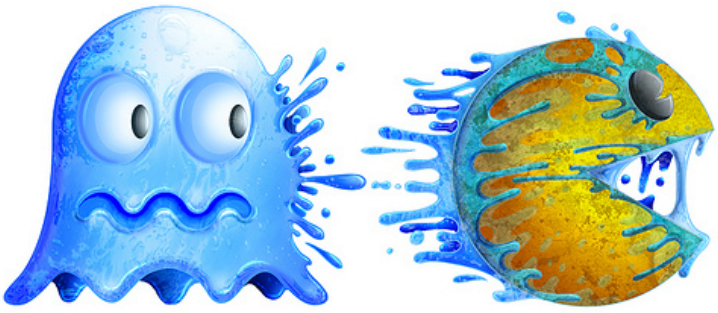
Achibet Merwan
Marchand de Kerchove Florent

Intelligence artificelle dans les jeux vidéos



Licence 2 Mathématiques-Informatique
UFR Sciences et Techniques, Université du Havre
Année universitaire 2008–2009

Table des matières

Illustration de Steven Lefcourt

Introduction

En recherche académique, le terme « intelligence artificielle » (IA) désigne le domaine d'études des agents intelligents. Le degré d'intelligence recherché varie selon les auteurs, mais le but premier de l'IA est de concevoir une machine au moins aussi intelligente qu'un humain, c'est à dire apte à résoudre les mêmes problèmes. C'est en ce sens que le terme d'IA est utilisé dans les jeux vidéos : l'intelligence artificielle dans les jeux vidéos, c'est la branche qui conçoit les agents intelligents qui peuplent les jeux.

Malheureusement, comme l'IA académique n'a pas encore atteint son but originel d'IA « forte », les agents intelligents des jeux vidéos ne sont pas plus éclairés qu'un concombre. La différence cependant, c'est que dans un jeu, l'illusion d'intelligence suffit. Si le joueur a l'impression que l'agent réagit de façon humaine, alors le jeu sera plus immersif, et l'expérience du joueur s'en trouvera meilleure [1].

Il faut garder à l'esprit que dans un jeu vidéo, tout élément est soumis à un budget. Il y a évidemment les contraintes monétaires liées au coût de développement du produit, mais également les contraintes matérielles, qui dépendent des machines des utilisateurs, mais le facteur le plus contraignant est sans doute le temps. La plupart des jeux vidéos doivent s'exécuter à plusieurs dizaines d'images par secondes, et par conséquent les agents ont moins d'un vingtième de seconde pour se mettre à jour.

Enfin, comme les avancées technologiques de ce domaine ont souvent lieu dans des entreprises commerciales, il est difficile d'obtenir des détails sur les algorithmes employés, qui comportent des améliorations spécifiques au jeu dans lequel ils sont employés. Les algorithmes les mieux documentés sont donc ceux qui datent le plus. Heureusement pour nous, ce sont aussi les plus employés, à quelques modifications près.

Nous présentons ici deux types de techniques : les machines à états pour le comportement, et les algorithmes de recherche de chemin pour les déplacements.

1 Machines à états

1.1 Machine à états simple

Les machines à états (ou FSM, pour Finite State Machines) représentent l'une des techniques les plus couramment utilisées dans le domaine du jeu vidéo pour simuler une intelligence réaliste. Dans un jeu, chaque entité gérée par une intelligence artificielle est potentiellement une machine à états (ennemi, allié, animal, objet).

Chaque machine est composée d'un nombre fini d'états (ou modèles comportementaux), chacun possédant des caractéristiques propres (direction du déplacement d'un personnage, vitesse, posture, animation...) permettant de répondre de manière cohérente à toute situation rencontrée. Chaque état est relié à au moins un autre état par une transition associée à des conditions. A chaque cycle de jeu, on vérifie pour chaque transition de l'état actuel si les conditions sont vérifiées, si oui, on passe à l'état relié par cette transition, sinon on reste sur le même état.

Pac-Man (Namco, 1980) est un exemple de jeu connu utilisant des machines à états basiques. Rapide rappel des règles du jeu : Pac-Man doit récupérer toutes les pac-gommes, ces minuscules pastilles jaunes parsemées sur le plateau de jeu, tout en évitant de croiser

[1] Nous considérons ici, en idéalistes, que la qualité d'un jeu se mesure au plaisir promulgué aux joueurs, et non au nombre de boîtes vendues. Notons cependant que l'un entraîne souvent l'autre.

