ne gagnera pas la partie en le tuant, il tentera alors de le ralentir pour tenter

d'amener son pointage au même niveau. C'est pourquoi il est possible de généraliser les buts des joueurs en quatre catégories : explorer pour amasser des objets, attaquer, se défendre et ralentir l'ennemi. La figure 6.1 présente une capture d'écran du jeu.

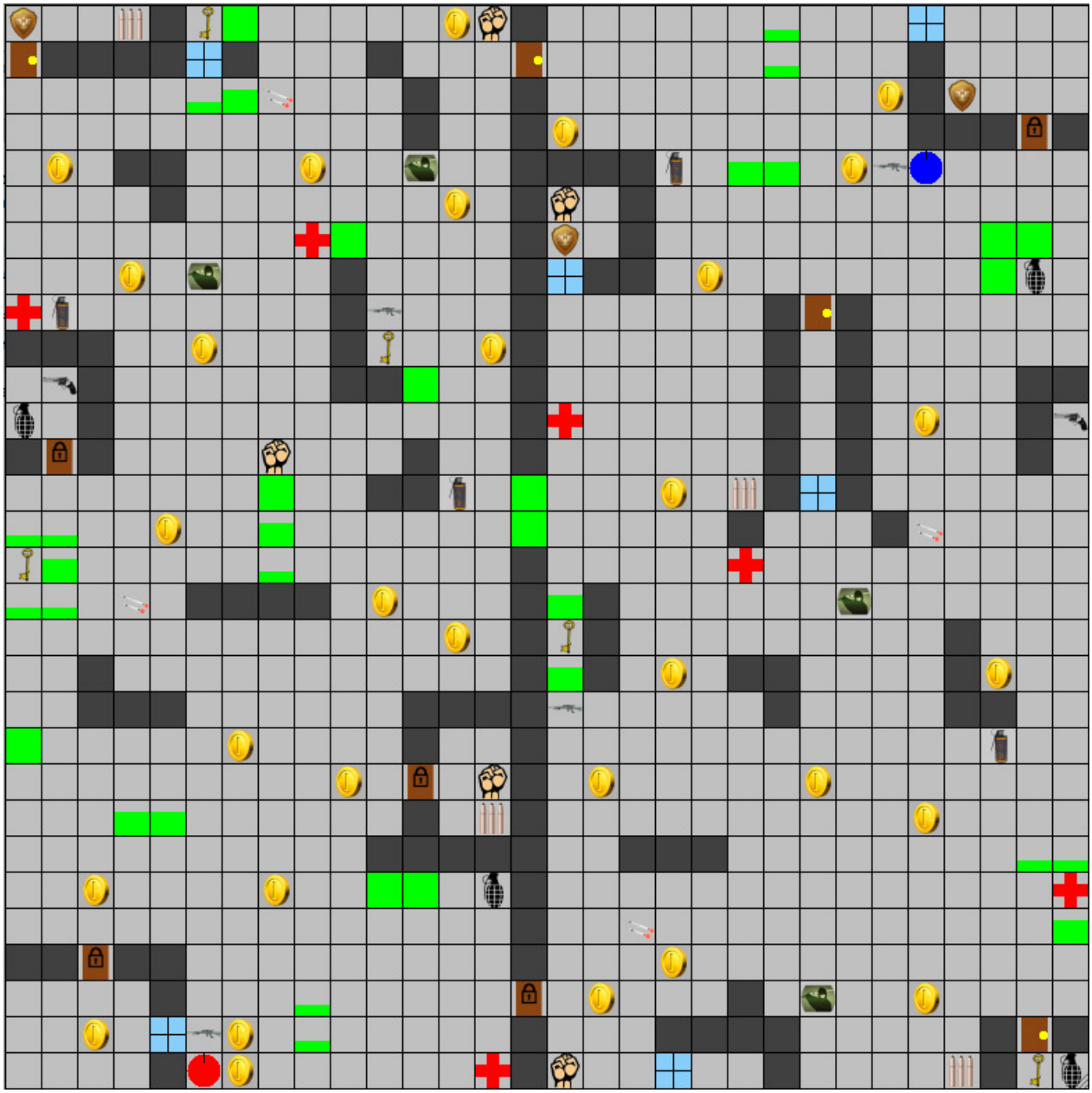


Figure 6.1 Capture d'écran de notre jeu.

Le monde

Le monde dans lequel évoluent les agents est une grille composée de cases. Chaque case peut contenir ou non une seule entité. Le tableau 6.2 présente toutes les entités du jeu.

Tableau 6.2 Entités du monde

Entité	**Description**
Joueur	Chaque joueur possède un certain nombre de points de vie, de la force et des points d'armure. La force permet de déplacer des obstacles et les points d'armure permettent de réduire les dommages. Un joueur possède également un certain nombre d'équipements et d'armes. Bien que le joueur puisse posséder plusieurs armes, il ne peut avoir qu'une seule arme active à la fois.
Mur	Les murs bloquent les tirs des joueurs et ne peuvent être détruits qu'avec une grenade.
Porte	Les portes peuvent être ouvertes à tout moment par le joueur. Elles peuvent également être brisées avec une grenade.
Porte barrée	Les portes barrées peuvent être ouvertes si le joueur possède une clé. Elles peuvent également être brisées avec une grenade.
Fenêtre	Les fenêtres peuvent être brisées à tout moment par le joueur. Elles peuvent également être brisées avec une grenade ou un tir.
Obstacles	Les obstacles peuvent être déplacés par le joueur s'il possède suffisamment de force. Un seul obstacle peut être déplacé à la fois. Il existe trois types d'obstacles : bas, moyen et haut. Plus l'obstacle est grand, plus il requiert de force pour être déplacé. Les obstacles ne peuvent pas être détruits avec une grenade.
Clé	Les clés permettent d'ouvrir les portes barrées. N'importe quelle clé permet d'ouvrir n'importe quelle porte, mais ne peut être utilisée qu'une seule fois.
Jeton	Les jetons peuvent être amassés par le joueur pour augmenter son pointage. Ils n'ont aucune autre utilité qu'augmenter le pointage, mais sont les objets qui donnent le plus de points.
Bonus vie	Les bonus de vie permettent à un joueur de regagner des points de vie perdus.
Bonus armure	Les bonus d'armure permettent d'augmenter l'armure du joueur jusqu'à un certain maximum.
Bonus force	Les bonus de force permettent d'augmenter de façon permanente la force du joueur.

Bonus ninja	Les bonus ninja permettent à un joueur de devenir silencieux pendant un instant choisi pour éviter de se faire repérer par un ennemi.
Grenade	Les grenades permettent de détruire des objets et d'infliger des dommages aux joueurs.
Grenade assourdissante	Les grenades assourdissantes permettent d'aveugler et d'assourdir temporairement l'ennemi.
Couteau	Chaque joueur débute la partie avec un couteau. Il s'agit de l'arme infligeant le plus de dommage. Par contre, le joueur doit être à proximité de son ennemi pour l'éliminer au corps-à-corps.
Magnum	Le magnum permet d'infliger des dommages de loin. Il possède un nombre total de munitions et un chargeur. Lorsqu'il n'y a plus de balles dans son chargeur, le magnum doit être rechargé.
Munition	Les munitions permettent de recharger le magnum.
Fusil à darts	Le fusil à darts permet de geler temporairement l'ennemi, lui empêchant ainsi tout mouvement. Tout comme le magnum, le fusil à darts possède un nombre total de munitions et un chargeur. Lorsqu'il n'y a plus de balles dans son chargeur, celui-ci doit être rechargé.
Dart	Les darts permettent de recharger le fusil à darts.

Les actions possibles

Tous les joueurs possèdent un certain nombre d'actions possibles. Ce sont les actions que nous avons nommées actions atomiques à la section 6.1.1. Ces actions sont les suivantes :

- Aller à la case voisine du haut
- Aller à la case voisine du bas
- Aller à la case voisine de droite
- Aller à la case voisine de gauche
- Se tourner vers le haut
- Se tourner vers le bas
- Se tourner vers la droite
- Se tourner vers la gauche
- Briser une fenêtre
- Ouvrir une porte
- Prendre un objet ou un obstacle

- Déposer un obstacle
- Lancer une grenade
- Lancer une grenade assourdissante
- Tirer
- Recharger
- Poignarder
- Changer d'arme pour le couteau
- Changer d'arme pour le magnum
- Changer d'arme pour le fusil à darts
- Utiliser le bonus ninja

Le modèle de perception

Chaque joueur possède une mémoire de ce qu'il a perçu. La mémoire est une grille tout comme le monde et elle est vide au départ. À chaque fois que le joueur prend une action, elle est mise à jour. Pour mettre à jour sa mémoire, le joueur possède deux façons pour percevoir son environnement : sa vue et son ouïe. Sa vue est un champ de vision triangulaire d'une profondeur définie. Le joueur voit en tout temps tout ce qui se trouve dans ce triangle, c'est-à-dire que sa vue ne peut être obstruée par aucun objet. Nous avons choisi cette approche pour que nos joueurs évitent de passer trop de temps à explorer. En second lieu, l'ouïe du joueur lui permet de détecter un ennemi en mouvement et les grenades qui tombent près de lui. Toutes les décisions prises par le joueur sont basées seulement sur ce qui se trouve dans sa mémoire. Jusqu'à preuve du contraire, tout ce qui s'y trouve existe encore. Par exemple, si un objet se trouvant dans sa mémoire a été pris par son ennemi, celui-ci ne disparaîtra de sa mémoire que lorsqu'il verra que la case où il se trouvait est maintenant vide.

6.3.2 Implémentation des architectures de prise de décision

Une fois le moteur de jeu complété, il fallait implémenter les architectures de prise de décision afin de pouvoir effectuer notre évaluation. Tel que mentionné précédemment, les trois architectures de prise de décision implémentées sont les suivantes : une machine à états finis servant de base de comparaison, une architecture GOAP et une architecture HTN.

Machine à états finis

La machine à états finis qui a été développée pour notre jeu est une machine à états finis hiérarchique telle que décrite dans la section 3.2.2. La figure 6.2 présente les sous-machines, les états ainsi que les transitions possibles entre les états.

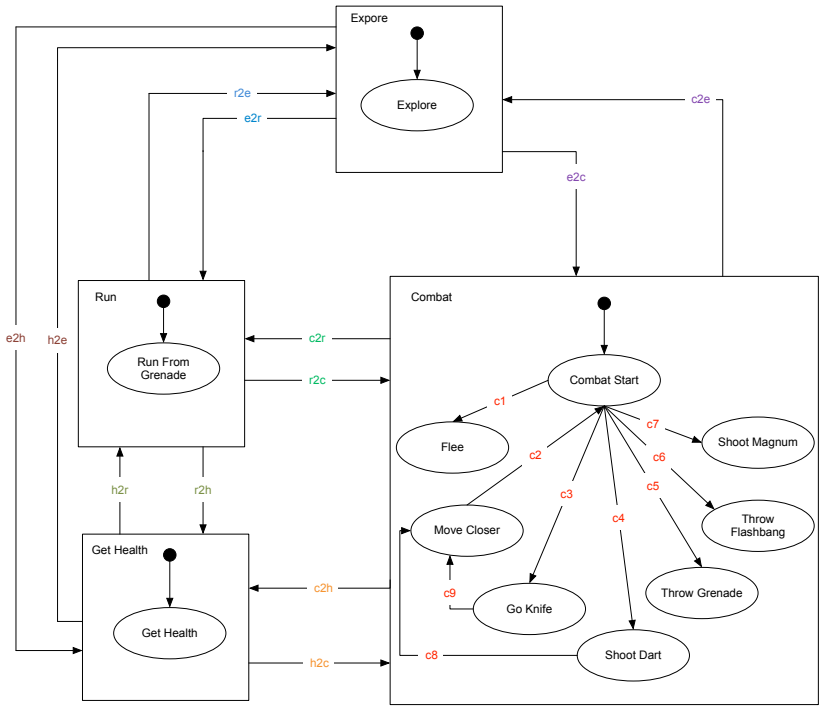


Figure 6.2 Machine à états hiérarchique développée pour notre jeu.

Les conditions de transition sont présentées à l'annexe A. Afin d'assurer la qualité de notre machine à états, celle-ci a subi un processus itératif durant lequel quatre testeurs observaient des parties jouées par des agents utilisant notre FSM. Ceux-ci notaient les comportements qu'ils jugeaient inadéquats ou à améliorer et nous modifiions notre machine à états en fonction de leurs commentaires. Chaque testeur a observé environ une centaine de parties. Une fois ce processus d'affinage terminé, nous avons observé le comportement des agents utilisant la FSM pour environ 50 parties afin de nous assurer que leur comportement était tel que désiré. Finalement, un agent utilisant la FSM possèdent un taux de 88.6% de victoires contre un agent complètement aléatoire.

Architecture GOAP

Comme première architecture de prise de décision utilisant un planificateur, nous avons implémenté une architecture GOAP en utilisant une exploration A* grandement basée sur l'architecture proposée par J. Orkin [90, 91, 92]. Son pseudocode est présenté à la figure 6.3. Pour représenter le monde du jeu ainsi que les actions, nous avons aussi utilisé le formalisme STRIPS. Nous avons également utilisé la même fonction heuristique pour guider la recherche, soit le nombre de symboles différents entre l'état courant et le but. Par contre, il est important de mentionner que pour les symboles possédant une valeur de type entier, la valeur à ajouter

à l'heuristique est la valeur absolue de la différence entre les valeurs des symboles et non pas seulement 1 lorsque ceux-ci sont différents. Par exemple, soit l'état désiré e_d et l'état courant e_c suivants :

$$e_d = AT(5,2) \wedge STRENGTH(3) \wedge HAS\_MAGNUM \wedge HAS\_MAGNUM\_AMMO$$

$$e_c = AT(4,1) \wedge STRENGTH(1) \wedge HAS\_MAGNUM \wedge HAS\_GRENADE$$

Afin de déterminer la valeur de l'heuristique, il faut comparer tous les symboles de l'état désiré avec ceux de l'état courant. Pour le symbole *AT*, puisque son type n'est pas un entier et que les valeurs diffèrent pour les deux états, nous devons ajouter 1 à l'heuristique. Pour le symbole *STRENGTH*, le type est un entier. Nous devons donc ajouter la valeur absolue de la différence entre les valeurs des symboles, soit $abs(3-1) = 2$. Le symbole *HAS_MAGNUM* n'a pas de valeur et est également présent dans l'état courant, nous n'ajoutons donc rien à la valeur de l'heuristique. Finalement, le symbole *HAS_MAGNUM_AMMO* n'est pas présent dans l'état courant, nous devons donc ajouter 1 à la valeur de l'heuristique pour une valeur finale de $h = 1+2+1 = 4$. Également, il est important de mentionner que, puisque chaque action peut ajouter/modifier plus d'un symbole, notre heuristique n'est pas admissible puisqu'elle peut surestimer le coût pour atteindre l'état désiré. Par exemple, bien que le coût pour l'action « Flash enemy » soit de 1, celle-ci ajoute deux symboles, soient *ENEMY_FLASHED* et *SLOW_ENEMY*. La solution retournée n'est donc pas toujours la solution optimale, mais nous jugeons que ce comportement est adéquat puisqu'il permet d'augmenter la diversité de notre agent et lui permet de prendre une décision rapidement.

Le pseudo-code de notre algorithme de recherche de plan est présenté à la figure 6.3.

```
fonction TROUVER_PLAN(etat, but) retourne un plan
    entrées : etat, l'état courant
              but, le symbole représentant le but à atteindre
    variables : ouverts, la liste des noeuds ouverts classée de sorte que le premier
                noeud soit toujours celui avec la valeur d'heuristique la plus faible
                fermes, la liste des noeuds fermés
                actions, la liste des actions possibles dans l'état courant
                buts, la liste des symboles qui doivent se trouver dans l'état désiré
                noeud_courant, le noeud courant
    buts ← but
    noeud_courant ← noeud avec buts = buts, etat = etat, noeud parent = null,
                    action = null, coût = 0, heuristique = 1
    ouverts ← ajouter noeud_courant à ouverts
    tant que ouverts n'est pas vide
        noeud_parent ← premier noeud de ouverts
        si la valeur de l'heuristique de noeud_parent = 0 alors retourner le plan
        fermes ← ajouter noeud_parent à fermes
        etat ← etat de noeud_parent
        buts ← buts de noeud_parent
        but_courant ← premier but de buts
        tant que etat contient but_courant alors
            buts ← enlever but_courant de buts
            but_courant ← premier but de buts
        actions ← les actions qui permettent de réaliser but_courant
        tant que actions n'est pas vide
            action ← première action de actions
            etat ← ajouter les effets de action à etat
            buts ← ajouter les préconditions symboliques de action à buts
            pour chaque but dans buts
                si but se trouve dans etat
                    buts ← retirer but de buts
            h ← taille de buts
            noeud_courant ← noeud avec buts = buts, etat = etat,
                            noeud parent = noeud_parent, action = action,
                            coût = coût de action, heuristique = h
            si fermes ne contient pas noeud_courant alors
                ouverts ← ajouter noeud_courant à ouverts
```

Figure 6.3 Pseudocode de notre algorithme A* utilisé pour notre architecture GOAP.

Tel qu'expliqué par J. Orkin [91], les actions plus spécifiques de notre architecture ont un coût moins élevé que les actions plus générales pour guider A* afin de prioriser les actions spécifiques. Par exemple, supposons un état où les préconditions des actions « Shoot magnum » et « Shoot magnum flashed » sont vérifiées, c'est-à-dire que les deux actions sont possibles et sont alors considérées par A*. Rappelons que dans l'algorithme de A*, chaque noeud n est évalué grâce à une fonction $f(n)$ qui est la somme du coût de l'état initial au noeud n, noté $g(n)$, et de la valeur de l'heuristique du noeud n, notée $h(n)$, soit $f(n) = g(n) + h(n)$. Soit $f(m) = g(m) + h(m)$ la fonction d'évaluation du noeud associé à l'action « Shoot magnum » et $f(f) = g(f) + h(f)$ la fonction d'évaluation du noeud associé à l'action « Shoot magnum flashed ». Puisque les deux actions ont le même effet, la valeur de l'heuristique sera la même après l'exécution des actions, c'est-à-dire que $h(m) = h(f)$. De ce fait, la différence entre $f(m)$ et $f(f)$ ne repose que sur le coût des actions. Puisque $g(m) = 6$ et $g(f) = 2$, $f(m) > f(f)$. Ainsi, A* priorisera l'action « Shoot magnum flashed », plus spécifique, donc l'agent profitera du fait qu'il possède une grenade assourdissante pour aveugler l'ennemi avant de l'attaquer.

Chaque recherche débute avec un état but qui est constitué d'un seul symbole. Ce symbole représente le but courant actif du joueur. Un joueur ne peut avoir qu'un seul but actif à la fois. Le planificateur utilise également la mémoire du joueur pour déterminer le plan. En effet, ceci est nécessaire pour connaître par exemple la position des objets qui l'entourent, les propriétés de son ennemi, etc. Ces informations ne sont pas contenues dans la représentation symbolique et doivent être connues par le planificateur. Celui-ci utilise ces informations pour vérifier les préconditions des actions. Les effets des actions sont également appliqués à la mémoire du joueur pendant la recherche. Le planificateur utilise donc deux représentations pour effectuer sa recherche : une représentation symbolique et une représentation plus bas niveau, soit la mémoire du joueur. Une fois le plan formulé, le joueur tentera d'exécuter les actions de celui-ci en ordre. Pour ce faire, lorsque vient le moment d'exécuter une action, celle-ci est traduite en un « sous-plan » d'actions atomiques présentées à la section 6.3.1. Toutes les actions atomiques sont alors exécutées par le joueur jusqu'à ce que le sous-plan soit complété. À ce point, l'action est considérée comme accomplie et le joueur tentera alors d'exécuter la prochaine action dans le plan.

Les symboles que nous avons utilisés pour représenter un état, toutes les actions de notre architecture ainsi que le pseudocode démontrant comment le but initial de l'agent est déterminé sont présentés à l'annexe B.

Afin de mieux comprendre le fonctionnement de notre architecture GOAP, analysons le fonctionnement de notre architecture lorsque l'agent se trouve dans la situation présentée à la figure 6.4.

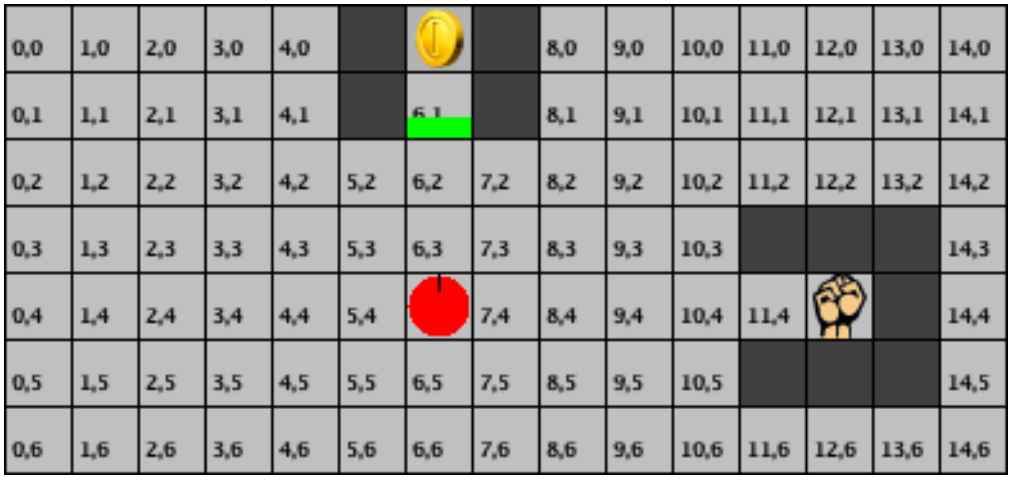
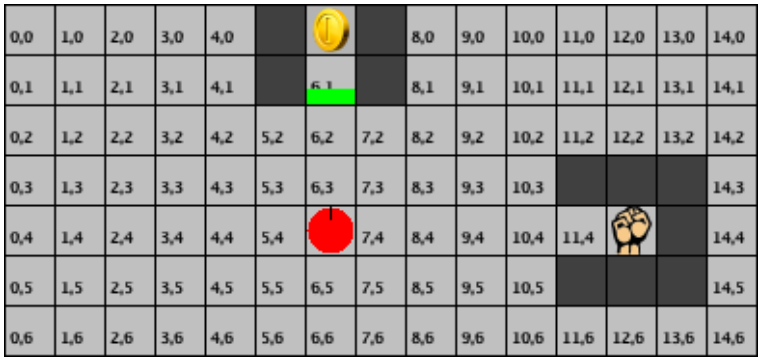


Figure 6.4 Exemple de cas qui doit être résolu par notre architecture GOAP.

L'agent se trouve à la position (6,4) et est en mode exploration, c'est-à-dire que son but courant est *INCREASE_SCORE*. La seule action répondant à ce but est l'action *GET_EVERY_ITEMS*. Cette action est différente de toutes les autres actions du jeu et a pour effet d'ajouter un noeud avec le but *NO_ITEM_AT* pour chacun des quatre objets les plus près de l'agent dans le premier niveau de l'arbre. Ceci a pour but de faire en sorte que l'agent ne considère pas seulement l'objet le plus près de lui lorsqu'il se trouve en mode exploration, dans l'éventualité ou celui-ci ne serait pas atteignable. Démarrons donc la recherche avec le noeud tentant de récupérer le jeton se trouvant à la position (6,0). Les informations du noeud sont les suivantes :

État symbolique courant : *STRENGTH(0)*
Mémoire courante :



But(s) à réaliser :

- *STRENGTH(0)*
- *NO_ITEM_AT(6,0)*

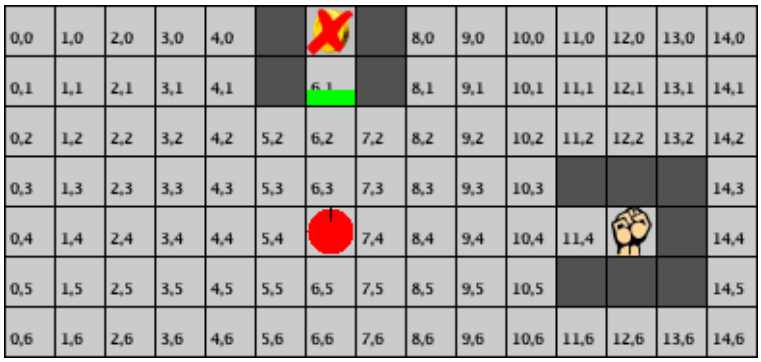
Action(s) considérée(s) : *PICK(6,0)*

L'action *PICK* possède *AT* comme précondition. Celle-ci est donc ajoutée aux buts à réaliser. La seule action permettant de réaliser ce but est l'action *GO_TO*. Les informations du prochain noeud seront donc :

État symbolique courant :

- *STRENGTH(0)*
- *NO_ITEM_AT(6,0)*

Mémoire courante :



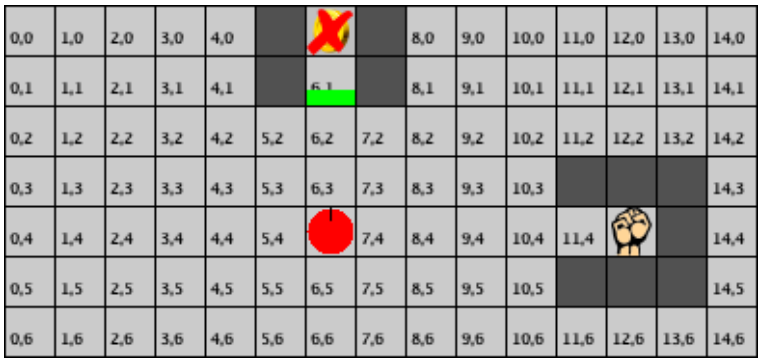
But(s) à réaliser : *AT(6,0)*
Action(s) considérée(s) : *GO_TO(6,0)*

L'action *GO_TO* appelle le planificateur de trajectoire sur la mémoire pour déterminer ses préconditions. Puisqu'il existe un obstacle bas dans la trajectoire de l'agent vers le jeton, la précondition *NO_LOW_OBSTACLE_AT* sera ajoutée comme but à réaliser. La seule action permettant de réaliser ce but est l'action *MOVE_LOW_OBSTACLE*. Les informations du prochain noeud seront :

État symbolique courant :

- *STRENGTH(0)*
- *NO_ITEM_AT(6,0)*

Mémoire courante :



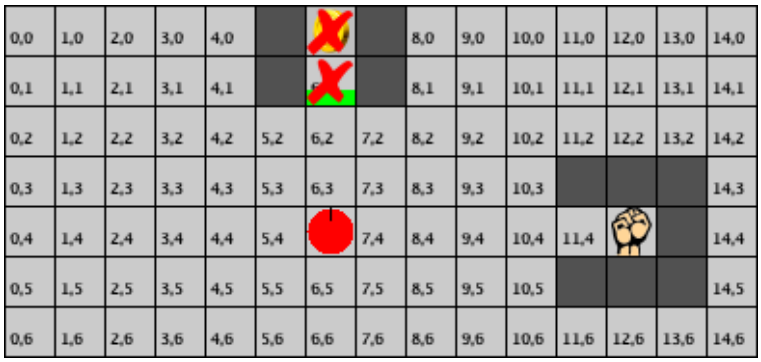
But(s) à réaliser : *NO_LOW_OBSTACLE_AT(6,1)*
Action(s) considérée(s) : *MOVE_LOW_OBSTACLE(6,1)*