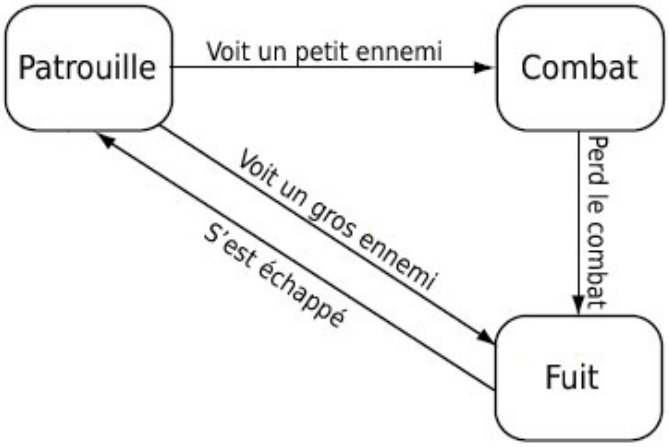


FIG. 1 – Cycle d'action simplifié d'un soldat de jeu de stratégie. (Millington, 2006)

le chemin des fantômes. Pac-Man n'est pas totalement sans défense car lorsqu'il mange une grosse pac-gomme (au nombre de quatre dans le niveau) les fantômes sont à sa merci pendant quelques secondes et disparaissent au moindre contact.

Les fantômes possèdent trois états (« recherche », « poursuite » et « fuite ») (Bourg et Seemann, 2004) déterminés par des transitions conditionnées par des tests booléens. Quand un fantôme est en état « recherche », il parcourt le niveau jusqu'à repérer Pac-Man puis bascule directement en mode « poursuite », dans lequel il se dirige vers le joueur par le plus court chemin possible afin de le pourchasser. A l'inverse, en mode « fuite » (déclenché quand Pac-Man mange une grosse pac-gomme), il va dans la direction opposée à la position actuelle du joueur et son apparence change. Malgré cette intelligence artificielle des plus sommaires, Pac-Man est, et a été, l'un des jeux les plus joués au monde, que ce soit sur borne d'arcade lors de sa sortie ou de nos jours sur téléphone portable.

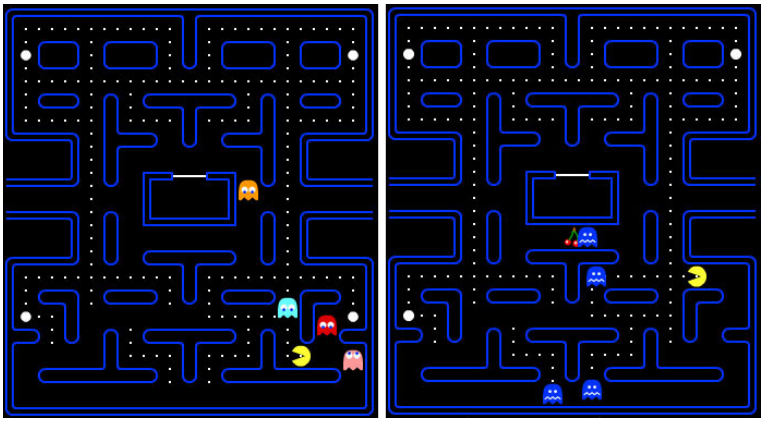


FIG. 2 – Les fantômes sont en état « poursuite », puis « fuite ». (`webpacman.com`)

Chaque machine à états est programmée à l'avance et ne pourra jamais sortir du cadre des actions prévues par le programmeur. Et contrairement à une intelligence artificielle plus évoluée gérée par des techniques telles que les réseaux de neurones, aucun comportement inattendu ne pourra émerger, néanmoins la plupart du temps des machines à états bien conçues suffiront à provoquer des réactions logiques et donc à immerger le joueur dans un univers cohérent et fidèle à la réalité.

1.2 Machine à états hiérarchisée

Des machines à états plus complexes peuvent être créées dans le but d'introduire une plus grande variété de comportements mais dans certains cas, notamment lors d'événements dits d'alarme (Millington, 2006), l'ajout de nouveaux états peut engendrer des répétitions et des lourdeurs dans la structure d'une FSM.

Réutilisons l'exemple de la première image, illustrant le cycle d'activité d'un soldat de jeu de stratégie. Imaginons qu'une nouvelle version de ce soldat ait été inventée : désormais quand le soldat entendra le signal d'alerte provenant du clocher de son village, il abandonnera son activité actuelle et courra défendre son village. L'audition du signal d'alarme par le soldat est un événement d'alerte puisqu'elle stoppera toute autre action en cours, la défense du village étant une priorité. En utilisant une machine à états simple, on pourrait implémenter cette nouvelle fonction comme sur la figure 3.

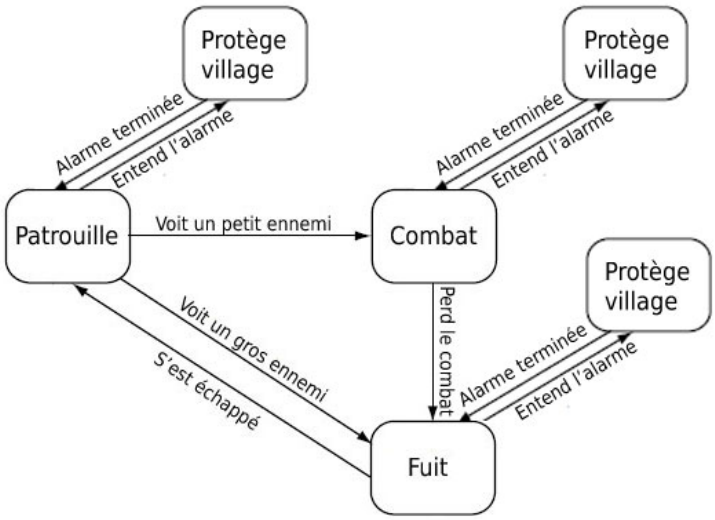


FIG. 3 – Un évènement d'alerte est ajouté au cycle d'activité du soldat. (Millington, 2006)

La machine à états comporte maintenant trois nouvelles occurrences du même état (reliées à tous les autres états puisque l'alerte peut survenir à n'importe quel moment, et donc pendant que le soldat est dans n'importe lequel des autres états). On peut voir qu'en ajoutant un simple événement d'alerte, le nombre d'états a été multiplié par deux, on imagine aisément qu'appliquer ce genre de méthode sur une intelligence artificielle plus complexe, comprenant des centaines d'états différents dont plusieurs états d'alerte, augmentera exponentiellement le nombre d'occurrences de ces états (Munoz-Avila, 2005), ce qui impliquerait une surcharge du travail de l'ordinateur (plus d'états donc plus de transitions entre ces états et donc plus de conditions à tester) ainsi qu'un travail supplémentaire au niveau humain puisque la création d'une IA cohérente deviendrait un véritable casse-tête pour les développeurs.

Cette répétition peut être évitée en utilisant des machines à états hiérarchisées. Ces FSM particulières, possèdent des états qui encapsulent eux-mêmes une autre machine à états. Pour notre exemple, on peut créer deux états globaux «surveille territoire» et «protège village» et placer une sous machine à états dans chacun de ces états globaux pour détailler le comportement du soldat en fonction de la situation courante. On peut voir sur la figure 4

qu'une FSM gérant la surveillance du territoire (trois états différents) est encapsulée dans la machine à états globale gérant les actions générales du soldat (deux états différents). Ainsi, les états de type «patrouille», «combat» et «fuit» ne peuvent être actifs que lorsque l'état de la FSM de niveau supérieur les contenant est lui-même actif (donc en cas d'absence d'alarme).

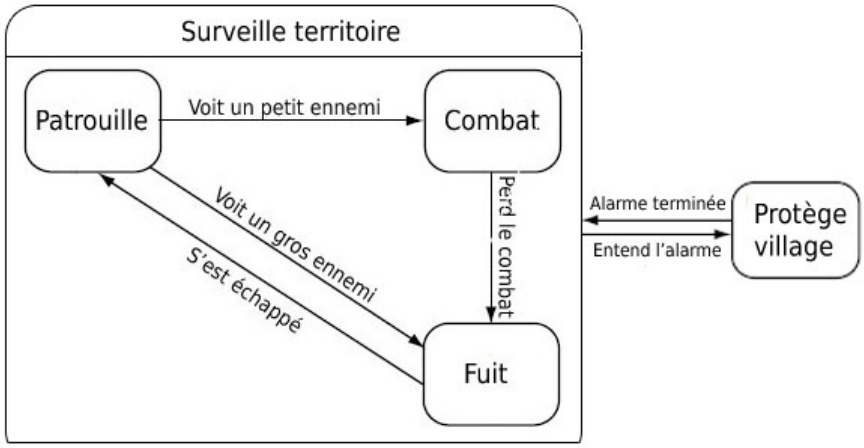


FIG. 4 – Les FSM peuvent contenir elles-mêmes des sous machines à états.

1.3 Machine à états pyramidale

Les machines à états pyramidales sont un type particulier de machine à états hiérarchisée. Plutôt que de ne pouvoir contenir qu'une seule sous-FSM par état, une machine à états pyramidale peut en encapsuler plusieurs qui géreront les comportements parallèles de différentes entités physiques. Elles sont principalement utilisées lorsque des groupes de plusieurs entités ont des buts communs.