L'action *NO_LOW_OBSTACLE_AT* exige que l'agent se trouve à la position de l'obstacle et possède 1 point de force. Les préconditions *AT* et *STRENGTH(1)* seront donc ajoutée aux buts à réaliser dans cet ordre, puisque l'agent doit avoir la force nécessaire avant d'aller déplacer l'obstacle. Le but à réaliser est donc *AT* et la seule action permettant de réaliser ce but est l'action *GO_TO*. Les informations du prochain noeud seront :

État symbolique courant :

- *STRENGTH(0)*
- *NOITEMAT(6,0)*
- *NO_LOW_OBSTACLE_AT(6,1)*

Mémoire courante :



But(s) à réaliser :

- *AT(6,1)*
- *STRENGTH(1)*

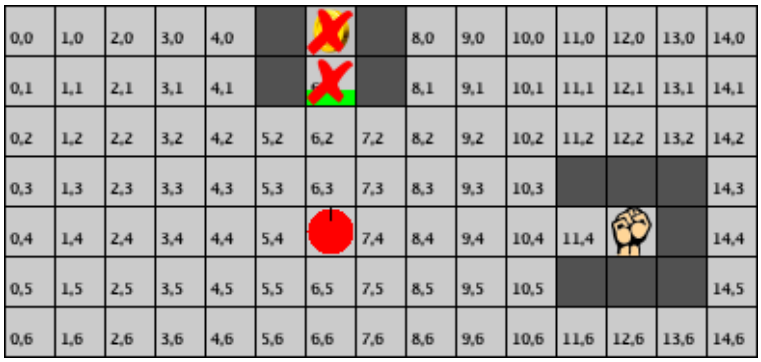
Action(s) considérée(s) : *GO_TO(6,1)*

Tel que mentionné plus haut, l'action *GO_TO* appelle le planificateur de trajectoire sur la mémoire pour déterminer ses préconditions. Puisque la trajectoire entre la position de l'agent (6,4) et l'obstacle (6,1) est directe, l'action *GO_TO* ne possède aucune précondition. Rien n'est ajouté à la liste de buts à réaliser. Le nouveau but à réaliser est donc *STRENGTH(1)*. Puisque notre agent ne possède aucun point de force (ce qui est traduit par *STRENGTH(0)* dans l'état symbolique courant), celui-ci doit aller récupérer un bonus de force. La seule action permettant de réaliser ce but est l'action *GET_STRENGTH*. Les informations du prochain noeud seront :

État symbolique courant :

- *STRENGTH(0)*
- *NO_ITEM_AT(6,0)*
- *NO_LOW_OBSTACLE_AT(6,1)*

Mémoire courante :



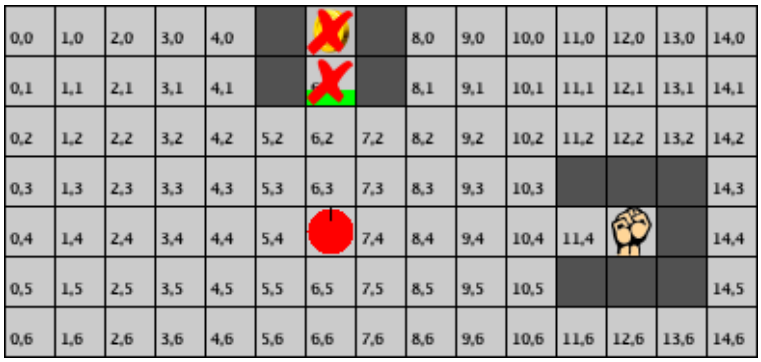
But(s) à réaliser : *STRENGTH(1)*
Action(s) considérée(s) : *GET_STRENGTH(12,4)*

L'action *GET_STRENGTH* possède *NO_ITEM_AT* comme précondition. Celle-ci est donc ajoutée aux buts à réaliser. La seule action permettant de réaliser ce but est l'action *PICK*. Les informations du prochain noeud seront donc :

État symbolique courant :

- *STRENGTH(1)*
- *NO_ITEM_AT(6,0)*
- *NO_LOW_OBSTACLE_AT(6,1)*

Mémoire courante :



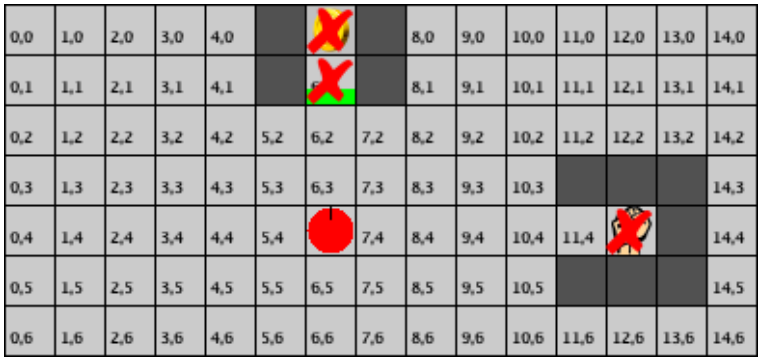
But(s) à réaliser : *NO_ITEM_AT(12,4)*
Action(s) considérée(s) : *PICK(12,4)*

Tel que mentionné plus haut, l'action *PICK* possède *AT* comme précondition. Celle-ci est donc ajoutée aux buts à réaliser. La seule action permettant de réaliser ce but est l'action *GO_TO*. Les informations du prochain noeud seront donc :

État symbolique courant :

- *STRENGTH(1)*
- *NO_ITEM_AT(6,0)*
- *NO_LOW_OBSTACLE_AT(6,1)*
- *NO_ITEM_AT(12,4)*

Mémoire courante :



But(s) à réaliser : *AT(12,4)*
Action(s) considérée(s) : *GO_TO(12,4)*

Tel que mentionné plus haut, l'action *GO_TO* appelle le planificateur de trajectoire sur la mémoire pour déterminer ses préconditions. Puisque la trajectoire entre la position de l'agent (6,4) et le bonus de force (12,4) est directe, l'action *GO_TO* ne possède aucune précondition. Rien n'est ajouté à la liste de buts à réaliser et comme celle-ci est vide, le plan est terminé. En remontant l'arbre, nous réalisons que le plan produit est : *GO_TO(12,4)* → *PICK(12,4)* → *GET_STRENGTH(12,4)* → *GO_TO(6,1)* → *MOVE_LOW_OBSTACLE(6,1)* → *GO_TO(6,0)* → *PICK(6,0)*.

Architecture HTN

Le planificateur HTN que nous avons implémenté est simple et est basé sur l'algorithme simplifié de SHOP2 [84]. Une différence notable de notre algorithme est que nos sous-tâches peuvent également être alternatives. Dans ce cas, le planificateur tentera d'accomplir chacune des sous-tâches dans des plans différents. D'autres simplifications ont également été faites. Le pseudocode de notre planificateur HTN est présenté ci-dessous.

```
fonction FINDPLAN(state, tasklist, plan)
        entrées : state, état courant du monde
                  tasklist, la liste de tâches à compléter
                  plan, le plan généré
        variable globale : plans, liste des plans

        si tasklist est vide
          plans ← ajouter plan à plans
          retourner
        T0 ← première tâche de tasklist
        tasklist ← enlever T0 de tasklist
        si les préconditions de T0 sont vérifiées avec state
              plan ← ajouter T0 à plan
              si T0 est une tâche primitive
                    state ← appliquer les effets de T0 sur state
                    FINDPLAN(state, tasklist, plan)
              sinon
                    si les sous-tâches de T0 sont de type alternatives
                          pour chaque sous-tâche S0 de T0
                                tasklist ← ajouter S0 à tasklist
                                FINDPLAN(state, tasklist, plan)
                    si les sous-tâches de T0 sont de type ordonnées
                          S ← liste de sous-tâches ordonnées de T0
                          tasklist ← ajouter S à tasklist
                          FINDPLAN(state, tasklist, plan)
                    si les sous-tâches de T0 sont de type non-ordonnées
                          S ← liste de sous-tâches ordonnées de T0
                          mélanger aléatoirement S
                          tasklist ← ajouter S à tasklist
                          FINDPLAN(state, tasklist, plan)
```

Figure 6.5 Pseudocode de notre planificateur HTN.

Le pseudocode présenté à la figure 6.5 remplit une liste avec tous les plans possibles. Une fois cette liste complétée, celle-ci est triée en fonction du coût total des plans et celui présentant le coût le moins élevé est choisi. Il est à noter que le coût d'un plan est la somme des coûts de toutes ses tâches, y compris les tâches composées. Cette méthode garantit que le plan retourné est toujours le moins coûteux, mais peut parfois exiger beaucoup de temps de calcul. Afin d'améliorer les performances du planificateur HTN, il aurait été possible de maintenir une liste de tous les plans alternatifs générés triés selon leur coût partiel, par exemple. Une autre amélioration possible de notre planificateur HTN serait de pouvoir traiter des sous-tâches imbriquées, c'est-à-dire des sous-tâches qui seraient une liste de sous-tâches. Ceci serait particulièrement utile dans le cas des sous-tâches de type alternatives.

La hiérarchie de méthodes et d'opérateurs que nous avons créée est une traduction du symbolisme utilisé dans notre architecture GOAP. Tout comme pour cette architecture, le joueur possède un seul but actif en tout temps, mais dans le cas de la planification HTN, ce but est une méthode appelée *tête*, c'est-à-dire la première méthode que le planificateur tente de réaliser (elle constitue alors la seule tâche dans la liste de tâches). Une fois la tête déterminée, celle-ci est envoyée au planificateur HTN qui déterminera le meilleur plan grâce au réseau de tâche que nous avons créé.

Tout comme pour l'architecture GOAP, les méthodes plus spécifiques ont un coût moins élevé que les méthodes plus générales pour prioriser les méthodes spécifiques. Également, encore une fois comme pour l'architecture GOAP, chaque opérateur dans le plan produit par le planificateur est traduit en un « sous-plan » d'actions atomiques lorsque vient le temps de l'exécuter. Toutes les actions atomiques sont alors exécutées par le joueur jusqu'à ce que le sous-plan soit complété. Finalement, une représentation du monde de plus bas niveau, la mémoire du joueur, est également utilisée par le planificateur HTN pour vérifier les préconditions des actions, tout comme pour l'architecture GOAP. Bien entendu, cette représentation est mise à jour tout au long de la recherche en y appliquant les effets des actions considérées par le planificateur. Toutes les méthodes et tous les opérateurs de notre architecture ainsi que le pseudocode démontrant comment la tête de l'agent est déterminé sont présentés à l'annexe C.

Également, afin de démontrer que notre architecture HTN et que notre archicture GOAP sont équivalentes, la correspondance entre leurs actions est présentée à l'annexe D. Il est possible d'y remarquer que chaque action GOAP correspond à une méthode ou un opérateur HTN. Par contre, le contraire n'est pas vrai. En effet, les têtes de l'architecture HTN ne possèdent pas d'équivalence dans l'architecture GOAP, puisque celles-ci sont représentées

par le but courant. Également, les actions qui se divisent en sous-actions alternatives (telles que *Kill enemy*, *Knife*) ne possède pas d'équivalence dans l'architecture GOAP puisqu'il s'agit tout simplement d'autres branches que l'algorightme de recherche prendra. Finalement certains opérateurs de l'architecture HTN ne correspondent également pas à aucune action de l'architecture GOAP, car la méthode mère est équivalente à une action GOAP qui est assez précise pour déterminer les actions à effectuer dans le sous-plan. Toutes les actions de l'architecture HTN qui n'ont pas d'actions correspondantes dans l'architecture GOAP ont un coût de zéro. Ceci fait en sorte que l'architecture HTN génère généralement le même plan que l'architecture GOAP lorsque confrontée à la même situation. En effet, il est possible que les deux architectures génèrent des plans différents pour la même situation puisque l'architecture HTN produira toujours le plan optimal, alors que ce n'est pas le cas pour l'architecture GOAP puisque notre heuristique n'est pas admissible tel que mentionné à la section 6.3.2.

CHAPITRE 7

Expérimentation et résultats

Ce chapitre présente notre expérimentation et nos résultats. Tout d'abord, notre méthode d'évaluation des architectures de prise de décision est présentée. Par la suite, nous présentons nos résultats obtenus suivis d'une discussion portant sur les métriques.

7.1 Évaluation des architectures de prise de décision développées

Cette activité consiste à effectuer les expérimentations nous permettant de comparer nos différentes architectures de prise de décision. Pour ce faire, nous avons tout d'abord implémenté un générateur automatique de niveaux. Celui-ci génère des niveaux de jeu aléatoires de la façon suivante. Chaque monde possède 30 rangées de 30 cellules, soit un total de 900 cellules. Les murs sont tout d'abord placés dans le monde. Pour ce faire, le monde est découpé en neuf régions de taille identique présentées sur la figure 7.1.

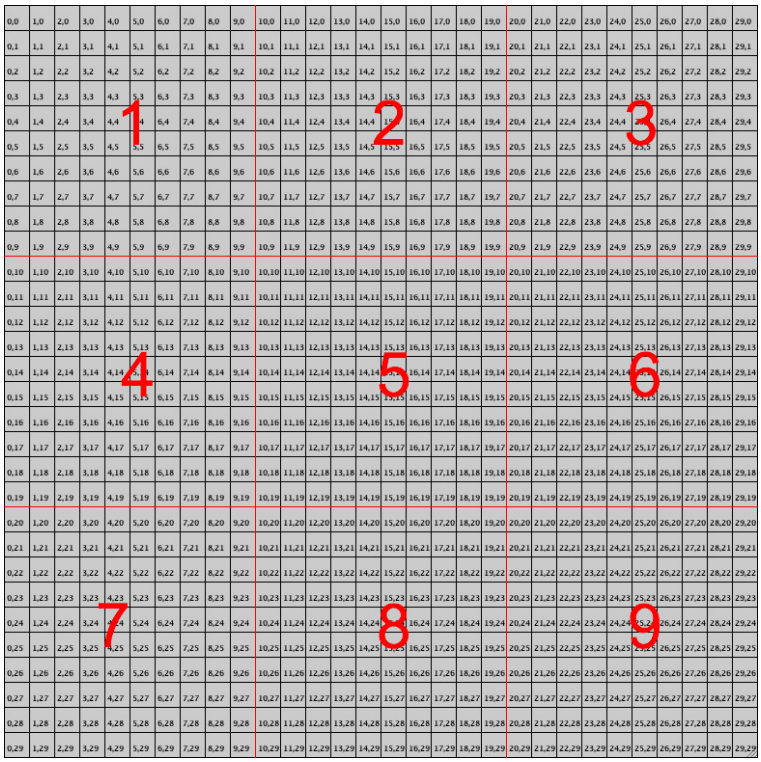


Figure 7.1 Découpage des mondes en régions.

Pour chaque région, environ une cinquantaine de patrons de murs ont tout d'abord été créés. Ces patrons nous assurent que les murs sont placés de façons intéressantes, de sorte qu'ils forment une pièce fermée par exemple. Un patron est choisi aléatoirement pour chacune des régions du monde. Ensuite, un nombre aléatoire de patrons de la région centrale (région 5 sur la figure 7.1) est ajouté au monde, étant donné que les patrons de cette région sont intéressants dans n'importe quelle région puisqu'elle ne possède pas de frontières. Par la suite, un nombre aléatoire compris entre 0 et une borne supérieure de murs sont ajoutés à des positions aléatoires également. Ceci permet de diversifier les mondes sans pour autant détruire les patrons intéressants de murs. Pour terminer le placement des murs, un nombre aléatoire compris entre 0 et une borne supérieure de murs sont enlevés du monde. Ceci permet de diversifier les mondes et de permettre de créer des pièces fermées plus intéressantes en sachant que les trous générés seront remplacés par d'autres entités. Par exemple, en enlevant un mur d'une pièce fermée, l'espace vide généré pourra être comblé par une porte, créant ainsi une pièce accessible différemment. Lorsque les murs sont placés dans le monde, les autres objets sont ajoutés. Pour ce faire, pour chaque type d'objet, un nombre aléatoire compris entre 0 et une borne supérieure de ce type d'objet sont ajoutés à des positions aléatoires. Afin de démontrer que les niveaux générés sont suffisamment différents, nous avons compilés les positions des entités et leur nombre par partie pour environ 50 000 niveaux générés aléatoirement. Ces statistiques sont présentés à l'annexe E.

Une fois le générateur de niveaux terminé, nous avons fait compétitionner nos différentes architectures les unes contre les autres dans des niveaux aléatoires. Pour ce faire, le cycle FSM vs. GOAP $\rightarrow$ FSM vs. HTN $\rightarrow$ GOAP vs. HTN est constamment répété. Puisqu'il n'est pas garanti que les niveaux générés soient valides, c'est-à-dire qu'il se peut qu'ils soient impossibles à terminer si les joueurs commencent à des endroits où ils sont bloqués, nous avons mis un temps limite après lequel la partie est interrompue. Les résultats des parties interrompues n'ont pas été compilés. De plus, puisque la position initiale d'un joueur peut l'avantager ou le désavantager, chaque partie est effectuée deux fois en interchangeant les positions de départ des deux joueurs.

Finalement, nous avons comparé les architectures de prise de décision selon deux approches différentes. Pour la première approche, chaque joueur est un processus léger (*thread*) qui prend sa décision de manière indépendante. Lorsqu'il a déterminé sa prochaine action, il l'envoie au maître du jeu qui va l'effectuer et mettre à jour son modèle de perception. Dès que ce dernier est mis à jour, le processus de prise de décision recommence. Cette approche défavorise les architectures nécessitant parfois beaucoup de temps de calcul, car rien

n'empêche un joueur d'effectuer plusieurs actions pendant que l'autre tente de choisir sa prochaine action. La seconde approche tente d'éliminer l'impact du temps de calcul requis pour prendre une décision sur les performances des architectures de prise de décision. Pour ce faire, tous les joueurs débutent leur prise de décision en même temps. Les deux joueurs sont alors bloqués jusqu'à ce qu'ils aient tous deux pris leur décision. Lorsque c'est le cas, le maître décidera aléatoirement lequel des deux joueurs effectuera son action en premier. Cette approche est justifiée par le fait qu'il est toujours possible d'optimiser nos architectures au niveau temps de calcul et qu'il serait également possible d'étaler leurs traitements sur plusieurs trames pendant que l'agent effectue ses actions pour que la prise de décision s'effectue toujours à l'intérieur d'un interval de temps requis. Nos résultats ont été amassés pour les deux approches.

7.2 Résultats obtenus et discussion

Les résultats obtenus sont présentés ci-dessous sous forme de tableaux. Les résultats ont été compilés sur environ 30 000 parties non interrompues pour l'approche tenant compte du temps de calcul et 75 000 parties non interrompues pour l'approche en faisant abstraction. Afin de s'assurer de la validité des résultats, un test d'hypothèses avec un intervalle de confiance bilatéral à 95% a été effectué pour toutes les moyennes et les variances utilisées dans le calcul de nos métriques. Pour les moyennes, le test utilisé fût un test t de Student avec l'hypothèse nulle que la moyenne égale zéro, c'est-à-dire que

$$H_0 : \mu = 0$$

$$H_1 : \mu \neq 0$$

alors que pour les variances, le test utilisé fût un test chi carré avec l'hypothèse nulle que la variance égale zéro, c'est-à-dire que

$$H_0 : \sigma^2 = 0$$

$$H_1 : \sigma^2 \neq 0$$

Pour tous ces tests, les valeurs des moyennes et des variances se trouvent dans l'intervalle de confiance avec une p-value inférieur à $2,2e-16$.

7.2.1 Qualité de l'agent

Tableau 7.1 Résultats obtenus quant à la qualité de l'agent.

	Approche tenant compte du temps de calcul			Approche faisant abstraction du temps de calcul		
	FSM	GOAP	HTN	FSM	GOAP	HTN
Nombre de victoires	14455	15510	14551	26906	43856	42950
Nombre de défaites	14457	14636	15423	47150	32912	33650
Nombre de parties nulles	48	36	32	98	76	62
Taux de victoires (%)	49,91	51,39	48,49	36,28	57,07	56,03
Pourcentage de parties terminées normalement	15,07	11,02	10,89	12,50	9,23	9,01
Pourcentage de parties pour lesquelles le joueur s'est fait éliminer	43,79	42,60	45,08	58,37	37,91	38,82
Pourcentage de parties pour lesquelles le joueur a éliminé l'autre joueur	41,10	46,29	43,95	29,11	52,84	52,14
Pointage moyen	603,62	561,66	542,34	536,35	565,32	554,05

Il est possible de remarquer sur le tableau 7.1 que le temps de calcul affecte beaucoup la performance des architectures. L'architecture utilisant la machine à états finis est avantagée pour l'approche tenant compte du temps de calcul. En effet, un agent possédant une architecture utilisant un planificateur nécessite beaucoup de temps de calcul pour planifier. Lorsqu'il tente de faire un plan, l'agent utilisant une machine à états pourra faire plusieurs actions pendant que son adversaire restera immobile. Ceci confère à l'agent utilisant une machine à états un avantage non négligeable. On peut remarquer que la qualité de l'agent utilisant la machine à états est supérieure pour l'approche tenant compte du temps de calcul tandis que celle des architectures utilisant la planification est inférieure pour cette approche. Ceci s'explique par le temps de calcul maximal des différentes architectures. En observant les résultats de l'approche tenant compte du temps de calcul, on remarque que le temps de calcul moyen maximal pour une partie est de 89 ms pour la machine à états finis, 226 ms pour l'architecture GOAP et 327 ms pour l'architecture HTN. Bien que l'architecture HTN devrait être avantagée par rapport à l'architecture GOAP grâce à la replanification partielle, ceci n'est pas démontré dans les résultats. Ceci est possiblement dû au fait que son temps de

calcul maximal est beaucoup plus élevé que l'architecture GOAP. Malgré ce désavantage, il est intéressant de remarquer que les architectures utilisant la planification obtiennent à peu près les mêmes performances que l'architecture FSM.

Pour ce qui est de l'approche faisant abstraction du temps de calcul, on peut remarquer que la qualité des agents avec une architecture utilisant la planification est supérieure à celle de l'agent utilisant une machine à états finis. Également, comme prévu, les résultats de l'architecture GOAP et de l'architecture HTN sont semblables. La qualité de l'agent de l'architecture GOAP légèrement supérieure à celle de l'architecture HTN peut s'expliquer par le fait que l'architecture HTN effectue parfois une replanification partielle. En effet, il est possible que les plans choisis par l'architecture GOAP soient parfois plus avantageux et que la replanification partielle ne confère pas l'avantage auquel nous nous attendions. Il est possible que, puisque la replanification partielle s'effectue à partir d'une situation pour laquelle le plan courant fût invalidé, il soit plus avantageux de réévaluer toutes les possibilités de nouveaux plans, ce que fait l'architecture GOAP lorsqu'elle effectue une replanification totale.

7.2.2 Qualité de la planification

Tableau 7.2 Résultats obtenus quant à la qualité de la planification.

	Approche tenant compte du temps de calcul			Approche faisant abstraction du temps de calcul		
	FSM	GOAP	HTN	FSM	GOAP	HTN
Temps de calcul moyen (ms)	6,29	3,87	4,29	5,07	3,40	3,85
Nombre moyen d'actions par plan	-	2,88	2,09	-	2,85	2,02
Nombre moyen d'actions atomiques par plan	-	9,55	9,90	-	9,34	9,23
Nombre de planifications par partie	-	31,03	25,89	-	26,25	26,42
Nombre de replanifications totales par partie	-	2,99	1,21	-	3,01	0,87
Nombre de replanifications partielles par partie	-	-	5,33	-	-	2,50

On remarque sur le tableau 7.2 que les architectures utilisant la planification exigent en moyenne moins de temps de calcul que l'architecture utilisant une machine à états finis. Ceci est explicable par le fait que la majorité du temps de calcul est passée à déterminer le plan. Une fois celui-ci déterminé, identifier la prochaine action est presque instantané (seule la traduction de l'action du plan en plan d'actions atomiques exige du temps de calcul à ce moment). Il est également possible de remarquer que, pour l'approche tenant compte du temps de calcul, l'architecture HTN planifie en moyenne moins dans une partie que l'architecture GOAP. Ceci est dû au fait que l'architecture HTN soit plus lente et explique la qualité de l'agent plus faible pour l'architecture HTN comme démontré sur le tableau 7.1. Pour l'approche faisant abstraction du temps, cette différence n'existe plus, comme prévu. Pour cette approche, les deux architectures utilisant la planification semblent générer des plans de longueurs semblables et planifier et replanifier environ le même nombre de fois. Finalement, le tableau 7.1 montre que l'architecture HTN replanifie plus souvent pour l'approche tenant compte du temps de calcul. Ceci s'explique par le fait que les autres architectures produisent plus d'actions que l'architecture HTN pour cette approche. Puisque les actions de l'adversaire peuvent invalider un plan, les plans produits par l'architecture HTN sont plus souvent invalidés par les actions de son adversaire.

7.2.3 Jouabilité

Tableau 7.3 Résultats obtenus quant à la jouabilité.

	Approche tenant compte du temps de calcul			Approche faisant abstraction du temps de calcul		
	FSM	GOAP	HTN	FSM	GOAP	HTN
Niveau de difficulté approprié de l'agent T	1,000	0,990	0,983	1,000	0,989	0,990
Niveau de difficulté approprié de l'agent T_2	0,869	0,882	0,885	0,893	0,888	0,886
Diversité du comportement de l'agent S	0,247	0,080	0,116	0,216	0,084	0,072
Diversité du comportement de l'agent S_2	0,209	0,203	0,200	0,201	0,194	0,202
Diversité spatiale de l'agent H_n	0,918	0,922	0,919	0,919	0,922	0,919
Jouabilité I	0,602	0,566	0,570	0,594	0,565	0,565

On remarque sur le tableau 7.3 que la jouabilité est semblable pour les trois architectures de prise de décision. Les seules métriques qui diffèrent beaucoup entre la machine à états finis et les architectures utilisant la planification sont le niveau de difficulté approprié de l'agent T et la diversité du comportement de l'agent S. Cependant, puisque ces métriques utilisent le temps moyen mis par l'agent pour éliminer son adversaire (t_k) et que ce dernier est calculé de façon différente pour la machine à états finis, la comparaison n'est pas pertinente. Par contre, dû à la similarité des deux architectures utilisant la planification, la comparaison des métriques utilisant t_k (c'est-à-dire T et S) est valable entre les architectures GOAP et HTN. Malgré tout, la jouabilité de la machine à états est supérieure aux architectures utilisant la planification.

En résumé, en tenant compte du temps de calcul, les architectures utilisant la planification possèdent une qualité de l'agent semblable à la machine à états finis et ce, sans optimisation. Par contre, celles-ci sont désavantagées puisque calculer un plan peut nécessiter plusieurs millisecondes. De ce fait, en faisant abstraction du temps, la qualité de l'agent est supérieure à celle de la machine à états finis. Également, les planificateurs nécessitent en moyenne moins de temps de calcul que la machine à états finis. Par contre, cette dernière possède une joua-

bilité supérieure aux planificateurs, que l'on fasse abstraction du temps de calcul ou non.

À la lumière de ces résultats, nous pouvons conclure que les performances des architectures utilisant la planification sont intéressantes par rapport à la machine à états finis. Il semble donc approprié d'utiliser les planificateurs dans une architecture de prise de décision d'un jeu vidéo. Nous avons compilé les résultats des architectures GOAP et HTN contre la machine à états finis afin de déterminer le planificateur le plus approprié. Les résultats sont présentés dans les tableaux ci-dessous.

Tableau 7.4 Résultats obtenus quant à la qualité de l'agent pour les architectures GOAP et HTN contre la machine à états finis.

	Approche tenant compte du temps de calcul		Approche faisant abstraction du temps de calcul	
	GOAP	HTN	GOAP	HTN
Nombre de victoires	7416	7041	23819	23331
Nombre de défaites	7126	7329	13293	13613
Nombre de parties nulles	26	22	56	42
Taux de victoires (%)	50,91	48,92	64,08	63,08
Pourcentage de parties terminées normalement	15,13	14,97	14,21	12,29
Pourcentage de parties pour lesquelles le joueur s'est fait éliminer	40,23	41,99	27,46	29,61
Pourcentage de parties pour lesquelles le joueur a éliminé l'autre joueur	44,56	43,00	58,33	58,08
Pointage moyen	589,72	569,48	611,41	597,42

Tableau 7.5 Résultats obtenus quant à la qualité de la planification pour les architectures GOAP et HTN contre la machine à états finis.

	Approche tenant compte du temps de calcul		Approche faisant abstraction du temps de calcul	
	GOAP	HTN	GOAP	HTN
Temps de calcul moyen (ms)	3,70	4,25	4,21	3,86
Nombre moyen d'actions par plan	2,86	2,09	2,84	2,00
Nombre moyen d'actions atomiques par plan	9,55	10,01	9,33	9,21
Nombre de planifications par partie	33,31	27,63	29,02	29,08
Nombre de replanifications totales par partie	3,00	1,08	3,01	0,91
Nombre de replanifications partielles par partie	-	5,40	-	2,48

Tableau 7.6 Résultats obtenus quant à la jouabilité pour les architectures GOAP et HTN contre la machine à états finis

	Approche tenant compte du temps de calcul		Approche faisant abstraction du temps de calcul	
	GOAP	HTN	GOAP	HTN
Niveau de difficulté approprié de l'agent T	0,989	0,979	0,985	0,990
Niveau de difficulté approprié de l'agent T_2	0,875	0,879	0,878	0,867
Diversité du comportement de l'agent S	0,083	0,139	0,105	0,071
Diversité du comportement de l'agent S_2	0,197	0,203	0,207	0,219
Diversité spatiale de l'agent H_n	0,924	0,921	0,923	0,922
Jouabilité I	0,564	0,573	0,570	0,568

En somme, les tableaux 7.4, 7.5 et 7.6 montrent que les architectures GOAP et HTN possèdent des performances globales similaires et, comme prévu, encore plus semblables pour l'approche faisant abstraction du temps de calcul. Néanmoins, l'architecture GOAP offre une qualité d'agent légèrement supérieure à l'architecture HTN.

7.3 Discussion sur les métriques qualitatives

Tel que mentionné à la section 6.1.5, nos observations et nos expériences personnelles d'implémentation nous ont permis de comparer les différentes architectures au niveau des limitations comportementales des agents, de la facilité d'implémentation et de la robustesse face aux changements de design.

7.3.1 Limitations comportementales

La principale limitation comportementale que nous avons identifiée est la difficulté des agents utilisant la planification à avoir un comportement réactif. Par exemple, lorsqu'un joueur décide qu'il veut éliminer son adversaire au corps-à-corps, l'agent utilisant la planification planifiera la séquence d'actions lui permettant de se rendre jusqu'à son adversaire et de le l'éliminer au corps-à-corps par la suite. Par contre, même si son adversaire bouge,

son plan n'est pas invalidé. Il devra donc se rendre à la position où était sa cible auparavant avant d'effectuer la replanification. Le joueur risque donc de pourchasser longtemps son adversaire avant de l'éliminer. Même avec des conditions plus strictes pour invalider les plans, par exemple une condition invalidant un plan destiné à éliminer au corps-à-corps son adversaire dès que celui-ci bouge, l'agent effectuerait beaucoup de replanification, ce qui le désavantagerait dans l'approche où le temps de calcul a un impact sur les performances des architectures de prise de décision par rapport à un agent utilisant une machine à états pour lequel il n'y a pas de transition d'états lorsqu'il a un but fixe (comme éliminer son adversaire au corps-à-corps).

7.3.2 Facilité d'implémentation

L'architecture la plus difficile à implémenter est certainement l'architecture GOAP. En effet, il faut tout d'abord créer le symbolisme STRIPS servant à représenter l'état du monde ainsi que les préconditions et les effets des actions. Cette tâche nécessite une bonne réflexion, mais est tout de même d'un niveau de complexité raisonnable. Par la suite, il faut créer les actions. Ceci est plutôt simple, puisque l'on sait ce que l'agent doit faire. Par contre, il ne faut pas oublier que chaque action possède des préconditions, des effets et un coût. Déterminer les préconditions et leur ordre est une tâche assez complexe. En effet, l'ordre des préconditions est très important dans une architecture GOAP, car la première précondition deviendra toujours le but que le planificateur tentera d'accomplir en premier. Elles ont donc une influence sur l'ordre dans lequel les actions seront effectuées sur le joueur. La difficulté majeure vient du fait que le planificateur effectue une recherche en commençant par le but, ce qui fait en sorte que les premières préconditions seront vérifiées par des actions que le joueur effectuera en réalité en dernier. En d'autres termes, pour chaque but que le planificateur tente d'accomplir, il faut garder en tête que toutes les actions déjà déterminées se trouvant dans le plan seront en réalité effectuées après et que toutes les prochaines actions qui seront ajoutées au plan s'effectueront avant. Il faut donc faire très attention à l'ordre des préconditions. La recherche arrière effectuée par le planificateur entraîne d'autres problèmes. Effectivement, tel qu'énoncé à la section 6.3.2, le planificateur n'utilise pas seulement un état du monde symbolique pour effectuer sa recherche. Une représentation moins grossière du monde, la mémoire du joueur, représentée par la grille du jeu, est nécessaire pour savoir si une action est possible ou non. Chaque action possède donc des préconditions et des effets sur cette représentation du monde également. Puisque la recherche se fait vers l'arrière, le joueur ne sait pas à priori comment sera le monde lorsqu'il tente des actions pour accomplir le but courant. Il faut donc faire très attention à comment cette représentation du monde est modifiée par les actions pour ne pas

que des actions futures influencent le présent, mais que les effets des actions futures ne soient tout de même pas ignorés. Afin de mieux comprendre ces problèmes, illustrons les par des exemples.

L'exemple suivant démontre le cas où une action future influence le présent. Supposons la situation présentée à la figure 7.2.

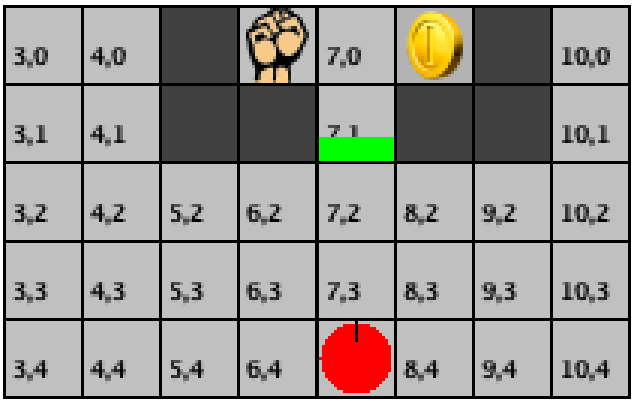
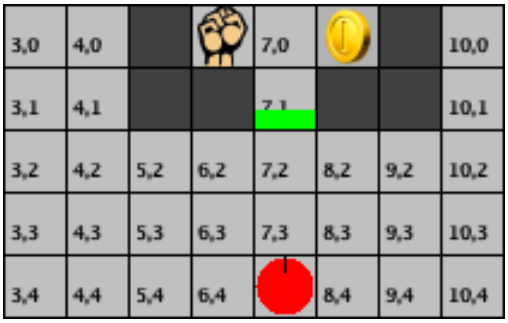


Figure 7.2 Exemple de cas où une action future influence le présent.

Supposons le cas où le joueur désirerait aller chercher le jeton situé à la position (8,0). Bien entendu, il n'est pas possible de générer un tel plan, mais présentons comment un plan invalide peut être généré si les représentations du monde ne sont pas modifiées correctement. Supposons que, lorsqu'un objet est amassé, la mémoire du joueur est modifiée de telle manière que cet objet n'apparaît plus dans celle-ci. Le premier noeud de l'arbre de recherche contiendrait les informations suivantes :

État symbolique courant : *STRENGTH(0)*
Mémoire courante :



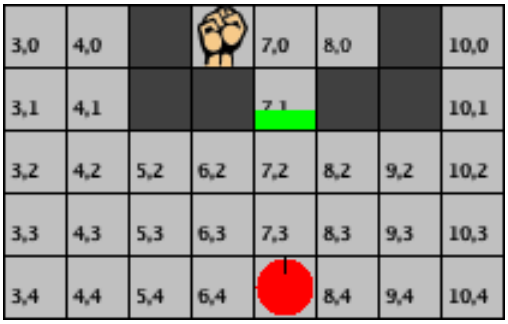
But(s) à réaliser :
NO_ITEM_AT(8,0)
Action(s) considérée(s) : *PICK(8,0)*

L'action *PICK* possède *AT* comme précondition. Celle-ci est donc ajoutée aux buts à réaliser. La seule action permettant de réaliser ce but est l'action *GO_TO*. Les informations du prochain noeud seront donc :

État symbolique courant :

- *STRENGTH(0)*
- *NO_ITEM_AT(8,0)*

Mémoire courante :



But(s) à réaliser : *AT(8,0)*
Action(s) considérée(s) : *GO_TO(8,0)*

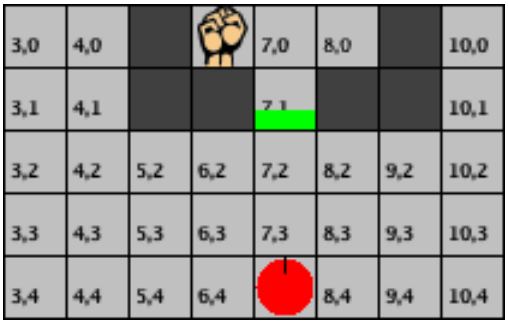
L'action *GO_TO* appelle le planificateur de trajectoire sur la mémoire pour déterminer

ses préconditions. Puisqu'il existe un obstacle bas dans la trajectoire de l'agent vers le jeton, la précondition *NO_LOW_OBSTACLE_AT* sera ajoutée comme but à réaliser. La seule action permettant de réaliser ce but est l'action *MOVE_LOW_OBSTACLE*. Les informations du prochain noeud seront :

État symbolique courant :

- *STRENGTH(0)*
- *NO_ITEM_AT(8,0)*

Mémoire courante :



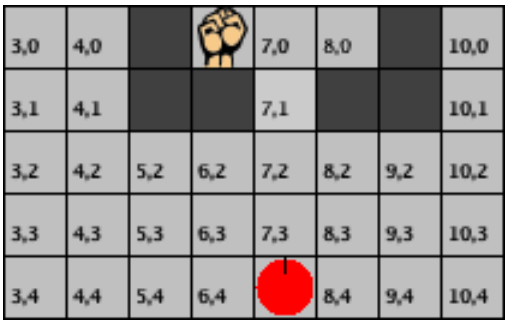
But(s) à réaliser : *NO_LOW_OBSTACLE_AT(7,1)*
Action(s) considérée(s) : *MOVE_LOW_OBSTACLE(7,1)*

L'action *NO_LOW_OBSTACLE_AT* exige que l'agent se trouve à la position de l'obstacle et possède 1 point de force. Les préconditions *AT* et *STRENGTH(1)* seront donc ajoutée aux buts à réaliser dans cet ordre, puisque l'agent doit avoir la force nécessaire avant d'aller déplacer l'obstacle. Le but à réaliser est donc *AT* et la seule action permettant de réaliser ce but est l'action *GO_TO*. Les informations du prochain noeud seront :

État symbolique courant :

- *STRENGTH(0)*
- *NOITEMAT(8,0)*
- *NO_LOW_OBSTACLE_AT(7,1)*

Mémoire courante :



But(s) à réaliser :

- *AT(7,1)*
- *STRENGTH(1)*

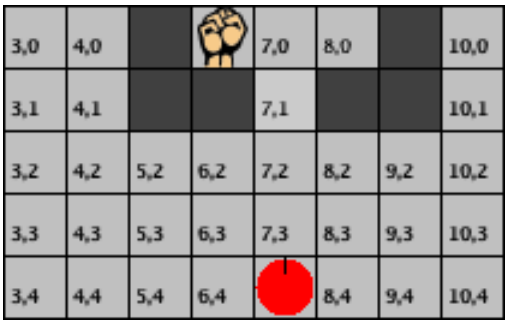
Action(s) considérée(s) : *GO_TO(7,1)*

Tel que mentionné plus haut, l'action *GO_TO* appelle le planificateur de trajectoire sur la mémoire pour déterminer ses préconditions. Puisque la trajectoire entre la position de l'agent (6,4) et l'obstacle (7,1) est directe, l'action *GO_TO* ne possède aucune précondition. Rien n'est ajouté à la liste de buts à réaliser. Le nouveau but à réaliser est donc *STRENGTH(1)*. Puisque notre agent ne possède aucun point de force (ce qui est traduit par *STRENGTH(0)* dans l'état symbolique courant), celui-ci doit aller récupérer un bonus de force. La seule action permettant de réaliser ce but est l'action *GET_STRENGTH*. Les informations du prochain noeud seront :

État symbolique courant :

- *STRENGTH(0)*
- *NO_ITEM_AT(8,0)*
- *NO_LOW_OBSTACLE_AT(7,1)*

Mémoire courante :



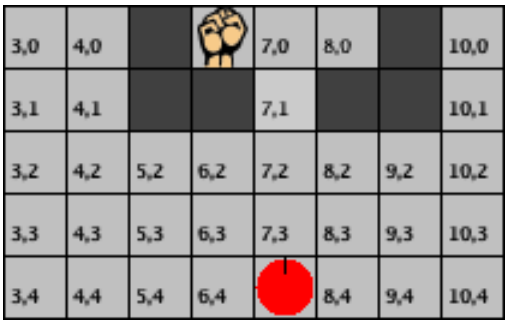
But(s) à réaliser : *STRENGTH(1)*
Action(s) considérée(s) : *GET_STRENGTH(6,0)*

L'action *GET_STRENGTH* possède *NO_ITEM_AT* comme précondition. Celle-ci est donc ajoutée aux buts à réaliser. La seule action permettant de réaliser ce but est l'action *PICK*. Les informations du prochain noeud seront donc :

État symbolique courant :

- *STRENGTH(1)*
- *NO_ITEM_AT(8,0)*
- *NO_LOW_OBSTACLE_AT(7,1)*

Mémoire courante :



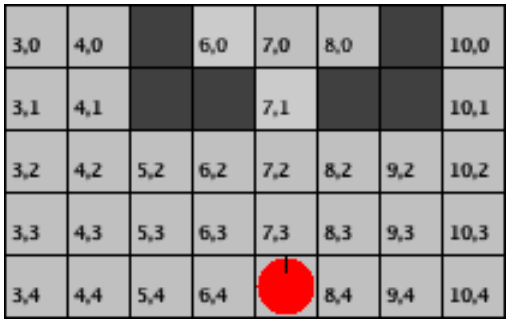
But(s) à réaliser : *NO_ITEM_AT(6,0)*
Action(s) considérée(s) : *PICK(6,0)*

Tel que mentionné plus haut, l'action *PICK* possède *AT* comme précondition. Celle-ci est donc ajoutée aux buts à réaliser. La seule action permettant de réaliser ce but est l'action *GO_TO*. Les informations du prochain noeud seront donc :

État symbolique courant :

- *STRENGTH(1)*
- *NO_ITEM_AT(6,0)*
- *NO_LOW_OBSTACLE_AT(6,1)*
- *NO_ITEM_AT(6,0)*

Mémoire courante :



But(s) à réaliser : *AT(6,0)*
Action(s) considérée(s) : *GO_TO(6,0)*

Tel que mentionné plus haut, l'action *GO_TO* appelle le planificateur de trajectoire sur la mémoire pour déterminer ses préconditions. Puisque la trajectoire entre la position de l'agent (6,4) et le bonus de force (12,4) est directe, l'action *GO_TO* ne possède aucune précondition. Rien n'est ajouté à la liste de buts à réaliser et comme celle-ci est vide, le plan est terminé. En remontant l'arbre, nous réalison que le plan produit est : *GO_TO(6,0)* $\rightarrow$ *PICK(6,0)* $\rightarrow$ *GET_STRENGTH(6,0)* $\rightarrow$ *GO_TO(7,1)* $\rightarrow$ *MOVE_LOW_OBSTACLE(7,1)* $\rightarrow$ *GO_TO(8,0)* $\rightarrow$ *PICK(8,0)*. Or, ce plan est invalide car l'agent ne peut se rendre à la position (6,0) sans avoir tout d'abord récupéré un bonus de force. Ce problème peut paraître simple à résoudre en évitant tout simplement d'appliquer des effets sur l'état courant, puisque de toute façon toutes les actions se trouvant dans le plan à tout moment n'ont jamais encore été effectuées en réalité. Par contre, ceci n'est pas possible, car un autre problème serait engendré : ignorer l'effet des actions futures peut engendrer des plans invalides.

L'exemple suivant démontre le cas où ignorer l'effet des actions futures engendre un plan invalide. Supposons la situation présentée à la figure 7.3.

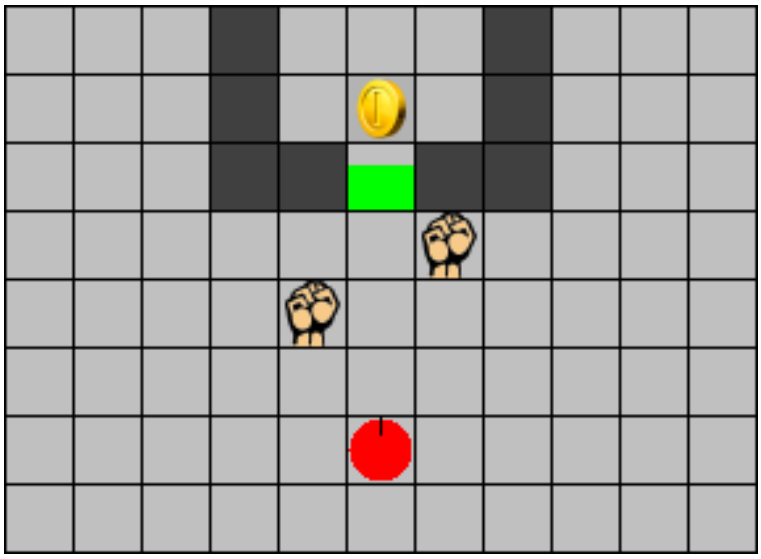


Figure 7.3 Exemple de cas où ignorer l'effet des actions futures engendre un plan invalide.

Le joueur désire récolter la pièce, mais celle-ci est bloquée par un obstacle nécessitant 2 de force. Le but courant du joueur deviendra donc d'augmenter sa force de 2. Une seule action permet d'accomplir ce but : aller chercher un bonus de force. Puisqu'il connaît l'emplacement de deux bonis de force, il choisira d'aller au bonus le plus près, soit celui se trouvant à sa gauche. Cette action sera donc ajoutée au plan. Supposons alors que l'état courant n'est pas modifié par les effets de l'action. Puisque cette action ne possède pas de précondition, le prochain but à accomplir sera augmenter sa force de 1. Encore une fois, la seule action permettant d'accomplir est d'aller chercher un bonus de force. Le joueur choisira donc d'aller chercher la force le plus près, soit celle à sa gauche. Ayant maintenant toute la force requise, toutes les préconditions sont vérifiées et le plan peut être retourné. Par contre, le plan contiendra deux fois l'action d'aller chercher la force à la gauche du joueur, ce qui est invalide. Afin de contourner les deux problèmes indiqués ci-dessus, nous avons choisi de ne pas enlever les objets amassés par le joueur de la représentation de bas niveau, mais plutôt de vérifier avec l'état courant symbolique s'ils étaient disponibles ou non. En effet, lorsqu'un item est amassé, le littéral *NO_ITEM_AT* est ajouté dans l'état symbolique. Celui-ci devient alors « non disponible ». Les objets marqués « non disponibles » ne peuvent être amassés puisqu'ils le seront dans le futur, mais doivent être considérés lors du calcul de trajectoires.

La gestion des effets n'est donc pas simple pour l'architecture GOAP. Ceci est en partie dû au fait qu'il existe deux représentations du monde indépendantes utilisées par le planificateur, soient la représentation symbolique et la représentation plus bas niveau (la mémoire du joueur) tel qu'énoncé à la section 6.3.2. Les effets doivent être appliqués différemment sur

chaque représentation, ce qui complique le problème (par exemple, les effets futurs doivent **toujours** être appliqués sur la représentation symbolique de l'état courant).

Également, tel que mentionné par J. Orkin [89, 90, 91, 92], il est vrai qu'il n'est pas nécessaire de penser à toutes les situations auxquelles le PNJ fera face lorsqu'on conçoit une architecture GOAP. Néanmoins, il ne faut pas oublier de penser aux actions qui **peuvent** mais ne **doivent** pas **obligatoirement** être effectuées avant d'autres actions. Dans ce cas, il faut générer d'autres actions avec les bonnes préconditions. Par exemple, nous savons que notre agent peut éliminer son adversaire en tirant sur lui. Par contre, il serait intéressant que le joueur aveugle son adversaire avant de le tirer. Néanmoins, « aveugler son adversaire » n'est pas une précondition à l'action « tirer », ce qui fait en sorte que ce comportement ne se produira jamais. Pour ce faire, une solution est de créer une action « tirer lorsque l'ennemi est aveuglé » qui possède « aveugler l'ennemi » comme précondition. Il faut par la suite ajuster les coûts des différentes actions pour effectuer les actions les plus spécifiques en premier. Ceci étant dit, ajuster les coûts des actions n'est pas simple dans une architecture GOAP, car les plans qui seront générés ne sont évidemment pas connus à priori. Il est donc difficile de savoir dans quels cas une action sera préférée à une autre qui répond aux mêmes préconditions. De plus, il est possible que nous ajustions les poids pour avoir un comportement désiré dans une certaine situation, mais que ceci fait en sorte que dans une autre situation le comportement ne soit pas optimal. Déterminer les coûts désirés nécessite donc plusieurs itérations. Une solution à ce problème permettant d'automatiser le processus et surtout d'optimiser les coût serait de faire apprendre les coûts à l'agent avec un algorithme d'itération de la valeur [99].

Pour ce qui est de la machine à états finis et de l'architecture HTN, les deux architectures sont simples à implémenter. Dans le cas de HTN, créer les méthodes et les opérateurs est un processus simple puisque la représentation de HTN est intuitive. De plus, HTN n'utilise pas de symbolisme STRIPS pour représenter le monde ainsi que les préconditions et les effets des actions. Une seule représentation de l'état courant (dans notre cas, il s'agit d'une représentation de la grille de jeu comme pour GOAP) est utilisée, ce qui est beaucoup plus simple que GOAP et prend moins de temps à concevoir. De plus, il est plus facile d'ajuster le coût des actions dans une architecture HTN que dans une architecture GOAP puisque les plans qui seront générés peuvent être plus facilement connus d'avance (puisqu'il ne suffit que de décomposer les hiérarchies). Néanmoins, l'ajustement des coûts n'est tout de même pas simple pour une architecture HTN et pourrait être automatisé de la même façon que pour une architecture GOAP décrite ci-dessus. Une difficulté de HTN se trouve au niveau de l'ordre dans lequel les actions sont effectuées. Tout comme GOAP, il faut faire attention

pour appliquer les effets correctement afin de ne pas engendrer les deux problèmes soulevés précédemment. Également, tout comme GOAP, il ne faut pas oublier de penser aux actions qui peuvent, mais ne doivent pas nécessairement être effectuées avant d'autres actions. Finalement, la machine à états finis est un peu plus simple à implémenter que l'architecure HTN. La tâche la plus complexe est de penser à toutes les transitions possibles entre les états, ce qui devient rapidement ardu lorsque le jeu atteint une certaine ampleur.

7.3.3 Robustesse

Notre projet a subi quelques changements de conception tout au long de son cycle de développement. Ces changements furent pour la plupart mineurs, mais à un certain moment dans le projet, le jeu a été grandement modifié : modification des règles, ajout d'objets, de nouvelles actions possibles, etc. Ce changement de conception majeur nous a permis de qualifier la robustesse de nos différentes architectures.

Nous en avons conclu que, dans le cadre de notre projet, les architectures GOAP et HTN ont été plus robustes face aux changements de design que l'architecture FSM. En effet, dans le cas des architectures GOAP et HTN, il suffit de modifier les actions et d'en ajouter/supprimer d'autres. L'architecture HTN est la plus robuste puisqu'il faut possiblement modifier le symbolisme dans le cas de GOAP. Pour ce qui est de l'architecture FSM, il a fallu refaire presque en entier notre machine à états. En effet, suite aux changements, la plupart des états n'étaient plus valides. Nous avons donc dû en créer de nouveaux, ainsi que les conditions de transition qui s'y rapportent.

Globalement, au niveau des métriques qualitatives, il a été plus difficile de donner un comportement réactif aux planificateurs dans notre projet. Par contre, ceux-ci possèdent l'avantage qu'il n'est pas nécessaire de penser à toutes les situations auxquelles l'agent fera face lorsqu'on les conçoit. Malgré tout, l'implémentation de l'architecture GOAP a été plus difficile que celle de la machine à états finis. L'implémentation de l'architecture HTN a pour sa part été simple grâce à sa représentation intuitive. Finalement, les architectures utilisant la planification se sont avérées plus robustes par rapport aux changements de conception lors de notre travail.