L'intelligence de chaque unité est donc «contenue» dans une FSM de niveau supérieur qui gérera l'action générale à suivre par le groupe. Chaque unité pourra agir de manière asynchrone mais dès que la FSM globale changera d'état, les unités stopperont leurs actions et passeront à un des sous-états disponibles dans le nouvel état de la FSM globale. Les machines à états pyramidales s'accordent particulièrement avec les jeux de stratégies en temps réel (figure 5), dans lesquels un grand nombre d'unités organisées en escouades poursuivent des buts communs indiqués par le joueur (défendre telle zone, attaquer tel bâtiment, ...).

On peut à nouveau reprendre l'exemple précédent. Cette fois-ci, quatre soldats patrouillent ensemble. La machine à états globale conserve les mêmes états que la première version à soldat unique mais de nouveaux sous-états ont été ajoutés afin d'augmenter la variété des actions. Afin de na pas surcharger l'illustration (figure 6), nous ne développerons que l'état global «combattent» mais les deux autres, «patrouillent» et «fuient» pourraient eux-aussi avoir été agrémentés de nouveaux sous-états.

En situation de combat, les soldats doivent maintenant viser et avoir une arme chargée avant d'attaquer un ennemi. Grâce à cette organisation pyramidale, les quatre soldats exécuteront les mêmes tâches et suivront les ordres du joueur en groupe et ce, tout en prenant en compte les données attachées à chaque unité. Si un soldat s'arrête de tirer pour

FIG. 5 – Warhammer 40,000 : Dawn Of War (Relic Entertainment, 2004)

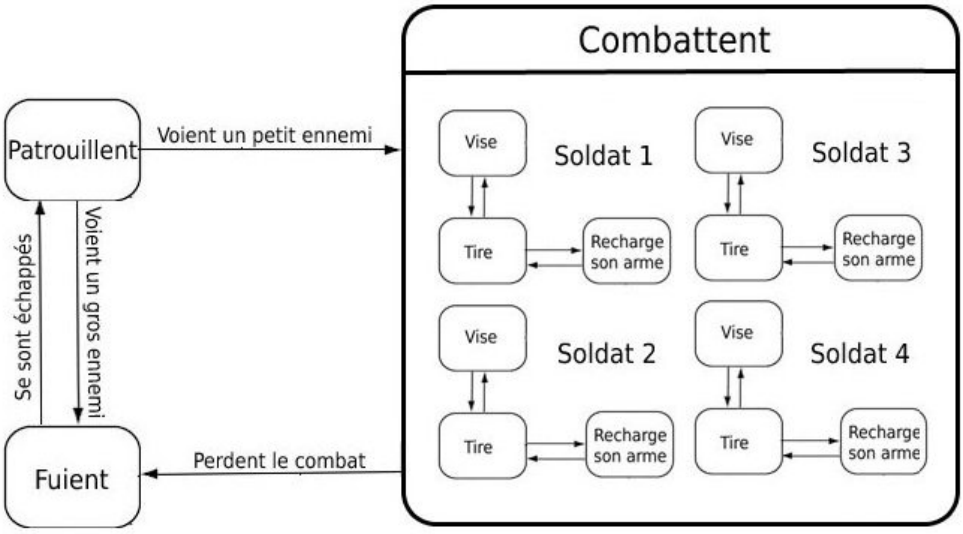


FIG. 6 – Une seule machine à état pour contrôler quatre entités différentes.

recharger car ses munitions diminuent dangereusement, les trois autres pourront continuer d'attaquer l'ennemi car chaque entité est indépendante.

Cette utilisation des machines à états permet d'organiser des comportements de groupe et donc un réalisme accru mais sera plus coûteuse en puissance de calcul puisqu'à chaque cycle de jeu, il faudra vérifier pour chaque unité si une des transitions de l'état courant est activable mais aussi réaliser les mêmes vérifications pour chaque niveau supérieur de la hiérarchie de la FSM contenant l'unité.

La machine à états a longtemps été la solution favorite des développeurs de jeu, attirés par la logique de son fonctionnement et sa capacité à simuler des réactions cohérentes. Néanmoins, le réalisme des dernières productions vidéoludiques a atteint des sommets ces dernières années et les techniques d'implémentation d'intelligence artificielle se doivent d'évoluer avec l'industrie. Les FSM, qui nécessitent une implémentation de chaque comportement possible, sont donc petit à petit remplacées par techniques moins lourdes et plus aisément réutilisables telles que les arbres décisionnels ou, à plus petite échelle, les réseaux de neurones.