CHAPITRE 8

CONCLUSION

En conclusion, l'utilisation de planificateurs dans l'élaboration d'une architecture de prise de décision dans le cadre de notre projet présente des avantages par rapport à l'utilisation d'une machine à états finis. En effet, en faisant abstraction du temps de calcul des architectures, les architectures utilisant la planification offrent une qualité d'agent supérieure à la machine à états finis. Précisément, leur taux de victoire et leur pointage moyen sont plus élevés. En tenant compte du temps de calcul, malgré le fait qu'aucune des architectures n'ait été optimisée et que déterminer un plan peut parfois nécessiter un temps considérable, ces valeurs sont semblables à celles de la machine à états finis. Également, il est intéressant de noter que les architectures utilisant la planification nécessitent en moyenne moins de temps de calcul que la machine à états finis. En matière d'implémentation, il est vrai que les architectures de planification possèdent l'avantage qu'il n'est pas nécessaire de penser à toutes les situations auxquelles le PNJ fera face lorsqu'on les conçoit, tel que mentionné par J. Orkin [89, 90, 91, 92]. Aussi, les architectures de planification se sont montrées plus robustes aux changements de conception que la machine à états finis dans le cadre de notre travail. Toutefois, les planificateurs ne sont pas sans défauts. Nous avons remarqué que, dans le cadre de notre projet, il est plus difficile pour un agent utilisant la planification d'avoir un comportement réactif. Aussi, l'implémentation de l'architecture GOAP s'est avérée une tâche plus complexe que l'implémentation de la machine à états.

Contre la machine à états finis, les performances de l'architecture GOAP et de l'architecture HTN sont très semblables, ce qui s'explique par le fait que l'une est simplement la traduction de l'autre. Néanmoins, la qualité de l'agent de l'architecture GOAP est légèrement supérieure à celle de l'architecture HTN, alors que nous prévoyions le contraire. Il semble donc que la replanification partielle, caractéristique de la planification HTN, ne confère pas les avantages auxquels nous nous attendions. Cependant, l'architecture HTN a été plus facile à implémenter grâce à sa représentation intuitive. Les caractéristiques uniques des deux planificateurs leur confèrent donc certains avantages l'un par rapport à l'autre, sans toutefois pouvoir déclarer que l'un est plus approprié que l'autre dans le contexte de notre travail.

Somme toute, les planificateurs offrent des avantages intéressants par rapport à la machine à états finis et sont appropriés pour la conception d'une architecture de prise de décision d'un jeu vidéo. Toutefois, ces avantages diffèrent en fonction du type de planificateur et aucun des

deux planificateurs n'a été supérieur sur tous les points. Le choix du type de planificateur utilisé devrait donc être réalisé en fonction des exigences spécifiques du projet.

8.1 Travaux futurs

Il serait possible d'optimiser les architectures utilisant la planification au niveau du temps de calcul. Comme mentionné précédemment, déterminer un plan peut parfois nécessiter un temps considérable, ce qui désavantage les architectures utilisant la planification par rapport à la machine à états finis. Leur optimisation permettrait de refléter une situation d'un réel jeu de tir. Également, il serait intéressant d'étendre la comparaison à d'autres types de planificateurs, voire même d'autres techniques d'intelligence artificielle. Par exemple, une comparaison avec des planificateurs basés sur les cas, également déjà utilisés pour la prise de décision dans les jeux vidéo [88, 100, 108], pourrait être effectuée.

Au niveau de l'industrie, il serait possible d'effectuer ce genre de comparaison sur des jeux vidéo existants ou en développement. En effet, notre projet utilisait plutôt une simulation d'un jeu vidéo de tir. Puisque celui-ci décrit une méthodologie de test, celle-ci pourrait être appliquée à un véritable jeu pour valider les performances et la faisabilité des différentes architectures de prise de décision dans un contexte et sous les exigences d'un jeu vidéo commercial. Également, les tests pourraient être effectués avec des techniques d'intelligence artificielle qui semblent appropriés au domaine des jeux vidéo, mais dont la viabilité et l'efficacité n'ont pas encore été démontrées.

RÉFÉRENCES

[1] Activision. F.E.A.R. : First Encounter Assault Recon [En ligne]. 2009. Disponible : http://www.activision.com/index.html#gamepage|en_US|gameId:fear&brandId:fear [Consulté le 3 mars 2009].

[2] Alex J. Champandard. Enabling Concurrency in Your Behavior Hierarchy [En ligne]. 2007. Disponible : http://aigamedev.com/hierarchical-logic/parallel/ [Consulté le 9 mars 2011].

[3] Alex J. Champandard. Popular Approaches to Behavior Tree Design [En ligne]. 2007. Disponible : http://aigamedev.com/open/articles/popular-behavior-tree-design/ [Consulté le 9 mars 2011].

[4] Alex J. Champandard. The Flexibility of Selectors for Hierarchical Logic [En ligne]. 2007. Disponible : http://aigamedev.com/hierarchical-logic/selector/ [Consulté le 9 mars 2011].

[5] Alex J. Champandard. The Power of Sequences for Hierarchical Behaviors [En ligne]. 2007. Disponible : http://aigamedev.com/hierarchical-logic/sequence/ [Consulté le 9 mars 2011].

[6] Alex J. Champandard. Understanding Behavior Trees [En ligne]. 2007. Disponible : http://aigamedev.com/open/articles/bt-overview/ [Consulté le 9 mars 2011].

[7] Alex J. Champandard. Using Decorators to Improve Behaviors [En ligne]. 2007. Disponible : http://aigamedev.com/hierarchical-logic/decorator/ [Consulté le 9 mars 2011].

[8] Alex J. Champandard. Behavior Trees for Next-Gen Game AI [En ligne]. 2008. Disponible : http://aigamedev.com/insider/article/behavior-trees/ [Consulté le 9 mars 2011].

[9] Association for the Advancement of Artificial Intelligence. Planning & Scheduling [En ligne]. 2008. Disponible : http://www.aaai.org/AITopics/pmwiki/pmwiki.php/AITopics/Planning [Consulté le 16 juin 2009].

[10] Avrim Blum and Merrick Furst and John Langford. Graphplan home page [En ligne]. 2001. Disponible : http://www.cs.cmu.edu/~avrim/graphplan.html [Consulté le 17 juin 2009].

[11] F. Bacchus and F. Kabanza. Using temporal logic to control search in a forward chaining planner. *New directions in AI planning*, pages 141–156, 1996.

[12] J.A. Baier, F. Bacchus, and S.A. McIlraith. A heuristic search approach to planning with temporally extended preferences. *Artificial Intelligence*, 2008.

[13] Blizzard Entertainment. StarCraft [En ligne]. 2008. Disponible : http://www.blizzard.com/us/starcraft/ [Consulté le 3 mars 2009].

[14] A.L. Blum and M.L. Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2) :281–300, 1997.

[15] J. Blythe, O. Etzioni, Y. Gil, R. Joseph, D. Kahn, C. Knoblock, S. Minton, A. Perez, S. Reilly, M. Veloso, et al. Prodigy4. 0 : The manual and tutorial. Technical report, Tech. rep. CMU-CS-92-150, Carnegie Mellon University, Pittsburgh, PA, 1992.

[16] B. Bonet and H. Geffner. HSP : Heuristic search planner. *AIPS-98 Planning Competition, Pittsburgh, PA*, 1998.

[17] B. Bonet and H. Geffner. Heuristic search planner 2.0. *AI Magazine*, 22(3) :77–80, 2001.

[18] A. Botea, M. Enzenberger, M. Müller, and J. Schaeffer. Macro-FF : Improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research*, 24(1) :581–621, 2005.

[19] Mat Buckland. *Programming Game AI by Example*. Worldware Publishing, Plano, Texas, 2005.

[20] Hei Chan, Alan Fern, Soumya Ray, Nick Wilson, and Chris Ventura. Extending Online Planning for Resource Production in Real-Time Strategy Games with Search. Dans *International Conference on Automated Planning and Scheduling (ICAPS) Workshop on Planning in Games*, Providence, Rhode Island, USA, 2007.

[21] Hsueh-Min Chang and Von-Wun Soo. Simulation-Based Story Generation with Theory of Mind. Dans *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2008*, 4th Artifical Intelligence for Interactive Digital Entertainment Conference, pages 16–21, Stanford, CA, United States, 2008.

[22] Yun-Gyung Cheong, Arnav Jhala, Byung-Chull Bae, and R. Michael Young. Automatically Generating Summary Visualizations from Game Logs. Dans *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2008*, 4th Artifical Intelligence for Interactive Digital Entertainment Conference, pages 167–172, Stanford, CA, United States, 2008.

[23] M.T. Cox, H.É.C. MuÑOz-Avila, and R. Bergmann. Case-based planning. *The Knowledge Engineering Review*, 20(03) :283–287, 2006.

[24] K. Currie and A. Tate. *O-plan : the open planning architecture*. University of Edinburgh, Artificial Intelligence Applications Institute, 1989.

[25] Maria Cutumisu and Duane Szafron. Agent Learning using Action-Dependent Learning Rates in Computer Role-Playing Games. Dans *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2008*, 4th Artifical Intelligence for Interactive Digital Entertainment Conference, pages 218–219, Stanford, CA, United States, 2008.

[26] Tomas de la Rosa and Sergio Jiménez. ROLLER : A Lookahead Planner Guided by Relational Decision Trees. Dans *6th International Planning Competition*. Citeseer, 2008.

[27] Tomas de la Rosa, Angel Garcıa Olaya, and Daniel Borrajo. CABALA : Case-based State Lookaheads. Dans *6th International Planning Competition*. Citeseer, 2008.

[28] Darren Doherty and Colm O'Riordan. Effects of Communication on the Evolution of Squad Behaviours. Dans *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2008*, 4th Artifical Intelligence for Interactive Digital Entertainment Conference, pages 30–35, Stanford, CA, United States, 2008.

[29] Epic Games. Unreal Tournament [En ligne]. 2008. Disponible : http://www.unrealtournament2003.com/utgoty/index.html [Consulté le 5 mars 2009].

[30] K. Erol, J. Hendler, and D.S. Nau. Semantics for hierarchical task-network planning. 1994.

[31] K. Erol, J. Hendler, and D.S. Nau. HTN planning : Complexity and expressivity. Dans *Proceedings of the National Conference on Artificial Intelligence*, pages 1123–1123. JOHN WILEY & SONS LTD, 1995.

[32] Thierry Fayard. Using a Planner to Balance Real Time Strategy Video Game. Dans *International Conference on Automated Planning and Scheduling (ICAPS) Workshop on Planning in Games*, Providence, Rhode Island, USA, 2007.

[33] Susana Fernandez, Roberto Adarve, Miguel Perez, Martin Rybarczyk, and Daniel Borrajo. Planning for an AI based virtual agents game. Dans *International Conference on Automated Planning and Scheduling (ICAPS) Workshop on AI Planning for Computer Games and Synthetic Characters*, English Lake District, UK, 2006.

[34] R. Fikes and N.J. Nilsson. STRIPS : A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3/4) :189–208, 1971.

[35] FileFront. F.E.A.R. SDK Download, F.E.A.R. SDKs [En ligne]. 2005. Disponible : http://fear.filefront.com/file/FEAR_SDK;50693 [Consulté le 18 avril 2010].

[36] FileFront. Unreal Tournament Source Code 432 Download, Unreal Tournament 3 Utilities [En ligne]. 2005. Disponible : http://unrealtournament2004.filefront.com/file/Unreal_Tournament_Source_Code;50393 [Consulté le 18 avril 2010].

[37] Michelle Galea and John Levine. L2Plan2 : Learning Generalised Policies via Evolutionary Computation. Dans *6th International Planning Competition*. Citeseer, 2008.

[38] Rocıo Garcıa-Duran, Fernando Fernandez, and Daniel Borrajo. REPLICA : Relational Policies Learning in Planning. Dans *6th International Planning Competition*. Citeseer, 2008.

[39] A. Gerevini and D. Long. Plan constraints and preferences in PDDL3. Dans *ICAPS Workshop on Soft Constraints and Preferences in Planning*, 2006.

[40] A. Gerevini, A. Saetti, and I. Serina. Planning through stochastic local search and temporal action graphs in LPG. *Journal of Artificial Intelligence Research*, 20 :239–290, 2003.

[41] A. Gerevini, A. Saetti, I. Serina, and P. Toninelli. Planning in PDDL2. 2 domains with LPG-TD. *International Planning Competition booklet (ICAPS-04)*, 2004.

[42] Google Code. OpenNERO - game platform for Artificial Intelligence research and education [En ligne]. 2010. Disponible : http://code.google.com/p/opennero/ [Consulté le 18 avril 2010].

[43] Peter Gorniak and Ian Davis. SquadSmart : Hierarchical Planning and Coordinated Plan Execution for Squads of Characters. Dans *Proceedings of the Third Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2007*, 3rd Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2007, pages 14–19, Stanford, CA, United States, 2007.

[44] Johan Hagelback and Stefan J. Johansson. The Rise of Potential Fields in Real Time Strategy Bots. Dans *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2008*, 4th Artifical Intelligence for Interactive Digital Entertainment Conference, pages 42–46, Stanford, CA, United States, 2008.

[45] K.J. Hammond. CHEF : A model of case-based planning. Dans *Proceedings of the Fifth National Conference on Artificial Intelligence*, volume 1, 1986.

[46] Ahmed S. Hefny, Ayat A. Hatem, Mahmoud M. Shalaby, and Amir F. Atiya. Cerberus : Applying Supervised and Reinforcement Learning Techniques to Capture the Flag Games. Dans *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2008*, 4th Artifical Intelligence for Interactive Digital Entertainment Conference, pages 179–184, Stanford, CA, United States, 2008.

[47] M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26(2006) :191–246, 2006.

[48] K.E.J. Hendler and D.S. Nau. UMCP : A sound and complete procedure for hierarchical task network planning. Dans *Proc. 2nd Int'l Conf. AI Planning Systems (AIPS 94)*, pages 249–254. Citeseer, 1994.

[49] Carlos Hernandez and Pedro Meseguer. Improving Real-Time Heuristic Search on Initially Unknown Maps. Dans *International Conference on Automated Planning and Scheduling (ICAPS) Workshop on Planning in Games*, Providence, Rhode Island, USA, 2007.

[50] Hai Hoang, Stephen Lee-Urban, and H. Munoz-Avila. Hierarchical Plan Representations for Encoding Strategic Game AI. Dans *Proceedings of Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2005*, Stanford, California, United States, 2005. AAAI Press.

[51] J. Homann and B. Nebel. The FF planning system : Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14 :253–302, 2001.

[52] A. Howe and E. Dahlman. A critical assessment of benchmark comparison in planning. *Journal of Artificial Intelligence Research*, 17 :1–33, 2002.

[53] A.E. Howe, E. Dahlman, C. Hansen, M. Scheetz, and A. Von Mayrhauser. Exploiting competitive planner performance. *Lecture notes in computer science*, pages 62–72, 2000.

[54] id Software. id Software : Technology Downloads [En ligne]. 2001. Disponible : http://www.idsoftware.com/business/techdownloads/ [Consulté le 18 avril 2010].

[55] id Software. Quake III Arena [En ligne]. 2008. Disponible : http://www.quake3arena.com/games/quake/quake3-arena/ [Consulté le 3 mars 2009].

[56] Okhtay Ilghami. Documentation for jshop2. Technical Report UW-CS-TR-1481, University of Maryland - Department of Computer Science, 2006. Disponible : http://sourceforge.net/projects/shop/files/JSHOP2/1.0.3/JSHOP2_1.0.3.zip/download.

[57] International Planning Competition. IPC 2002 Website [En ligne]. 2002. Disponible : http://planning.cis.strath.ac.uk/competition/ [Consulté le 22 Juin 2009].

[58] International Planning Competition. IPC 2004 Website [En ligne]. 2004. Disponible : http://www.tzi.de/~edelkamp/ipc-4/ [Consulté le 22 Juin 2009].

[59] International Planning Competition. IPC 2006 Website [En ligne]. 2006. Disponible : http://zeus.ing.unibs.it/ipc-5/ [Consulté le 22 Juin 2009].

[60] International Planning Competition. IPC 2008 Website, Deterministic Part [En ligne]. 2008. Disponible : http://ipc.informatik.uni-freiburg.de/ [Consulté le 22 Juin 2009].

[61] D. Isla. Handling complexity in the Halo 2 AI. Dans *Game Developers Conference*, 2005.

[62] Damian Isla and Bruce Blumberg. Blackboard architectures. Dans *AI Game Programming Wisdom*, pages 333–344. Charles River Media, Hingham, Massachusetts, 2002.

[63] M. Renee Jansen and Nathan R. Sturtevant. Direction Maps for Cooperative Pathfinding. Dans *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2008*, 4th Artifical Intelligence for Interactive Digital Entertainment Conference, pages 185–190, Stanford, CA, United States, 2008.

[64] H. Kautz, D. McAllester, and B. Selman. Encoding plans in propositional logic. Dans *PRINCIPLES OF KNOWLEDGE REPRESENTATION AND REASONING-INTERNATIONAL CONFERENCE-*, pages 374–385. MORGAN KAUFMANN PUBLISHERS, 1996.

[65] H. Kautz and B. Selman. BLACKBOX : A new approach to the application of theorem proving to problem solving. Dans *In Workshop on Planning as Combinatorial Search, in conjunction with AIPS-98 (Conference on Artificial Intelligence Planning Systems*, 1985.

[66] H. Kautz, B. Selman, and J. Hoffmann. Satplan : Planning as satisfiability. Dans *5th International Planning Competition*. Citeseer, 2006.

[67] John-Paul Kelly, Adi Botea, and Sven Koenig. Planning with Hierarchical Task Networks in Video Games. Dans *International Conference on Automated Planning and Scheduling (ICAPS) Workshop on Planning in Games*, Providence, Rhode Island, USA, 2007.

[68] John-Paul Kelly, Adi Botea, and Sven Koenig. Offline Planning with Hierarchical Task Networks in Video Games. Dans *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2008*, 4th Artifical Intelligence for Interactive Digital Entertainment Conference, pages 60–65, Stanford, CA, United States, 2008.

[69] C.A. Knoblock. Search reduction in hierarchical problem solving. Dans *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 686–691, 1991.

[70] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. *Extending planning graphs to an ADL subset*. Springer, 1997.

[71] Lionhead Studios. Black & White [En ligne]. 2008. Disponible : http://www.lionhead.com/bw/Default.aspx [Consulté le 3 mars 2009].

[72] D. Long and M. Fox. Efficient implementation of the plan graph in STAN. *Journal of Artificial IntelligenceResearch*, 10 :87–115, 1999.

[73] D. Long, H. Kautz, B. Selman, B. Bonet, H. Geffner, J. Koehler, M. Brenner, J. Hoffmann, F. Rittinger, C.R. Anderson, et al. The AIPS-98 planning competition. *AI magazine*, 21(2) :13, 2000.

[74] D. McAllester, D. Rosenblitt, Massachusetts Institute of Technology, and Artificial Intelligence Laboratory. *Systematic nonlinear planning*. Citeseer, 1991.

[75] D. McDermott, M. Ghallab, A. Howe, C.A. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wikins. PDDL-the planning domain definition language. Technical report, Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[76] Michelle McPartland and Marcus R. Gallagher. Learning to be a Bot : Reinforcement learning in shooter games. Dans *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2008*, 4th Artifical Intelligence for Interactive Digital Entertainment Conference, pages 78–83, Stanford, CA, United States, 2008.

[77] Michelle McPartland and Marcus R. Gallagher. Creating a multi-purpose first person shooter bot with reinforcement learning. Dans *Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games*, 2008 IEEE Symposium on Computational Intelligence and Games, pages 143–150, Perth, Australia, 2009.

[78] Microsoft. Halo 2 [En ligne]. 2008. Disponible : http://www.microsoft.com/games/halo2/ [Consulté le 9 mars 2011].

[79] Ian Millington. *Artificial Intelligence for Games*. Morgan Kaufmann, San Fransisco, 2006.

[80] Monolith Productions. No One Lives Forever [En ligne]. 2003. Disponible : http://nolf2.sierra.com/site.html [Consulté le 3 mars 2009].

[81] Hector Munoz-Avila and Todd Fisher. Strategic planning for Unreal Tournament Bots. Dans *19th National Conference on Artificial Intelligence*, volume WS-04-04 of *AAAI Workshop - Technical Report*, pages 22–25, San Jose, CA, United States, 2004. American Association for Artificial Intelligence Menlo Park CA 94025-3496 United States.

[82] Hector Munoz-Avila and Hai Hoang. Coordinating teams of bots with hierarchical task networks planning. Dans *AI Game Programming Wisdom 3*, pages 417–427. Charles River Media, Hingham, Massachusetts, 2006.

[83] I. Murugeswari and NS Narayanaswamy. Tuning Search Heuristics for Classical Planning with Macro Actions. 2009.

[84] D. Nau, T.C. Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu, and F. Yaman. SHOP2 : An HTN planning system. *Journal of Artificial Intelligence Research*, 20(1) :379–404, 2003.

[85] D. Nau, Cao Yue, A. Lotem, and H. Munoz-Avila. SHOP : simple hierarchical ordered planner. Dans *Proceedings of Sixteenth International Joint Conference on Artificial Intelligence. IJCAI 99*, Stockholm, Sweden, 1999. Morgan Kaufmann Publishers Place of publication :San Francisco CA USA Material Identity Number :XX2001-01696.

[86] R.S. Nigenda, X.L. Nguyen, and S. Kambhampati. AltAlt : Combining the advantages of Graphplan and heuristic state search. *Proc. KBCS-2000, Mumbai, India*, 2000.

[87] John O'Brien. A flexible goal-based planning architecture. Dans *AI Game Programming Wisdom*, pages 375–383. Charles River Media, Hingham, Massachusetts, 2002.

[88] Santiago Ontanon, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. Case-based planning and execution for real-time strategy games. Dans *7th International Conference on Case-Based Reasoning, ICCBR 2007*, volume 4626 LNAI of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 164–178, Belfast, Northern Ireland, United Kingdom, 2007. Springer Verlag Heidelberg D-69121 Germany.

[89] Jeff Orkin. Applying goal-oriented action planning to games. Dans *AI Game Programming Wisdom 2*, pages 217–227. Charles River Media, Hingham, Massachusetts, 2003.

[90] Jeff Orkin. Symbolic representation of game world state : Toward real-time planning in games. Dans *19th National Conference on Artificial Intelligence*, volume WS-04-04 of *AAAI Workshop - Technical Report*, pages 26–30, San Jose, CA, United States, 2004. American Association for Artificial Intelligence Menlo Park CA 94025-3496 United States.

[91] Jeff Orkin. Agent Architecture Considerations for Real-Time Planning in Games. Dans *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2005*, Marina del Rey, California, USA, 2005.

[92] Jeff Orkin. Three States and a Plan : The A.I. of F.E.A.R. Dans *Game Developer's Conference*, San Jose, California, United States, 2006.

[93] E.P.D. Pednault. ADL and the state-transition model of action. *Journal of Logic and Computation*, 4(5) :467–512, 1994.

[94] J.S. Penberthy and D. Weld. UCPOP : A sound, complete, partial order planner for ADL. Dans *proceedings of the third international conference on knowledge representation and reasoning*, pages 103–114. Citeseer, 1992.

[95] David Pizzi, Marc Cavazza, Alex Whittaker, and Jean-Luc Lugrin. Automatic Generation of Game Level Solutions as Storyboards. Dans *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2008*, 4th Artifical Intelligence for Interactive Digital Entertainment Conference, pages 96–101, Stanford, CA, United States, 2008.

[96] Martha E. Pollack. Evaluating planners, plans, and planning agents. *SIGART Bull.*, 6(1) :4–7, 1995.

[97] Steve Rabin. Common game ai techniques. Dans *AI Game Programming Wisdom 2*, pages 3–14. Charles River Media, Hingham, Massachusetts, 2003.

[98] Steve Rabin. Promising game ai techniques. Dans *AI Game Programming Wisdom 2*, pages 15–27. Charles River Media, Hingham, Massachusetts, 2003.

[99] Stuart Russell and Peter Norvig. *Intelligence artificielle*. Pearson Education, Paris, 2006.

[100] Antonio Sanchez-Ruiz, Stephen Lee-Urban, Hector Munoz-Avila, Belen Diaz-Agudo, and Pedro Gonzalez-Calero. Game AI for a Turn-based Strategy Game with Plan Adaptation and Ontology-based retrieval. Dans *International Conference on Automated Planning and Scheduling (ICAPS) Workshop on Planning in Games*, Providence, Rhode Island, USA, 2007.

[101] Mei Si, Stacy C. Marsella, and Mark O. Riedl. Integrating Story-Centric and Character-Centric Processes for Authoring Interactive Drama. Dans *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2008*, Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2008, pages 203–208, Stanford, CA, United States, 2008.

[102] Stuart Slater, Kevan Buckley, and Kamal Bechkoum. Body Mind and Emotion, an Overview of Agent Implementation in Mainstream Computer Games. Dans *International Conference on Automated Planning and Scheduling (ICAPS) Workshop on AI Planning for Computer Games and Synthetic Characters*, English Lake District, UK, 2006.

[103] Christina R. Strong and Michael Mateas. Talking with NPCs : Toward Dynamic Generation of Discourse Structures. Dans *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2008*, 4th Artifical Intelligence for Interactive Digital Entertainment Conference, pages 114–119, Stanford, CA, United States, 2008.

[104] A. Tate. Generating project networks. Dans *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 888–893, 1977.

[105] TechRepublic. No One Lives Forever 2 Toolkit (Windows) [En ligne]. 2003. Disponible : http://software.techrepublic.com.com/abstract.aspx?docid=684183 [Consulté le 18 avril 2010].

[106] The Wargus Team. Wargus [En ligne]. 2007. Disponible : http://wargus.sourceforge.net/ [Consulté le 18 avril 2010].

[107] Paul Tozour. The evolution of game ai. Dans *AI Game Programming Wisdom*, pages 3–15. Charles River Media, Hingham, Massachusetts, 2002.

[108] Andrew Trusty, Santiago Ontanon, and Ashwin Ram. Stochastic Plan Optimization for Real-Time Strategy Games. Dans *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2008*, 4th Artifical Intelligence for Interactive Digital Entertainment Conference, pages 126–131, Stanford, CA, United States, 2008.

[109] Valve Software. Valve Developer Community [En ligne]. 2009. Disponible : http://developer.valvesoftware.com/wiki/Main_Page [Consulté le 18 avril 2010].

[110] P. Van Beek and X. Chen. CPlan : A constraint programming approach to planning. *constraints*, 100 :1, 1999.

[111] M. Veloso and J. Blythe. Linkability : Examining causal link commitments in partial-order planning. Dans *Proceedings of the Second International Conference on AI Planning Systems*, volume 72, 1994.

[112] Neil Wallace. Hierarchical planning in dynamic worlds. Dans *AI Game Programming Wisdom 2*, pages 229–236. Charles River Media, Hingham, Massachusetts, 2003.

[113] I. Warfield, C. Hogg, S. Lee-Urban, and H. Muñoz-Avila. Adaptation of hierarchical task network plans. Dans *Proceedings of the Twentieth International FLAIRS Conference (FLAIRS-07)*, 2007.

[114] Wikipedia. First-person shooter [En ligne]. 2008. Disponible : http://en.wikipedia.org/wiki/First_person_shooter [Consulté le 5 mars 2009].

[115] Wikipedia. Real-time strategy [En ligne]. 2008. Disponible : http://en.wikipedia.org/wiki/Real-time_strategy [Consulté le 5 mars 2009].

[116] Wikipedia. Strategy video game [En ligne]. 2008. Disponible : http://en.wikipedia.org/wiki/Strategy_video_game [Consulté le 5 mars 2009].

[117] Wikipedia. Third-person shooter [En ligne]. 2008. Disponible : http://en.wikipedia.org/wiki/Third_person_shooter [Consulté le 5 mars 2009].

[118] Wikipedia. Turn-based strategy [En ligne]. 2008. Disponible : http://en.wikipedia.org/wiki/Turn-based_strategy [Consulté le 5 mars 2009].

[119] Wikipedia. Video game industry [En ligne]. 2008. Disponible : http://en.wikipedia.org/wiki/Video_game_industry [Consulté le 17 novembre 2008].

[120] D.E. Wilkins. Can AI planners solve practical problems ? *Computational Intelligence*, 6(4) :232–246, 1990.

[121] G.N. Yannakakis and J. Hallam. Towards optimizing entertainment in computer games. *Applied Artificial Intelligence*, 21(10) :933–971, 2007.

ANNEXE A

Détails de la machine à états finis

Les conditions de transition de notre machine à états finis sont les suivantes :

- e2r = throwngrenade
- e2h = !throwngrenade $\wedge$ health $<$ 50 $\wedge$ healthavailable
- e2c = !throwngrenade $\wedge$ health $\geq$ 50 $\wedge$ enemy
- r2h = !throwngrenade $\wedge$ health $<$ 50 $\wedge$ healthavailable
- r2c = !throwngrenade $\wedge$ enemy $\wedge$ health $\geq$ 50
- r2e = !throwngrenade $\wedge$!enemy $\wedge$ health $\geq$ 50
- h2r = throwngrenade
- h2c = !throwngrenade $\wedge$ enemy $\wedge$ health $\geq$ 50
- h2e = !throwngrenade $\wedge$!enemy $\wedge$ health $\geq$ 50
- c2r = throwngrenade
- c2h = !throwngrenade $\wedge$ health $<$ 50 $\wedge$ healthavailable
- c2e = !throwngrenade $\wedge$!enemy $\wedge$ health $\geq$ 50
- c1 = (!canseeme $\wedge$!killwin $\wedge$!dartgun $\wedge$!flash) $\vee$ (canseeme $\wedge$!killwin $\wedge$!dartgun $\wedge$!flash) $\vee$ (canseeme $\wedge$ killwin $\wedge$!grenade $\wedge$!dartgun $\wedge$!flash) $\vee$ (canseeme $\wedge$!killwin $\wedge$!dartgun $\wedge$!flash) $\vee$ (frozen $\wedge$!killwin) $\vee$ (flashed $\wedge$!killwin)
- c2 = donemoving $\vee$ canseeme
- c3 = (!canseeme $\wedge$ killwin $\wedge$ ninja $\wedge$ magnum) $\vee$ (frozen $\wedge$ killwin $\wedge$ closeenoughfrozen) $\vee$ (flashed $\wedge$ killwin) $\vee$ (frozen $\wedge$ killwin $\wedge$!magnum $\wedge$!grenade) $\vee$ (flashed $\wedge$ killwin $\wedge$!magnum $\wedge$!grenade)
- c4 = (!flashed $\wedge$!frozen $\wedge$!canseeme $\wedge$ killwin $\wedge$!ninja $\wedge$!magnum $\wedge$ dartgun) $\vee$ (!flashed $\wedge$!frozen $\wedge$ canseeme $\wedge$!canattackme $\wedge$ killwin $\wedge$!magnum $\wedge$!grenade $\wedge$ dartgun) $\vee$ (!flashed $\wedge$!frozen $\wedge$!canseeme $\wedge$!killwin $\wedge$ dartgun) $\vee$ (!flashed $\wedge$!frozen $\wedge$ canseeme $\wedge$ canattackme $\wedge$ killwin $\wedge$!grenade $\wedge$ dartgun) $\vee$ (!flashed $\wedge$!frozen $\wedge$ canseeme $\wedge$ canattackme $\wedge$!killwin $\wedge$ dartgun) $\vee$ (!flashed $\wedge$!frozen $\wedge$ canseeme $\wedge$!canattackme $\wedge$!killwin $\wedge$ dartgun)
- c5 = (!canseeme $\wedge$ killwin $\wedge$!ninja $\wedge$!magnum $\wedge$!dartgun $\wedge$!flash $\wedge$ grenade) $\vee$ (canseeme $\wedge$!canattackme $\wedge$ killwin $\wedge$!magnum $\wedge$!flash $\wedge$ grenade) $\vee$ (canseeme $\wedge$ canattackme $\wedge$ killwin $\wedge$!flash $\wedge$ grenade) $\vee$ (frozen $\wedge$ killwin $\wedge$!magnum $\wedge$ grenade) $\vee$ (flashed $\wedge$ killwin $\wedge$!magnum $\wedge$ grenade)
- c6 = (!frozen $\wedge$!flashed $\wedge$!canseeme $\wedge$ killwin $\wedge$!ninja $\wedge$!magnum $\wedge$!dartgun $\wedge$ flash)

∨ (!frozen ∧ !flashed ∧ !canseeme ∧ !killwin ∧ !dartgun ∧ flash) ∨ (!frozen ∧ !flashed ∧ canseeme ∧ !canattackme ∧ killwin ∧ !magnum ∧ !dartgun ∧ flash) ∨ (!frozen ∧ !flashed ∧ canseeme ∧ !canattackme ∧ !killwin ∧ !dartgun ∧ flash) ∨ (!frozen ∧ !flashed ∧ canseeme ∧ canattackme ∧ killwin ∧ !dartgun ∧ flash) ∨ (!frozen ∧ !flashed ∧ canseeme ∧ canattackme ∧ !killwin ∧ !dartgun ∧ flash)

- c7 = (!canseeme ∧ killwin ∧ !ninja ∧ magnum) ∨ (canseeme ∧ !canattackme ∧ killwin ∧ magnum) ∨ (canseeme ∧ canattackme ∧ killwin ∧ !grenade ∧ !dartgun ∧ !flash ∧ magnum) ∨ (frozen ∧ killwin ∧ magnum) ∨ (flashed ∧ killwin ∧ magnum)
- c8 = killwin ∧ !canseeme ∧ !closeenough
- c9 = killwin ∧ !canseeme ∧ !closeenough

où

- throwngrenade = le joueur perçoit au moins une grenade qui a été lancée
- health = les points de vie du joueur
- healthavailable = il y a au moins un bonus de vie accessible au joueur présent dans sa mémoire
- enemy = le joueur a détecté la présence de son ennemi
- canseeme = l'ennemi peut voir le joueur
- canattackme = l'ennemi peut attaquer le joueur
- killwin = si le joueur élimine son adversaire, il remportera la partie
- flashed = l'ennemi est aveuglé
- frozen = l'ennemi est gelé
- magnum = le joueur possède un magnum
- dartgun = le joueur possède un fusil à darts
- grenade = le joueur possède au moins une grenade
- flash = le joueur possède au moins une grenade assourdissante
- ninja = le joueur possède un moins un bonus ninja
- closeenough = le joueur est assez près pour se rendre à temps à son ennemi pour le poignarder lorsque celui-ci est aveuglé ou gelé
- donemoving = le joueur est assez près pour poignarder l'ennemi

ANNEXE B

Détails de l'architecture GOAP

Les symboles que nous avons utilisés pour représenter un état dans l'architecture avec planificateur GOAP sont les suivants :

- AT(pos) = le joueur se trouve à la position « pos »
- CLOSE_ENOUGH_FROZEN = le joueur est assez près pour pouvoir poignarder son ennemi gelé à temps
- CLOSE_ENOUGH_NINJA = le joueur est assez près pour pouvoir poignarder son ennemi à temps en étant silencieux
- DARTGUN_EQUIPPED = le joueur est équippé du fusil à darts
- DARTGUN_READY = le fusil à darts est prêt à être utilisé, c'est-à-dire que son chargeur contient un dart
- ENEMY_DEAD = l'ennemi est mort
- ENEMY_FLASHED = l'ennemi est aveuglé
- ENEMY_FROZEN = l'ennemi est gelé
- HAS_DART = le joueur possède au moins un dart
- HAS_DARTGUN = le joueur possède le fusil à darts
- HAS_ENOUGH_HEALTH = le joueur a assez de vie (plus de 50 points)
- HAS_FLASHBANG = le joueur possède au moins une grenade assourdissante
- HAS_GRENADE = le joueur possède au moins une grenade
- HAS_KEY = le joueur possède au moins une clé
- HAS_MAGNUM = le joueur possède le magnum
- HAS_MAGNUM_AMMO = le joueur possède des munitions pour le magnum
- HAS_NINJA = le joueur possède au moins un bonus ninja
- INCREASE_SCORE = le joueur a augmenté son pointage
- KNIFE_EQUIPPED = le joueur est équipé du couteau
- MAGNUM_EQUIPPED = le joueur est équipé du magnum
- MAGNUM_READY = le magnum est prêt à être utilisé, c'est-à-dire que son chargeur contient des munitions
- NO_DOOR_AT(pos) = il n'y a pas de porte à la position « pos »
- NO_HIGH_OBSTACLE_AT(pos) = il n'y a pas d'obstacle haut à la position « pos »
- NO_ITEM_AT(pos) = il n'y a pas d'objet à la position « pos »

- NO_LOCKED_DOOR_AT(pos) = il n'y a pas de porte barrée à la position « pos »
- NO_LOW_OBSTACLE_AT(pos) = il n'y a pas d'obstacle bas à la position « pos »
- NO_MID_OBSTACLE_AT(pos) = il n'y a pas d'obstacle moyen à la position « pos »
- NOT_IN_ENEMYS_SIGHT = le joueur ne se trouve pas dans le champ de vision de son ennemi
- NOT_IN_GRENADE_RADIUS = le joueur ne se trouve pas dans le rayon d'une ou plusieurs grenades
- NO_WALL_AT(pos) = il n'y a pas de mur à la position « pos »
- NO_WINDOW_AT(pos) = il n'y a pas de fenêtre à la position « pos »
- SLOW_ENEMY = l'ennemi est ralenti (gelé ou aveuglé)
- STRENGTH(value) = le joueur possède la valeur « value » de force

Le tableau B.1 présente toutes les actions de notre architecture utilisant un planificateur GOAP.

Tableau B.1 Actions de l'architecture GOAP.

Nom	**Précondition**	**Effets**	**Coût**
Attack with grenade	HAS_GRENADE	ENEMY_DEAD	6
Attack with grenade flashed	HAS_GRENADE ∧ ENEMY_FLASHED	ENEMY_DEAD	2
Attack with grenade frozen	HAS_GRENADE ∧ ENEMY_FROZEN	ENEMY_DEAD	1
Break door	AT(pos) ∧ HAS_GRENADE	NO_LOCKED_DOOR_AT(pos)	4
Break wall	AT(pos) ∧ HAS_GRENADE	NO_WALL_AT(pos)	1
Break window	AT(pos)	NO_WINDOW_AT(pos)	1
Flash enemy	HAS_FLASHBANG	ENEMY_FLASHED ∧ SLOW_ENEMY	1
Freeze enemy	DARTGUN_READY ∧ DARTGUN_EQUIPPED	ENEMY_FROZEN ∧ SLOW_ENEMY	1
Get dart	NO_ITEM_AT(pos)	HAS_DART	1
Get dartgun	NO_ITEM_AT(pos)	HAS_DARTGUN ∧ DARTGUN_READY	1
Get flashbang	NO_ITEM_AT(pos)	HAS_FLASHBANG	1
Get grenade	NO_ITEM_AT(pos)	HAS_GRENADE	1
Get health	NO_ITEM_AT(pos)	HAS_ENOUGH_HEALTH	1
Get every items	NO_ITEM_AT(pos)	INCREASE_SCORE	0
Get key	NO_ITEM_AT(pos)	HAS_KEY	1
Get magnum	NO_ITEM_AT(pos)	HAS_MAGNUM ∧ MAGNUM_READY	1
Get magnum ammo	NO_ITEM_AT(pos)	HAS_MAGNUM_AMMO	1
Get ninja	NO_ITEM_AT(pos)	HAS_NINJA	1
Get strength	NO_ITEM_AT(pos)	STRENGTH(+1)	1
Go to	en fonction de ce qui se trouve sur la trajectoire	AT(pos)	1
Knife flashed enemy	ENEMY_FLASHED ∧ KNIFE_EQUIPPED ∧ CLOSE_ENOUGH_FROZEN	ENEMY_DEAD	2

Knife frozen enemy	ENEMY_FROZEN ∧ KNIFE_EQUIPPED	ENEMY_DEAD	1
Knife silent	MAGNUM_READY ∧ HAS_NINJA ∧ KNIFE_EQUIPPED ∧ NOT_IN_ENEMYS_SIGHT ∧ CLOSE_ENOUGH_NINJA	ENEMY_DEAD	3
Move high obstacle	AT(pos) ∧ STRENGTH(3)	NO_HIGH_OBSTACLE_AT(pos)	1
Move mid obstacle	AT(pos) ∧ STRENGTH(2)	NO_MID_OBSTACLE_AT(pos)	1
Move low obstacle	AT(pos) ∧ STRENGTH(1)	NO_LOW_OBSTACLE_AT(pos)	1
Open door	AT(pos)	NO_DOOR_AT(pos)	1
Open locked door	AT(pos) ∧ HAS_KEY	NO_LOCKED_DOORA_T(pos)	1
Pick	AT(pos)	NO_ITEM_AT(pos)	1
Reload dart gun	HAS_DARTGUN ∧ HAS_DART ∧ DART-GUN_EQUIPPED	DARTGUN_READY	1
Reload magnum	MAGNUM_EQUIPPED ∧ HAS_MAGNUM_AMMO ∧ HAS_MAGNUM	MAGNUM_READY	1
Run from grenade	AT(pos)	NOT_IN_GRENADE_RADIUS	1
Shoot magnum	MAGNUM_READY ∧ MAG-NUM_EQUIPPED	ENEMY_DEAD	6
Shoot magnum flashed	MAGNUM_READY ∧ MAGNUM_EQUIPPED ∧ ENEMY_FLASHED	ENEMY_DEAD	3
Shoot magnum frozen	MAGNUM_READY ∧ MAGNUM_EQUIPPED ∧ ENEMY_FROZEN	ENEMY_DEAD	2
Switch to dart gun	HAS_DARTGUN	DARTGUN_EQUIPPED	1
Switch to knife	aucune	KNIFE_EQUIPPED	1
Switch to magnum	HAS_MAGNUM	MAGNUM_EQUIPPED	1

Le pseudocode suivant démontre comment le but initial de l'agent est déterminé :

si le joueur est menacé par une grenade
    but = NOT_IN_GRENADE_RADIUS
sinon si le joueur est menacé par un ennemi
    si health < 50 & healthavailable
        but ← HAS_ENOUGH_HEALTH
    sinon si killwin
        but ← ENEMY_DEAD
    sinon si !flashed & !frozen
        but ← SLOW_ENEMY
    sinon
        but ← INCREASE_SCORE
sinon
    si health < 50 & healthavailable

```
            but ← HAS_ENOUGH_HEALTH
    sinon
            but ← INCREASE_SCORE
```

Les conditions utilisées dans le pseudocode sont les mêmes que celles présentées à l'annexe A.

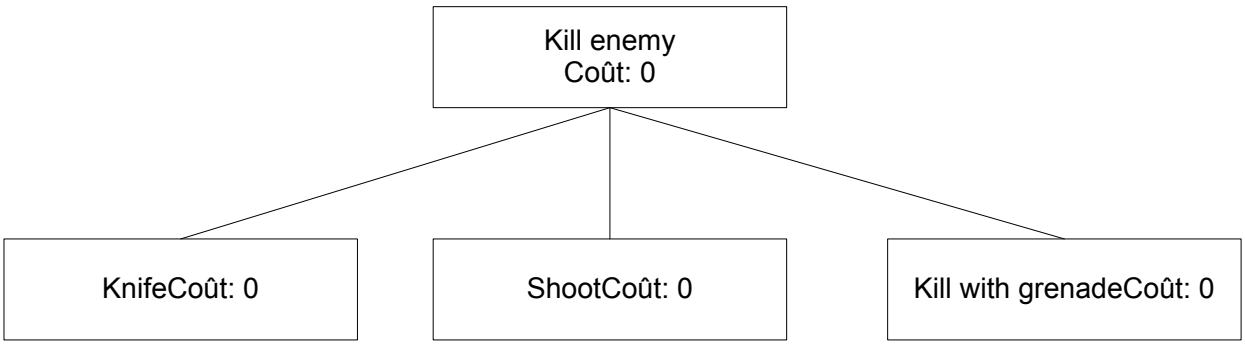
ANNEXE C

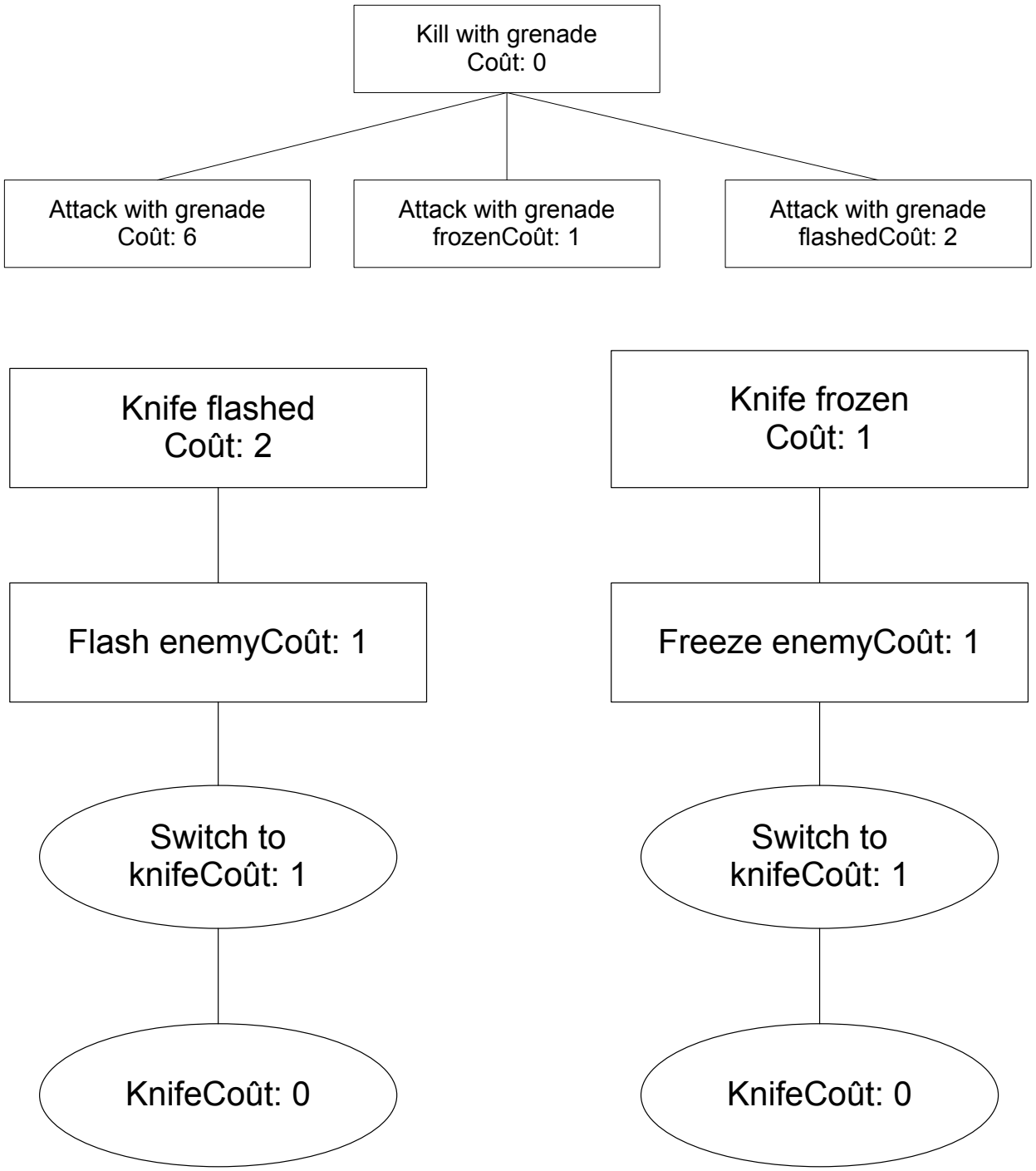
Détails de l'architecture HTN

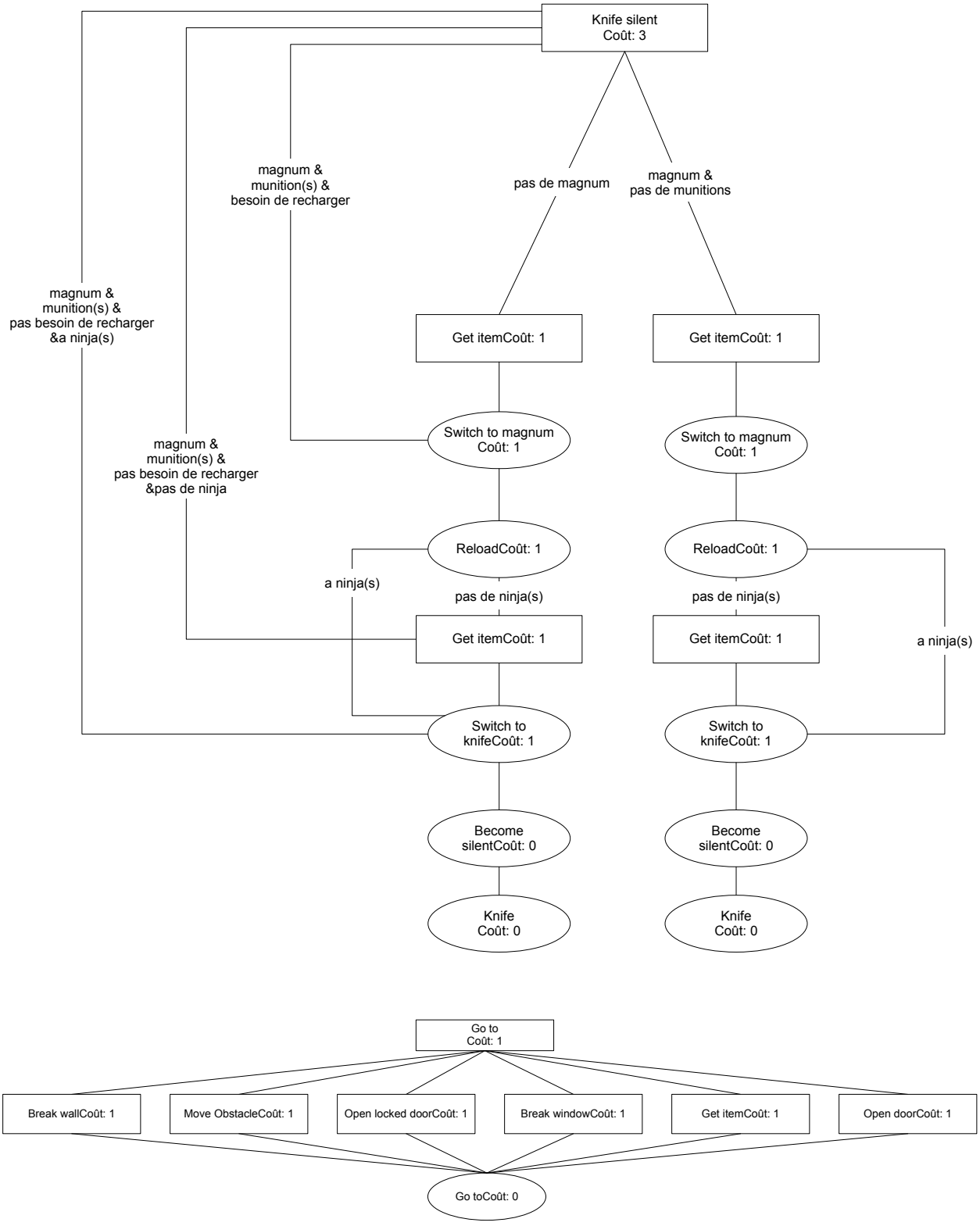
Le pseudocode suivant démontre comment la tête de l'agent est déterminé :

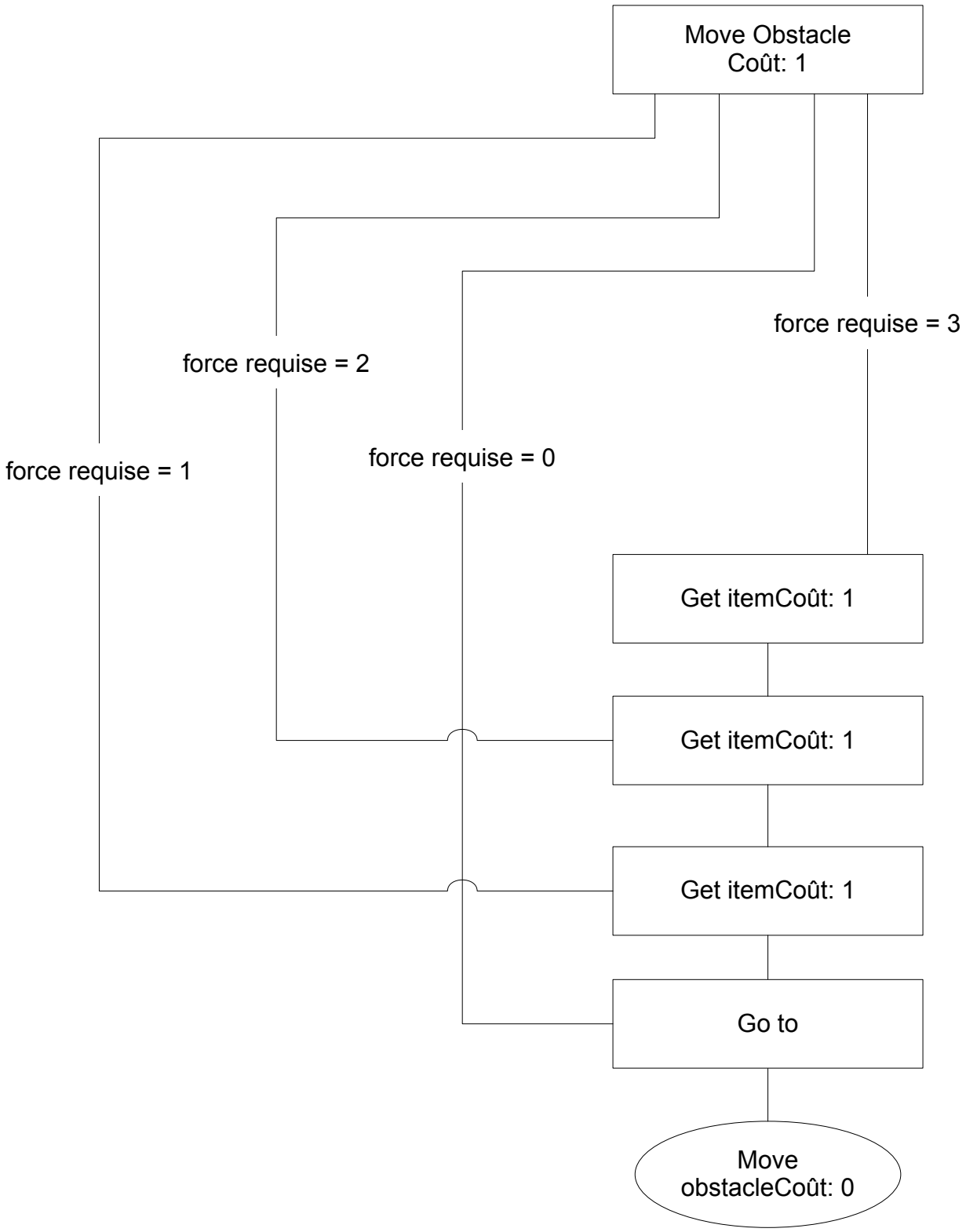
si le joueur est menacé par une grenade
    tête ← Run from grenade
sinon si le joueur est menacé par un ennemi
    si health < 50 & healthavailable
        tête ← Get health
    sinon si killwin
        tête ← Kill enemy
    sinon si !flashed & !frozen
        tête ← Slow enemy
    sinon
        tête ← Get every items
sinon
    si health < 50 & healthavailable
        tête ← Get health
    sinon
        tête ← Get every items

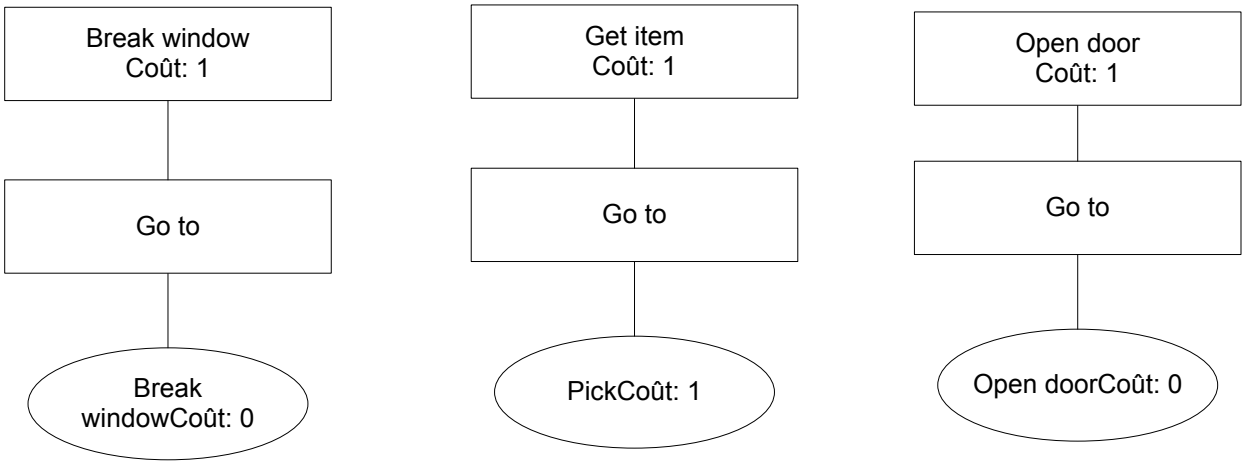
Les conditions utilisées dans le pseudocode sont les mêmes que celles présentées aux A et B. Les figures suivantes présentent le réseau de tâches que nous avons créé. Les méthodes sont représentées par des carrés alors que les opérateurs sont représentés par des ronds.

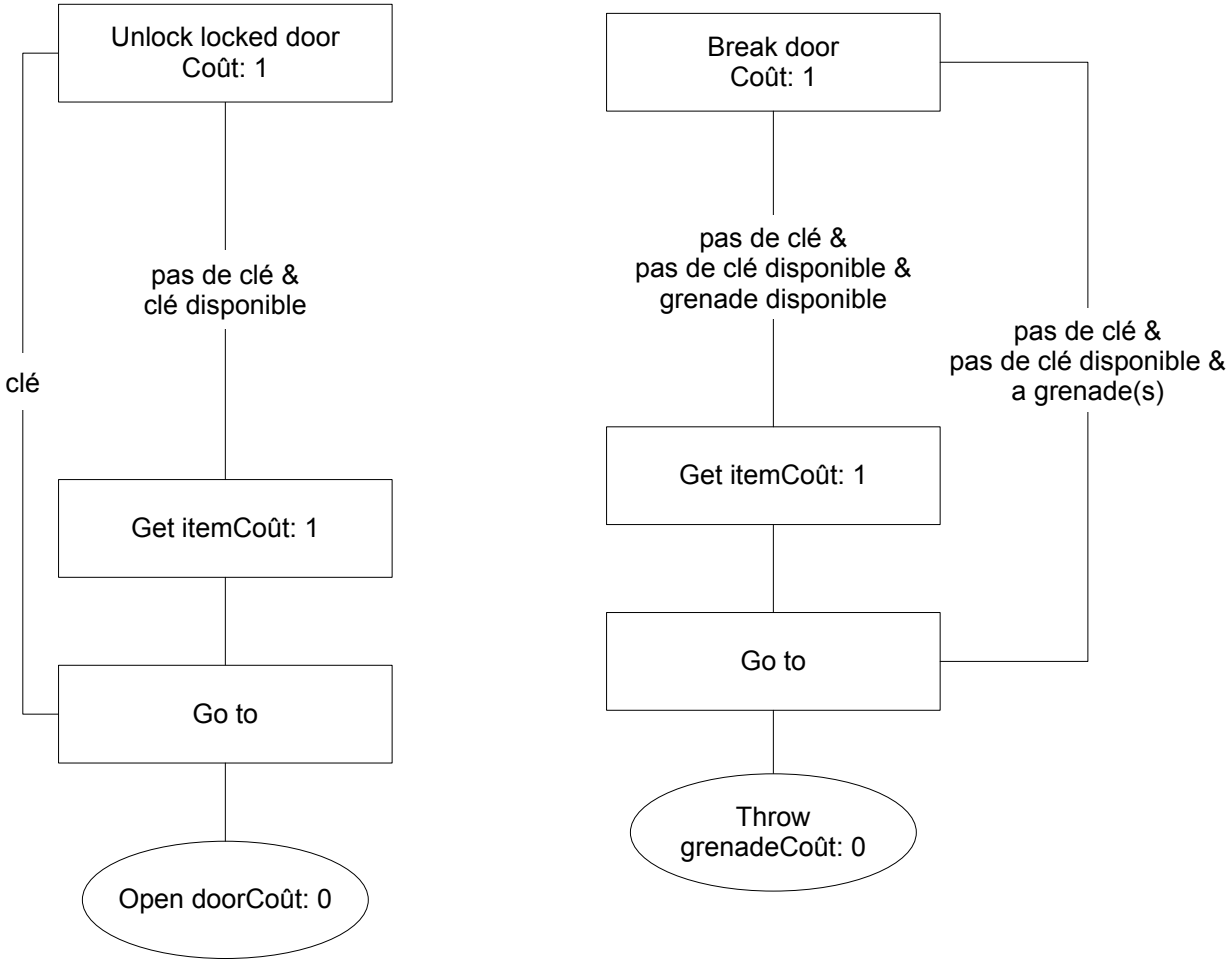


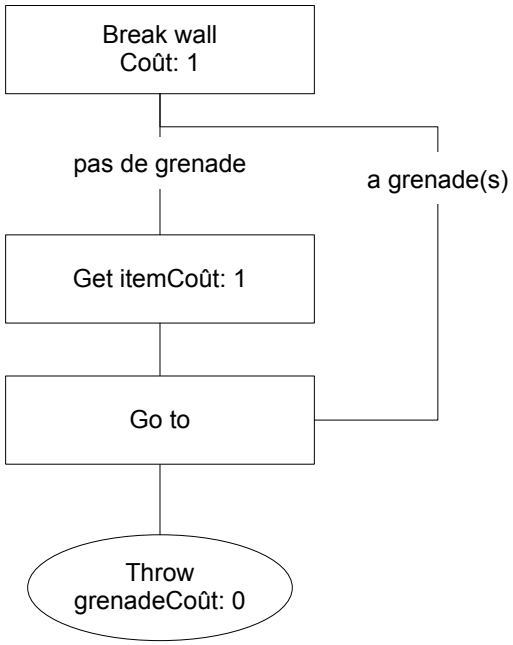

Kill with grenade
Coût: 0
Attack with grenade
Coût: 6
Attack with grenade
frozenCoût: 1
Attack with grenade
flashedCoût: 2
Knife flashed
Coût: 2
Knife frozen
Coût: 1
Flash enemyCoût: 1
Freeze enemyCoût: 1
Switch to
knifeCoût: 1
Switch to
knifeCoût: 1
KnifeCoût: 0
KnifeCoût: 0

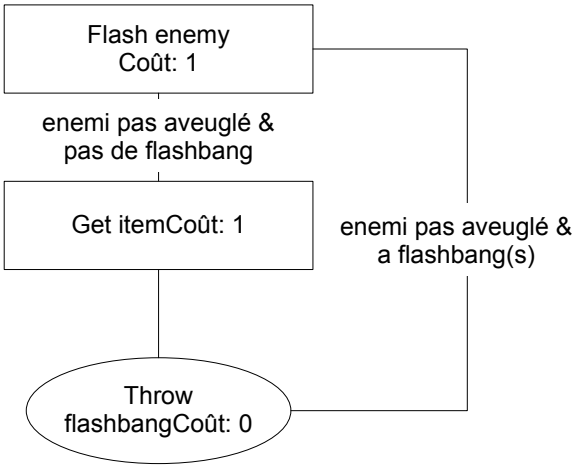

Knife silent
Coût: 3
magnum &
munition(s) &
besoin de recharger
pas de magnum
magnum &
pas de munitions
magnum &
munition(s) &
pas besoin de recharger
&a ninja(s)
Get itemCoût: 1
Get itemCoût: 1
magnum &
munition(s) &
pas besoin de recharger
&pas de ninja
Switch to magnum
Coût: 1
Switch to magnum
Coût: 1
ReloadCoût: 1
ReloadCoût: 1
a ninja(s)
pas de ninja(s)
pas de ninja(s)
Get itemCoût: 1
Get itemCoût: 1
a ninja(s)
Switch to
knifeCoût: 1
Switch to
knifeCoût: 1
Become
silentCoût: 0
Become
silentCoût: 0
Knife
Coût: 0
Knife
Coût: 0
Go to
Coût: 1
Break wallCoût: 1
Move ObstacleCoût: 1
Open locked doorCoût: 1
Break windowCoût: 1
Get itemCoût: 1
Open doorCoût: 1
Go toCoût: 0

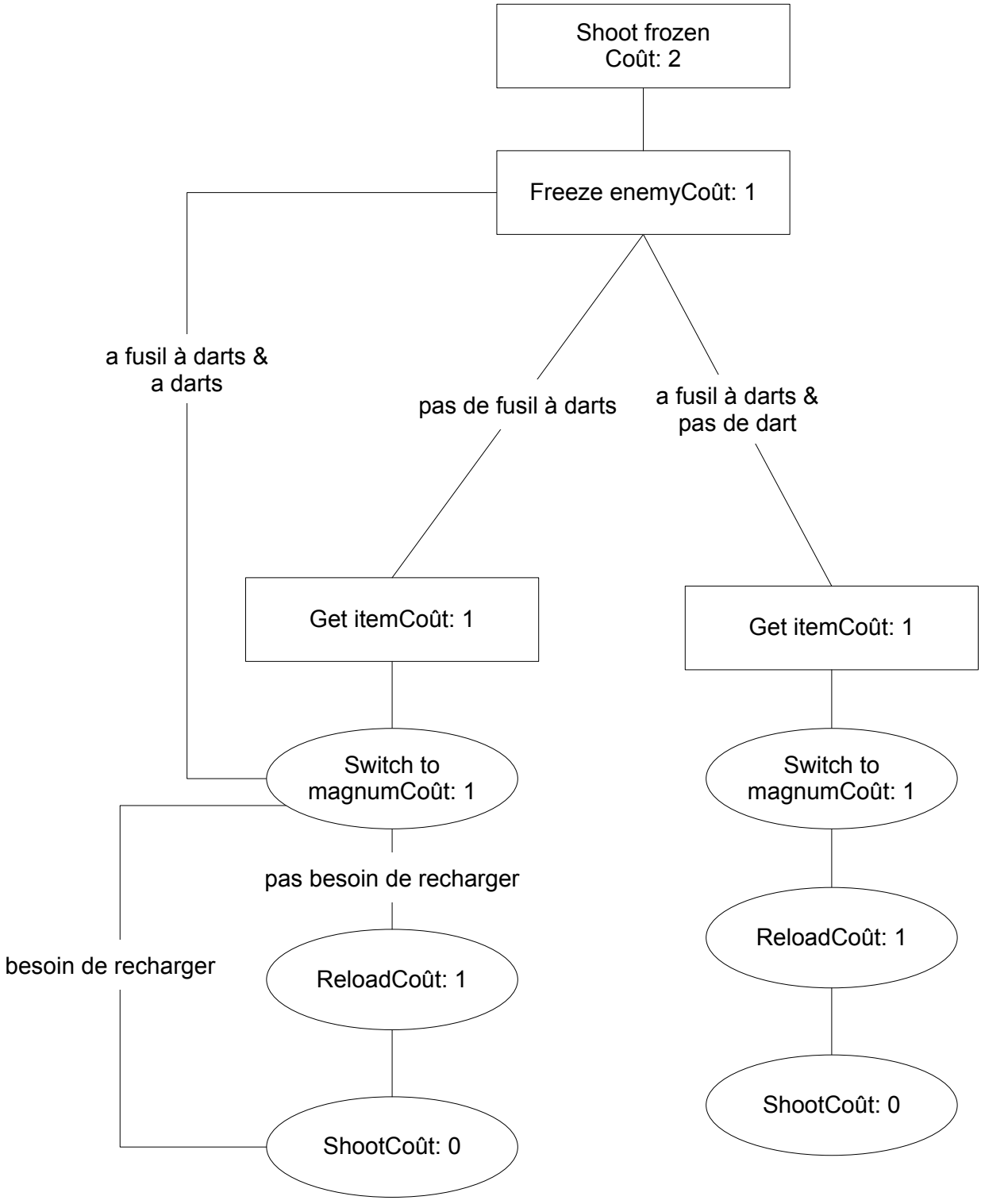

Move Obstacle
Coût: 1
force requise = 3
force requise = 2
force requise = 1
force requise = 0
Get itemCoût: 1
Get itemCoût: 1
Get itemCoût: 1
Go to
Move
obstacleCoût: 0

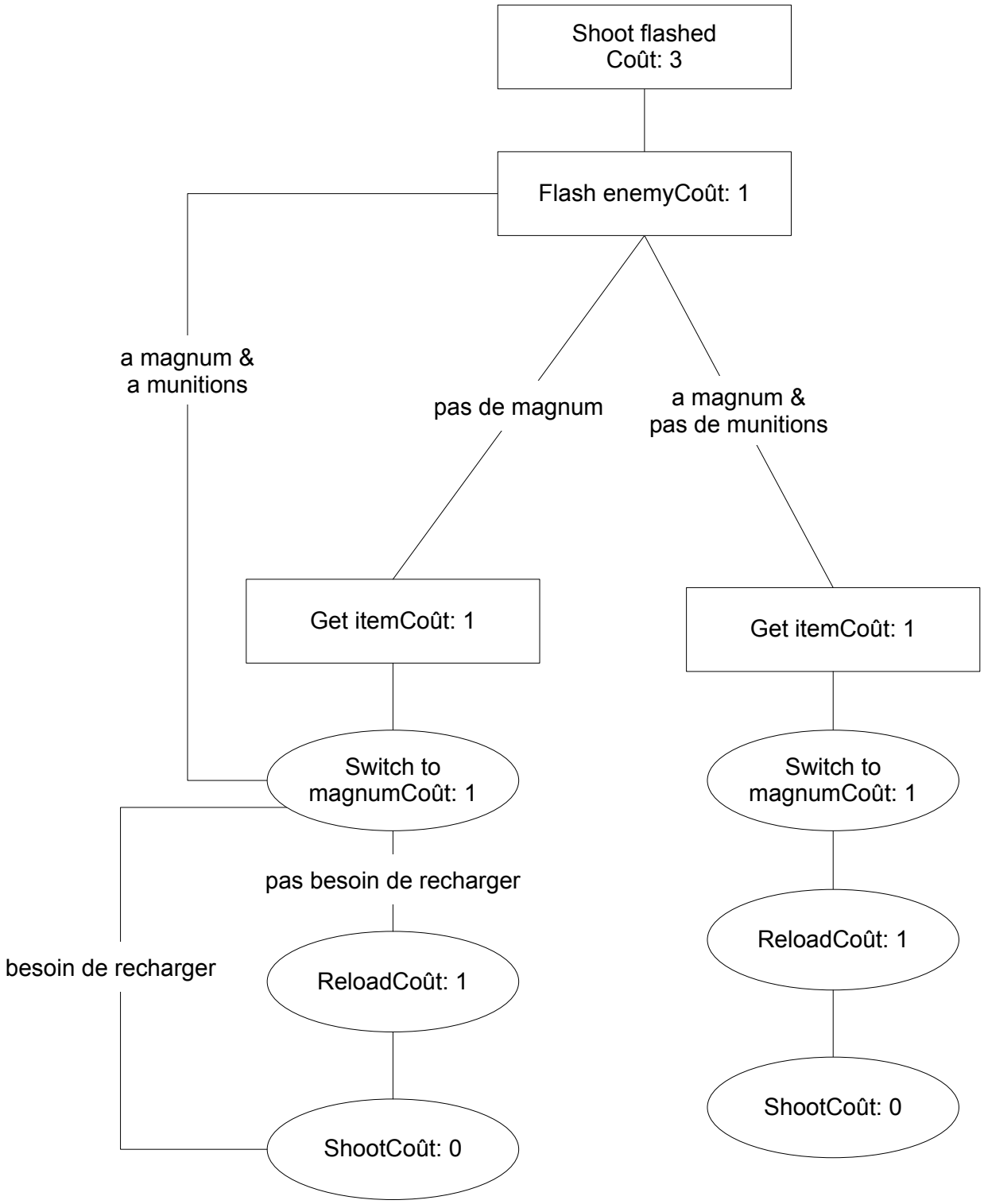

Break window
Coût: 1
Go to
Break windowCoût: 0
Get item
Coût: 1
Go to
PickCoût: 1
Open door
Coût: 1
Go to
Open doorCoût: 0

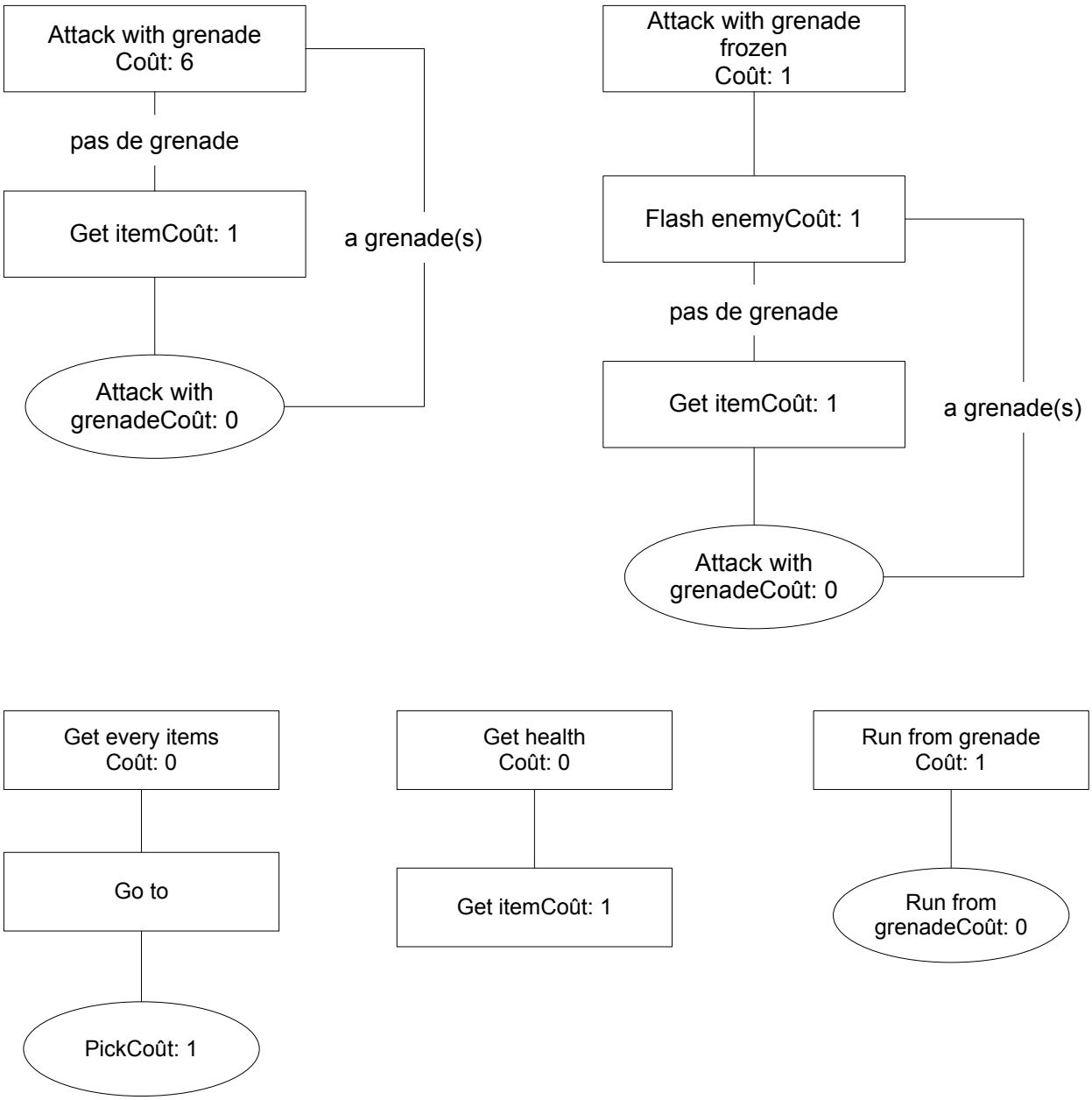

Unlock locked door
Coût: 1
pas de clé &
clé disponible
clé
Get itemCoût: 1
Go to
Open doorCoût: 0
Break door
Coût: 1
pas de clé &
pas de clé disponible &
grenade disponible
pas de clé &
pas de clé disponible &
a grenade(s)
Get itemCoût: 1
Go to
Throw grenadeCoût: 0

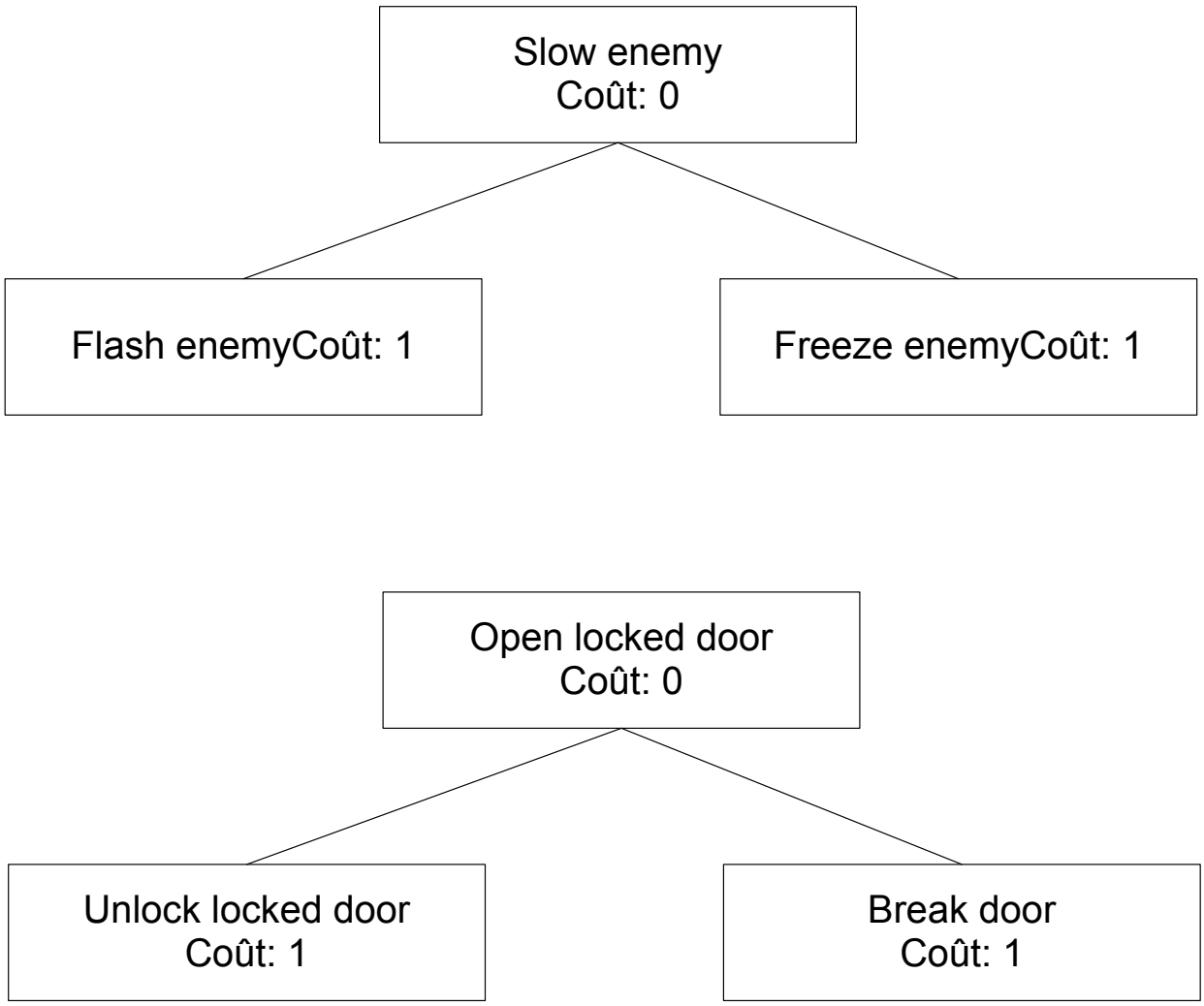

Break wall
Coût: 1
pas de grenade
a grenade(s)
Get itemCoût: 1
Go to
Throw
grenadeCoût: 0

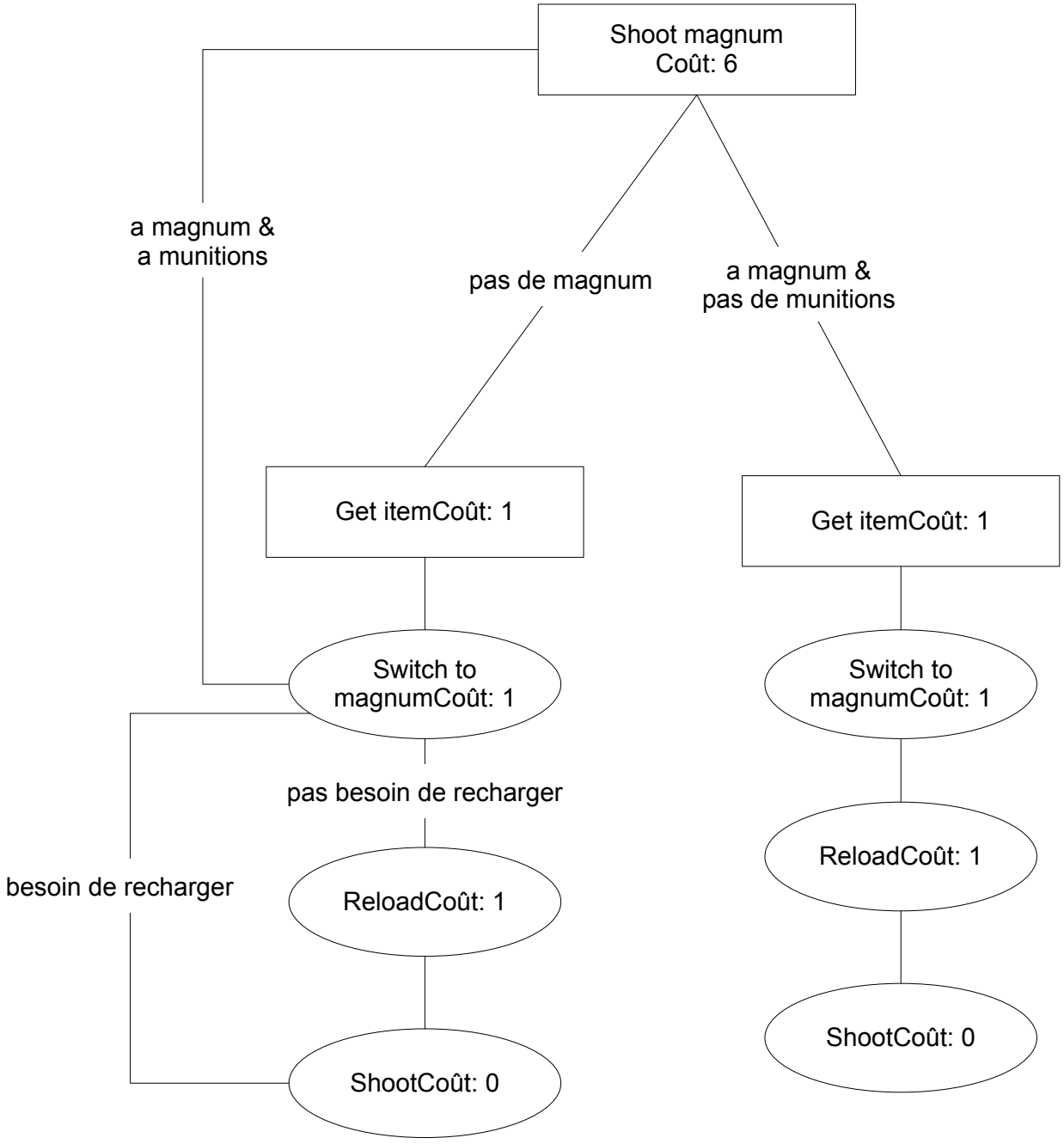

Flash enemy
Coût: 1
enemi pas aveuglé &
pas de flashbang
Get itemCoût: 1
enemi pas aveuglé &
a flashbang(s)
Throw
flashbangCoût: 0

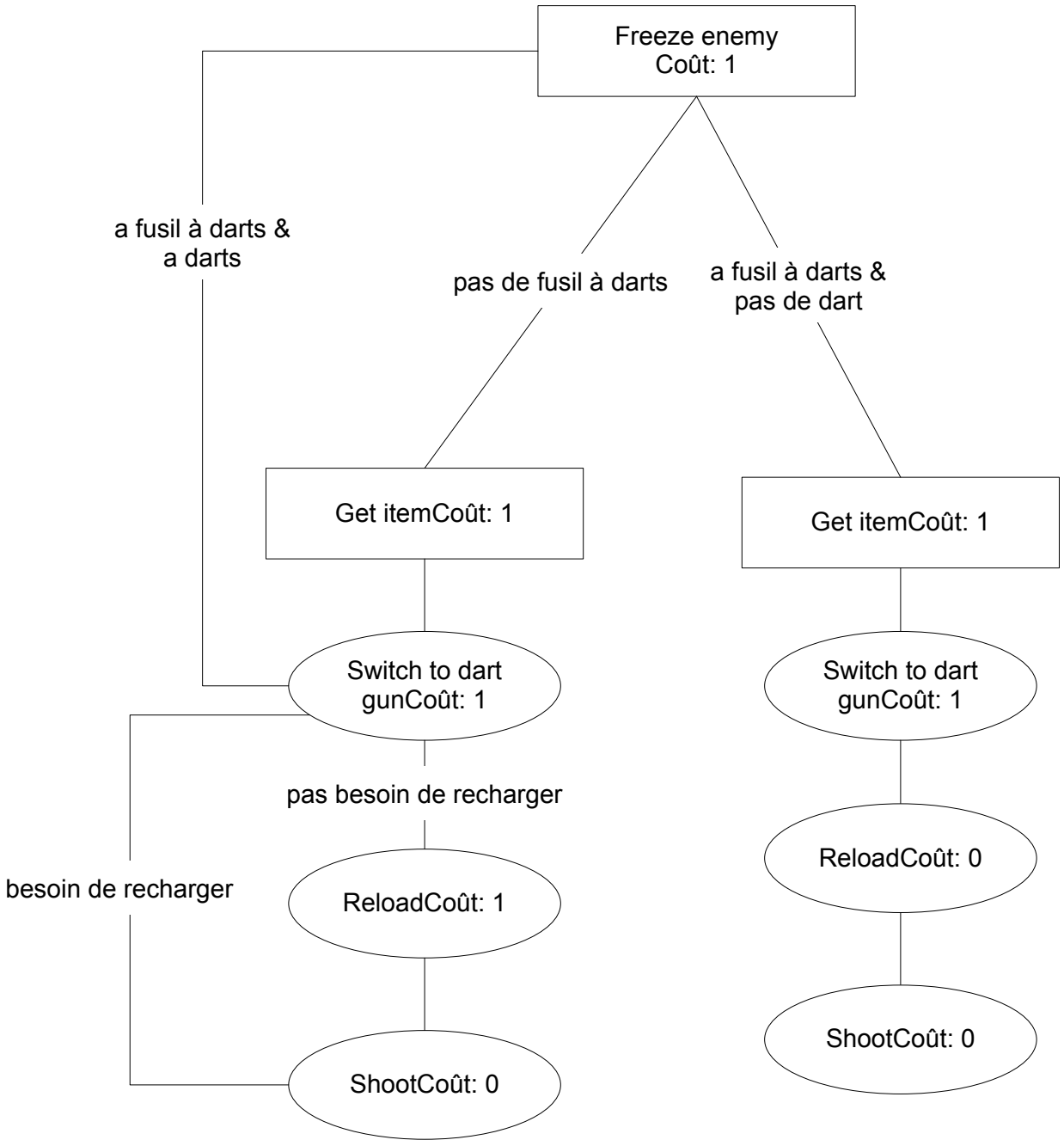

Shoot frozen
Coût: 2
Freeze enemyCoût: 1
a fusil à darts &
a darts
pas de fusil à darts
a fusil à darts &
pas de dart
Get itemCoût: 1
Get itemCoût: 1
Switch to
magnumCoût: 1
Switch to
magnumCoût: 1
pas besoin de recharger
besoin de recharger
ReloadCoût: 1
ReloadCoût: 1
ShootCoût: 0
ShootCoût: 0


Shoot flashed
Coût: 3
Flash enemyCoût: 1
a magnum &
a munitions
pas de magnum
a magnum &
pas de munitions
Get itemCoût: 1
Get itemCoût: 1
Switch to
magnumCoût: 1
Switch to
magnumCoût: 1
pas besoin de recharger
besoin de recharger
ReloadCoût: 1
ReloadCoût: 1
ShootCoût: 0
ShootCoût: 0


Attack with grenade
Coût: 6
pas de grenade
Get itemCoût: 1
a grenade(s)
Attack with
grenadeCoût: 0
Attack with grenade
frozen
Coût: 1
Flash enemyCoût: 1
pas de grenade
Get itemCoût: 1
a grenade(s)
Attack with
grenadeCoût: 0
Get every items
Coût: 0
Go to
PickCoût: 1
Get health
Coût: 0
Get itemCoût: 1
Run from grenade
Coût: 1
Run from
grenadeCoût: 0


Slow enemy
Coût: 0
Flash enemyCoût: 1
Freeze enemyCoût: 1
Open locked door
Coût: 0
Unlock locked door
Coût: 1
Break door
Coût: 1


Shoot magnum
Coût: 6
a magnum &
a munitions
pas de magnum
a magnum &
pas de munitions
Get itemCoût: 1
Get itemCoût: 1
Switch to
magnumCoût: 1
Switch to
magnumCoût: 1
pas besoin de recharger
ReloadCoût: 1
besoin de recharger
ReloadCoût: 1
ShootCoût: 0
ShootCoût: 0


Freeze enemy
Coût: 1
a fusil à darts &
a darts
pas de fusil à darts
a fusil à darts &
pas de dart
Get itemCoût: 1
Get itemCoût: 1
Switch to dart
gunCoût: 1
Switch to dart
gunCoût: 1
pas besoin de recharger
ReloadCoût: 0
besoin de recharger
ReloadCoût: 1
ShootCoût: 0
ShootCoût: 0

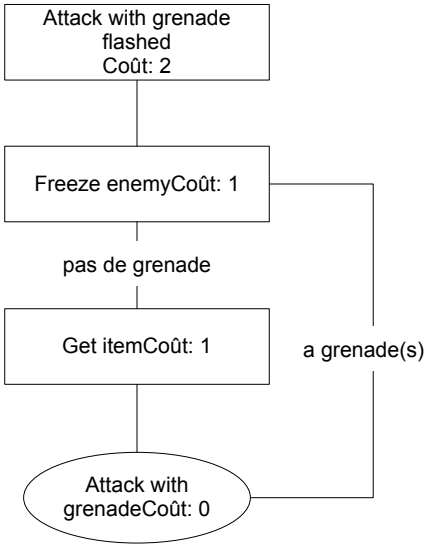


Figure C.1 Hiérarchie des méthodes et des opérateurs de l'architecture utilisant le planificateur HTN.

ANNEXE D

Correspondance entre les architectures GOAP et HTN

Le tableau D.1 présente la correspondance entre les actions de l'architecture HTN et celles de l'architectures GOAP.

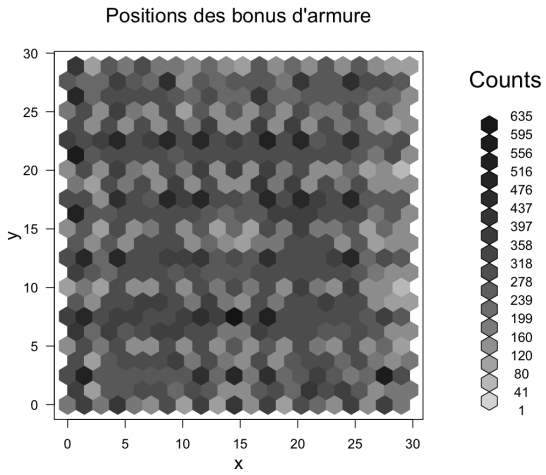
Tableau D.1 Correspondance entre les actions de l'architecture HTN et celles de l'architectures GOAP.

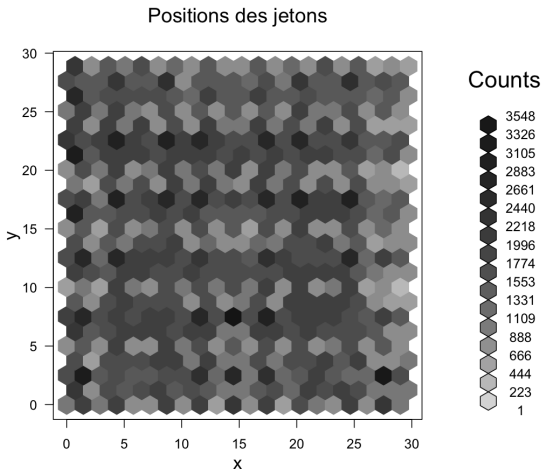
Action HTN	**Action GOAP correspondante**	**Coût**
Méthode AttackWithGrenade	AttackWithGrenade	6
Méthode AttackWithGrenade-Flashed	AttackWithGrenadeFlashed	2
Méthode AttackWithGrenade-Frozen	AttackWithGrenadeFrozen	1
Méthode BreakDoor	BreakDoor	4
Méthode BreakWall	BreakWall	1
Méthode BreakWindow	BreakWindow	1
Méthode FlashEnemy	FlashEnemy	1
Méthode FreezeEnemy	FreezeEnemy	1
Méthode GetEveryItems	GetEveryItems	0
Méthode GetHealth	GetHealth	1
Méthode GetItem	GetDart GetDartGun GetFlashbang GetGrenade GetKey GetMagnum GetMagnumAmmo GetNinja GetStrength	1
Méthode GoTo	GoTo	1
Méthode KillEnemy	Aucune	0

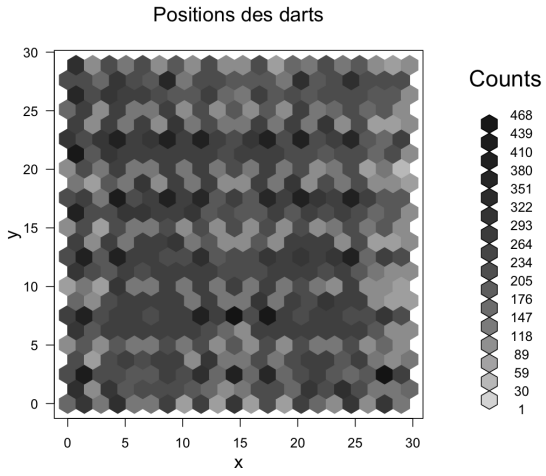
Méthode KillWithGrenade	Aucune	0
Méthode Knife	Aucune	0
Méthode KnifeFlashed	KnifeFlashedEnemy	2
Méthode KnifeFrozen	KnifeFrozenEnemy	1
Méthode KnifeSilent	KnifeSilent	3
Méthode MoveObstacle	MoveHighObstacle MoveLowObstacle MoveMidObstacle	1
Méthode OpenDoor	OpenDoor	1
Méthode OpenLockedDoor	Aucune	0
Méthode RunFromGrenade	RunFromGrenade	1
Méthode Shoot	Aucune	0
Méthode ShootFlashed	ShootMagnumFlashed	3
Méthode ShootFrozen	ShootMagnumFrozen	2
Méthode ShootMagnum	ShootMagnum	6
Méthode SlowEnemy	Aucune	0
Méthode UnlockLockedDoor	OpenLockedDoor	1
Opérateur AttackWithGrenade	Aucune	0
Opérateur BreakWindow	Aucune	0
Opérateur GoTo	Aucune	0
Opérateur Knife	Aucune	0
Opérateur BecomeSilent	Aucune	0
Opérateur MoveObstacle	Aucune	0
Opérateur OpenDoor	Aucune	0
Opérateur Pick	Pick	1
Opérateur Reload	ReloadDartGun ReloadMagnum	1
Opérateur RunFromGrenade	Aucune	0
Opérateur Shoot	Aucune	0
Opérateur SwitchToDartGun	SwitchToDartGun	1
Opérateur SwitchToKnife	SwitchToKnife	1
Opérateur SwitchToMagnum	SwitchToMagnum	1
Opérateur ThrowFlashbang	Aucune	0
Opérateur ThrowGrenade	Aucune	0

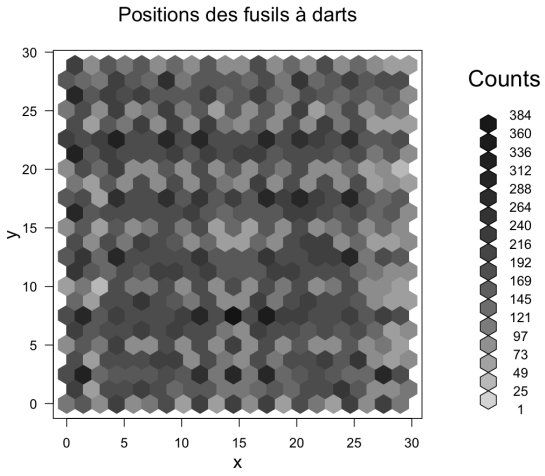
ANNEXE E

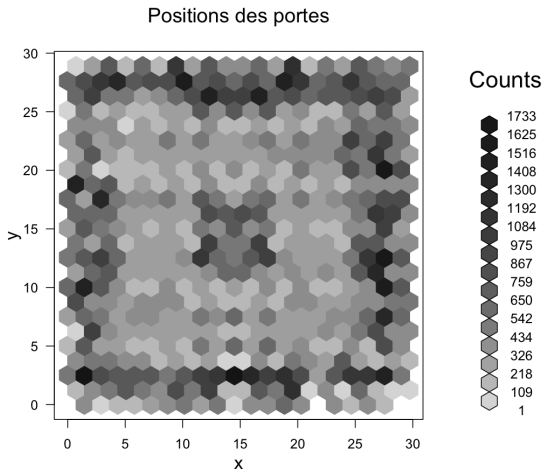
Statistiques sur les entités de 50 000 niveaux générés aléatoirement

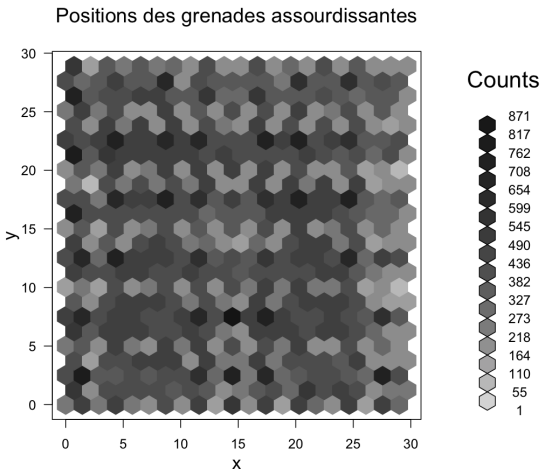

Positions des bonus d'armure
Counts
635
595
556
516
476
437
397
358
318
278
239
199
160
120
80
41
1
x
y
0
5
10
15
20
25
30

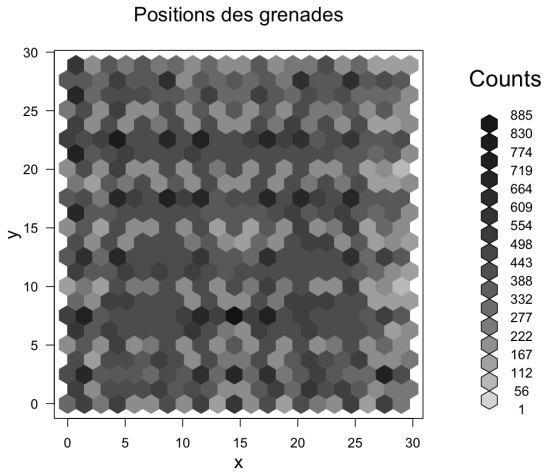

Positions des jetons
Counts
3548
3326
3105
2883
2661
2440
2218
1996
1774
1553
1331
1109
888
666
444
223
1
x
y
0
5
10
15
20
25
30

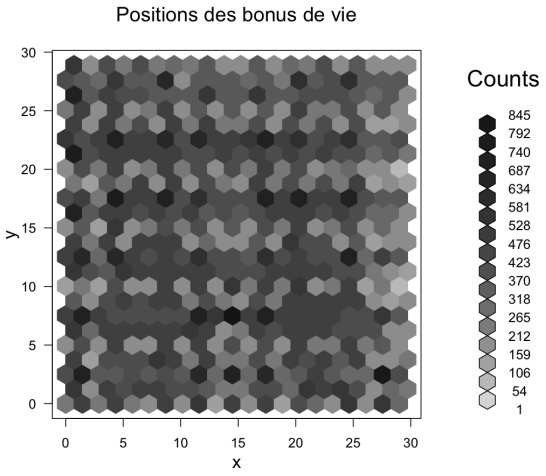

Positions des darts
Counts
468
439
410
380
351
322
293
264
234
205
176
147
118
89
59
30
1
x
y
0
5
10
15
20
25
30

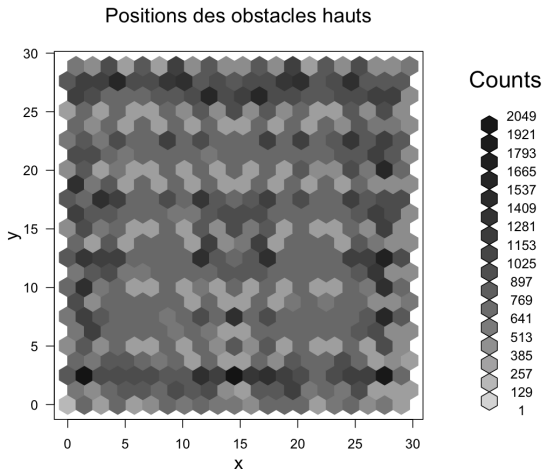

Positions des fusils à darts
Counts
384
360
336
312
288
264
240
216
192
169
145
121
97
73
49
25
1
x
y
0
5
10
15
20
25
30

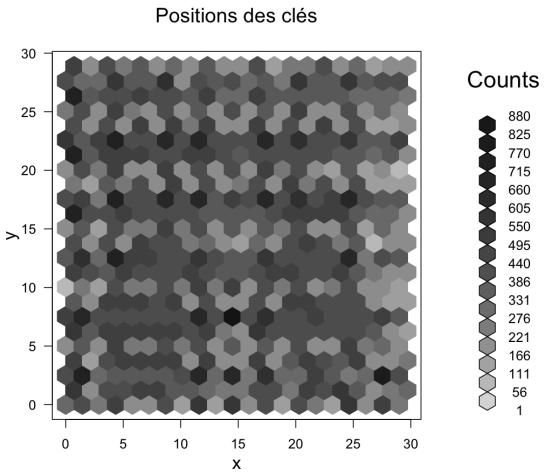

Positions des portes
Counts
1733
1625
1516
1408
1300
1192
1084
975
867
759
650
542
434
326
218
109
1
y
x
0
5
10
15
20
25
30

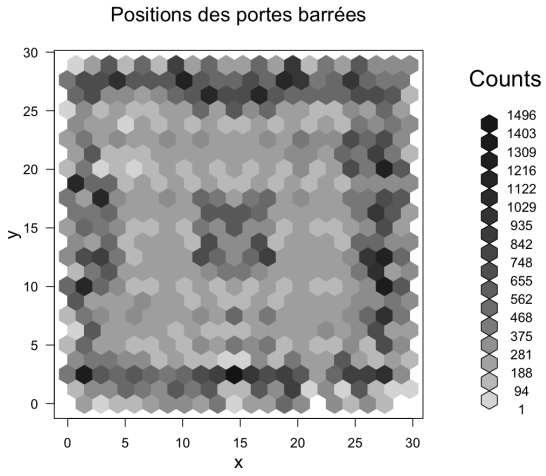

Positions des grenades assourdissantes
Counts
871
817
762
708
654
599
545
490
436
382
327
273
218
164
110
55
1
y
x
0
5
10
15
20
25
30

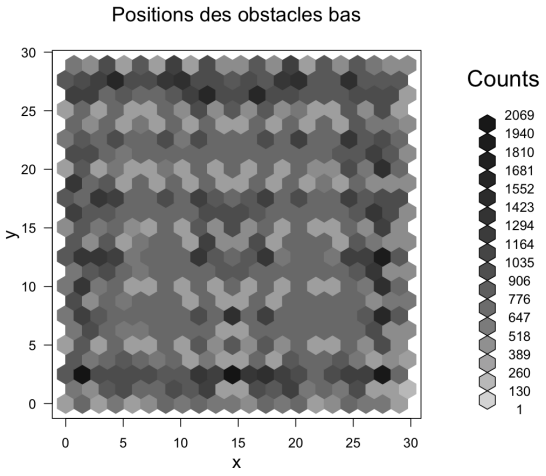

Positions des grenades
Counts
885
830
774
719
664
609
554
498
443
388
332
277
222
167
112
56
1
y
x
0
5
10
15
20
25
30

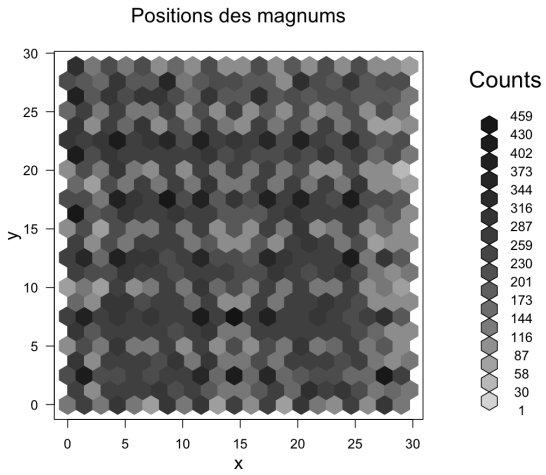

Positions des bonus de vie
Counts
845
792
740
687
634
581
528
476
423
370
318
265
212
159
106
54
1
y
x
0
5
10
15
20
25
30

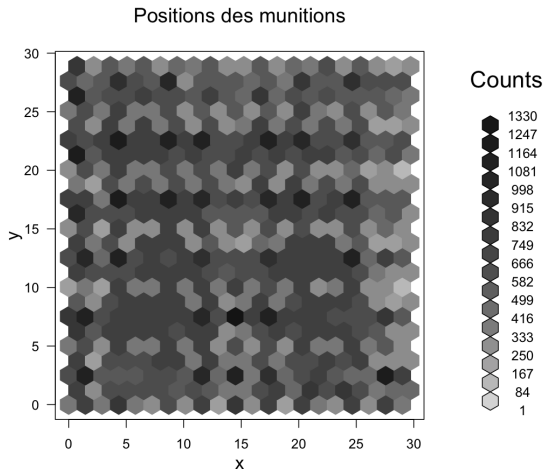

Positions des obstacles hauts
Counts
2049
1921
1793
1665
1537
1409
1281
1153
1025
897
769
641
513
385
257
129
1
x
y
0
5
10
15
20
25
30

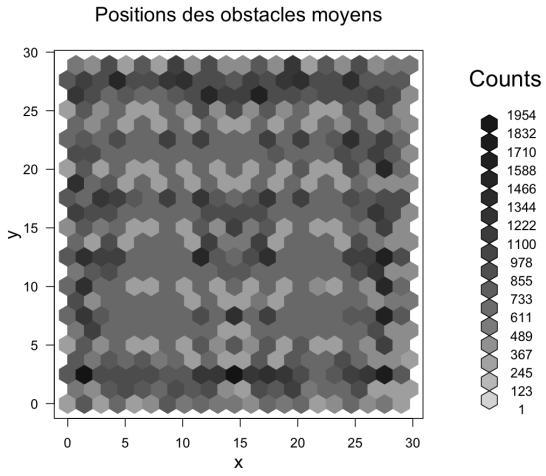

Positions des clés
Counts
880
825
770
715
660
605
550
495
440
386
331
276
221
166
111
56
1
x
y
0
5
10
15
20
25
30

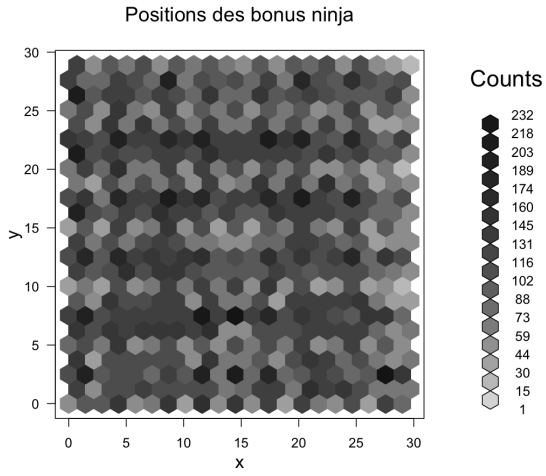

Positions des portes barrées
Counts
1496
1403
1309
1216
1122
1029
935
842
748
655
562
468
375
281
188
94
1
x
y
0
5
10
15
20
25
30


Positions des obstacles bas
Counts
2069
1940
1810
1681
1552
1423
1294
1164
1035
906
776
647
518
389
260
130
1
x
y
0
5
10
15
20
25
30


Positions des magnums
Counts
459
430
402
373
344
316
287
259
230
201
173
144
116
87
58
30
1
x
y
0
5
10
15
20
25
30


Positions des munitions
Counts
1330
1247
1164
1081
998
915
832
749
666
582
499
416
333
250
167
84
1
x
y
0
5
10
15
20
25
30


Positions des obstacles moyens
Counts
1954
1832
1710
1588
1466
1344
1222
1100
978
855
733
611
489
367
245
123
1
x
y
0
5
10
15
20
25
30


Positions des bonus ninja
Counts
232
218
203
189
174
160
145
131
116
102
88
73
59
44
30
15
1
x
y
0
5
10
15
20
25
30

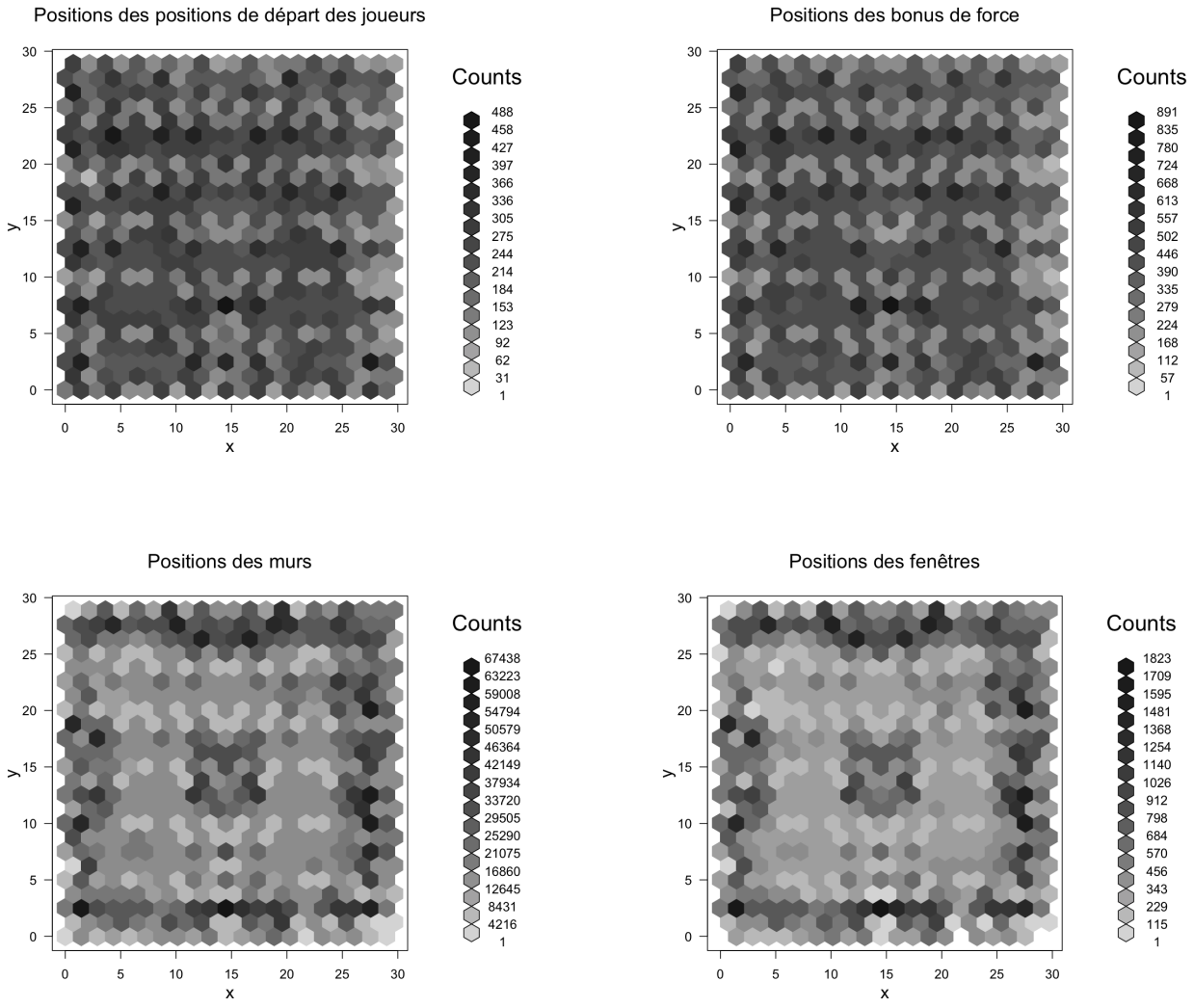
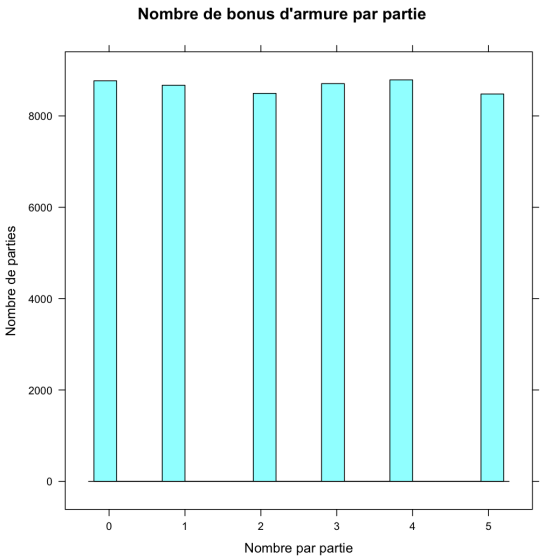
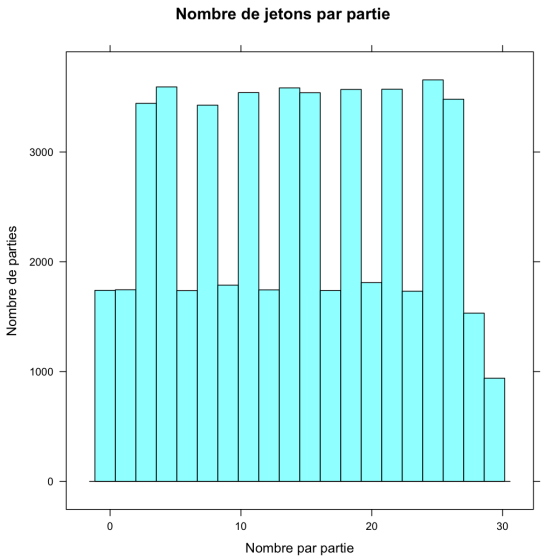
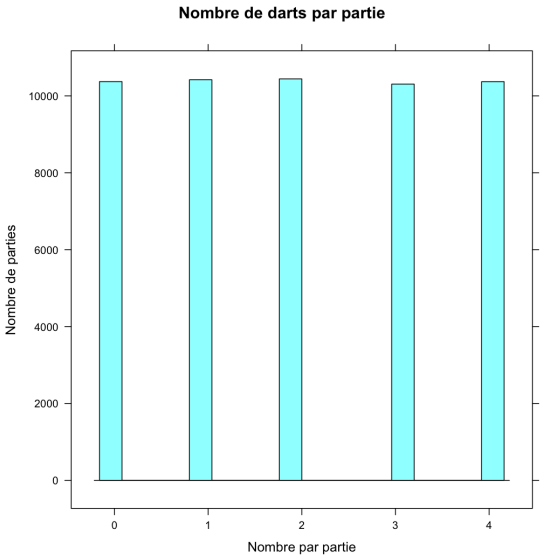
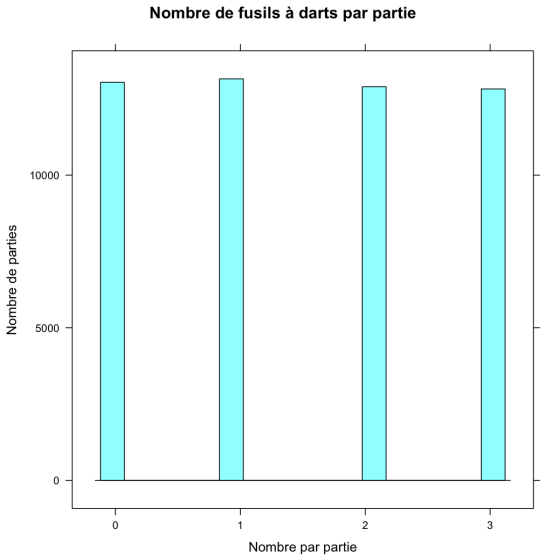


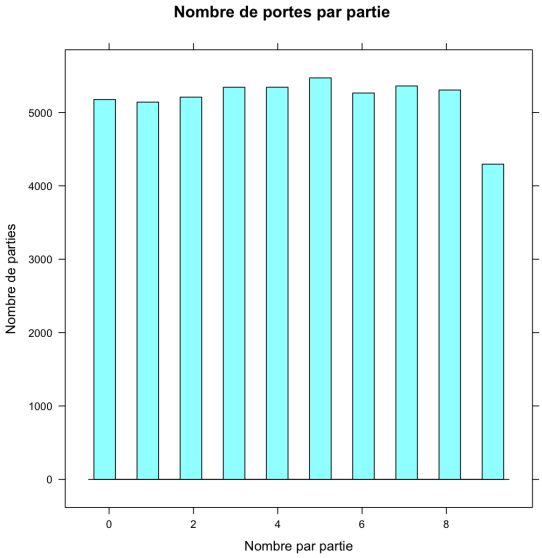
Figure E.1 Positions des entités pour environ 50 000 niveaux générés aléatoirement.

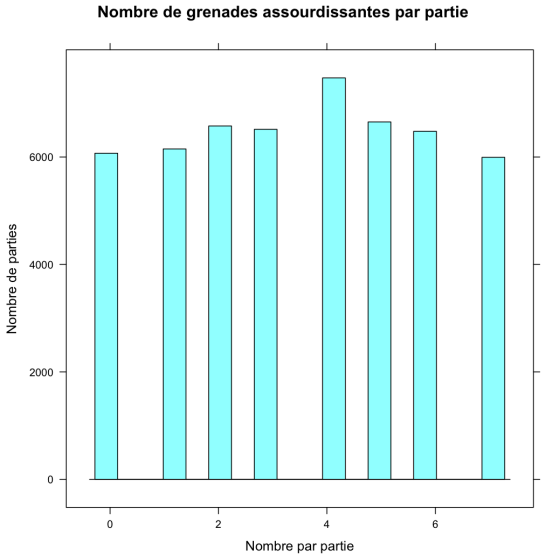

Nombre de bonus d'armure par partie
Nombre de parties
0
2000
4000
6000
8000
0
1
2
3
4
5
Nombre par partie

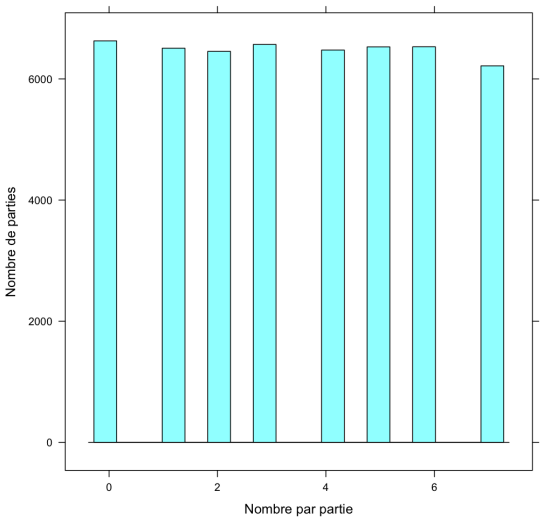

Nombre de jetons par partie
Nombre de parties
0
1000
2000
3000
0
10
20
30
Nombre par partie

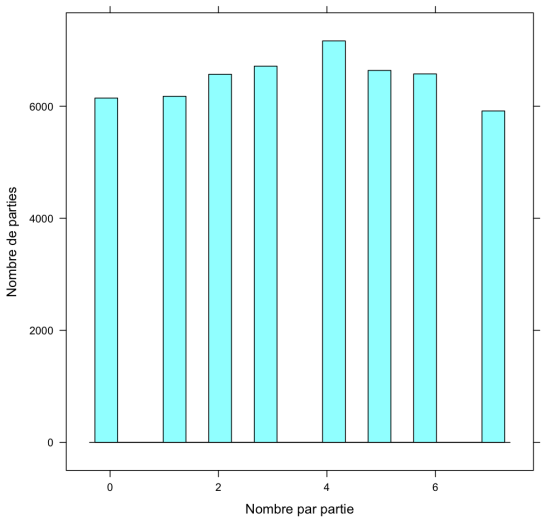

Nombre de darts par partie
Nombre de parties
0
2000
4000
6000
8000
10000
0
1
2
3
4
Nombre par partie

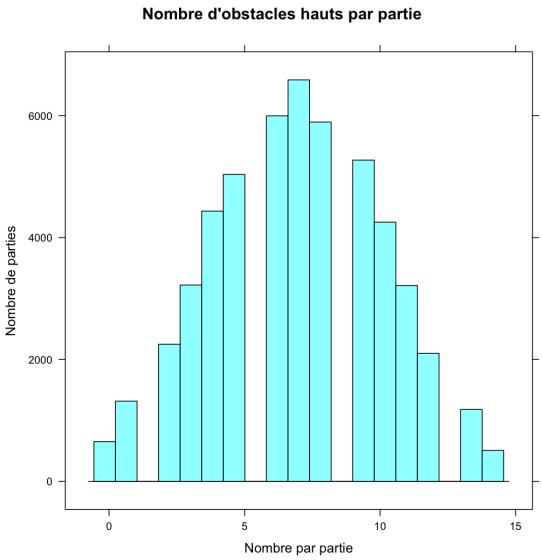

Nombre de fusils à darts par partie
Nombre de parties
0
5000
10000
0
1
2
3
Nombre par partie

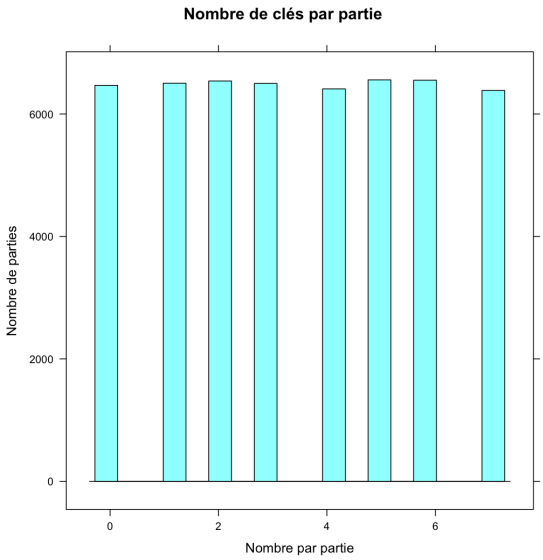

Nombre de portes par partie
Nombre de parties
0
1000
2000
3000
4000
5000
0
2
4
6
8
Nombre par partie

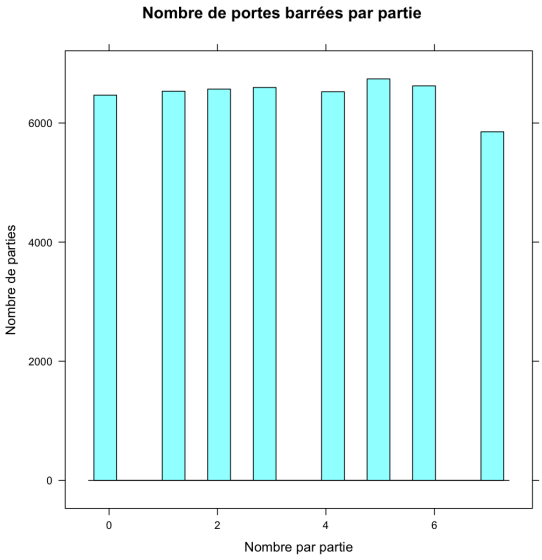

Nombre de grenades assourdissantes par partie
Nombre de parties
0
2000
4000
6000
0
2
4
6
Nombre par partie

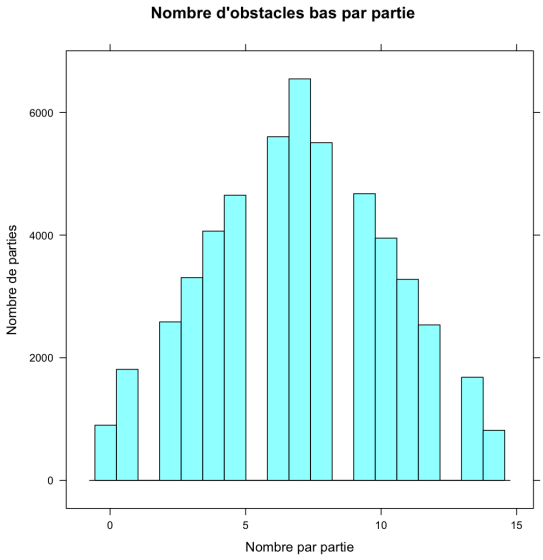

Nombre de grenades par partie
Nombre de parties
0
2000
4000
6000
0
2
4
6
Nombre par partie

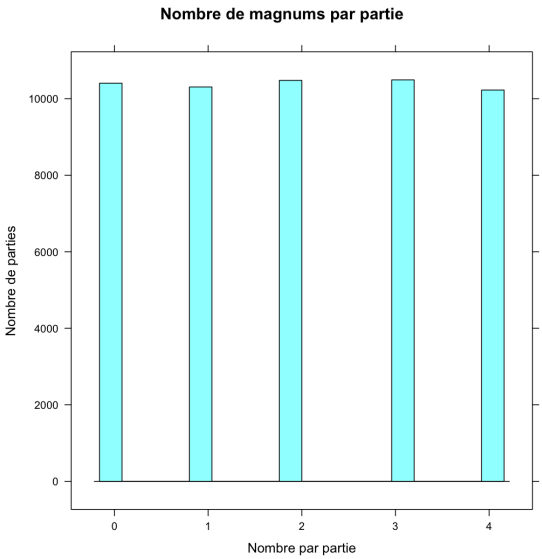

Nombre de bonus de vie par partie
Nombre de parties
0
2000
4000
6000
0
2
4
6
Nombre par partie

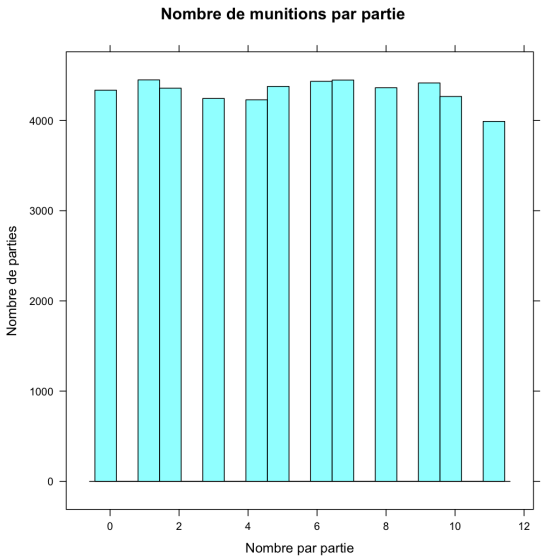

Nombre d'obstacles hauts par partie
Nombre de parties
0
2000
4000
6000
0
5
10
15
Nombre par partie

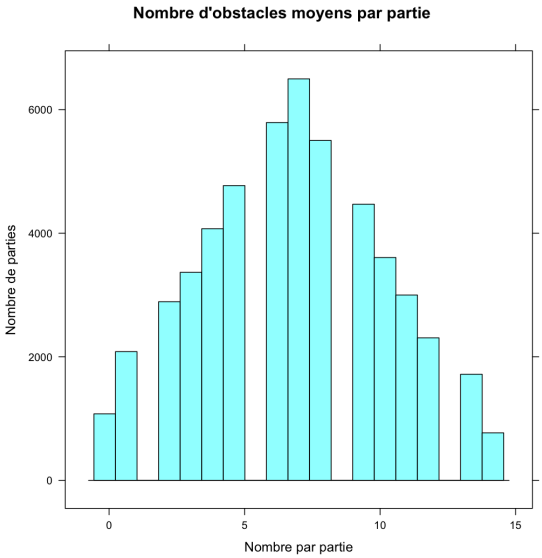

Nombre de clés par partie
Nombre de parties
0
2000
4000
6000
0
2
4
6
Nombre par partie

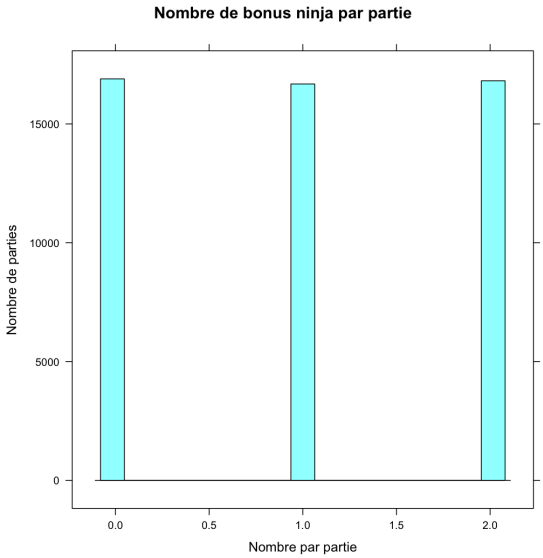

Nombre de portes barrées par partie
Nombre de parties
0
2000
4000
6000
0
2
4
6
Nombre par partie


Nombre d'obstacles bas par partie
Nombre de parties
0
2000
4000
6000
0
5
10
15
Nombre par partie


Nombre de magnums par partie
Nombre de parties
Nombre par partie
0
2000
4000
6000
8000
10000
0
1
2
3
4


Nombre de munitions par partie
Nombre de parties
Nombre par partie
0
1000
2000
3000
4000
0
2
4
6
8
10
12


Nombre d'obstacles moyens par partie
Nombre de parties
Nombre par partie
0
2000
4000
6000
0
5
10
15


Nombre de bonus ninja par partie
Nombre de parties
Nombre par partie
0
5000
10000
15000
0.0
0.5
1.0
1.5
2.0

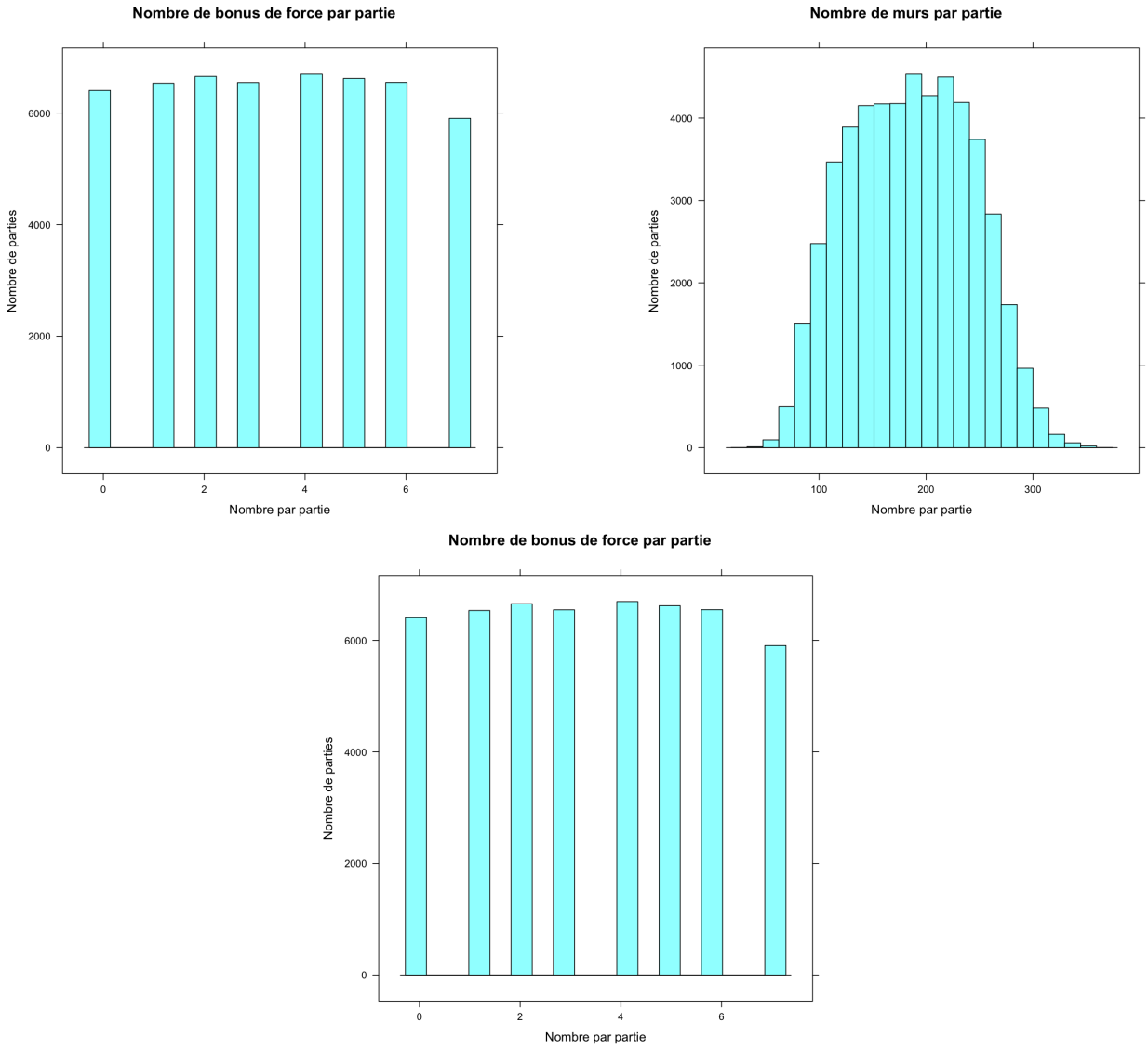


Figure E.2 Nombre d'entités par partie pour environ 50 000 niveaux générés aléatoirement.