2 Recherche de chemin

L'efficacité des déplacements d'un personnage non joueur dans un jeu vidéo est décisive à la fois pour la jouabilité, et pour maintenir l'impression de réalisme.

Dans un jeu de stratégie en temps réel, lorsqu'un joueur ordonne à l'une de ses unités de se déplacer vers un coin de la carte pour défendre sa base, il est souvent crucial que l'unité en question choisisse le chemin qui l'y emmènera le plus rapidement. Si cette unité s'engage dans une impasse, ou s'obstine à rentrer dans un mur perpétuellement, alors l'impression d'intelligence de l'unité est perdue, et la patience du joueur avec.

Pour s'assurer des résultats, les développeurs peuvent définir à l'avance les déplacements des PNJ à l'aide de points de cheminement (pour des chemins de patrouille). Cette méthode, bien qu'efficace, ne donne que des chemins statiques et n'est donc pas utilisable dans tous les types de jeu. Dans tous les jeux où un chemin doit être déterminé dynamiquement (à la suite d'un ordre d'un joueur, ou si un élément du décor a changé), des algorithmes doivent s'en charger.

La recherche de chemin est un sujet bien établi dans la littérature scientifique. De nombreux algorithmes existent pour donner les plus courts chemins (il en existe souvent plusieurs équivalents) reliant deux sommets, que nous appellerons départ et arrivée, dans un graphe. Les graphes considérés sont souvent pondérés : un coût de passage est associé à chaque arête.

L'algorithme de choix dans les jeux vidéos est A* (Hart *et al.*, 1968). A* répond bien au problème car il est optimal, c'est à dire qu'il donne un des plus courts chemins entre les deux sommets, et il est informé. Les algorithmes de recherche informés s'aident d'une fonction heuristique qui estime la distance à l'arrivée, ce qui permet d'orienter la recherche vers le but, au lieu d'explorer toutes les possibilités, à l'aveugle.

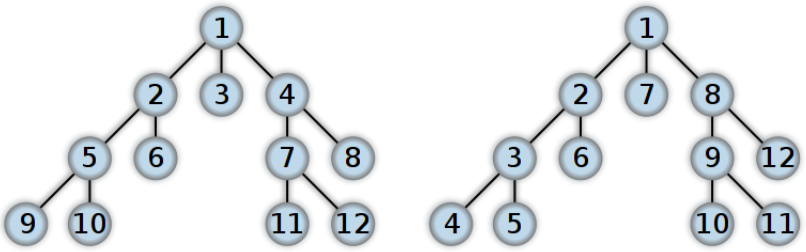
2.1 A*

Le schéma de fonctionnement général d'un algorithme de recherche de chemin dans un graphe est le suivant (Russel et Norvig, 2003) : on maintient une file d'attente de sommets à visiter (la liste ouverte), et tant que cette file n'est pas vide, on teste le premier élément pour savoir si c'est l'arrivée (on arrête alors la recherche car on a trouvé le chemin), puis on étend la file de sommets en y rajoutant les voisins des sommets qu'elle contient (les sommets connectés par une arête à un sommet de la liste) à l'aide d'une fonction qui choisit quels sommets voisins à ajouter ainsi que l'ordre dans lequel ils sont ajoutés. C'est cette fonction, un paramètre dans l'algorithme général, qui va définir le comportement de la recherche.

Par exemple, si l'on choisit d'étendre la file en largeur, c'est à dire en rajoutant les sommets racine avant les voisins, alors le sous graphe exploré durant la recherche va croître en cercle autour du sommet de départ, en testant les sommets plus proche du départ avant ceux plus éloignés. À l'inverse, si la file est étendue en profondeur, en rajoutant les sommets racines après leurs voisins, le sous graphe exploré va se construire par « branches » qui vont pousser successivement. La figure 7 illustre ces deux parcours.

Ces deux exemples de fonctions définissent les algorithmes dits de parcours en largeur et de parcours en profondeur, respectivement. Ce sont des algorithmes dits aveugles, car ils ne se servent que des informations données par le graphe (adjacence des sommets et coûts des arêtes)

L'ordre choisi par la fonction peut aussi être défini par une heuristique, une estimation de la distance entre un sommet donné et le sommet de départ, ou le sommet d'arrivée.



(a) Recherche en largeur (b) Recherche en profondeur

FIG. 7 – Algorithmes de parcours de graphe. Les sommets sont explorés dans l'ordre indiqué par les nombres qu'ils contiennent. (Wikipedia)

On ne peut que se satisfaire d'une estimation de la distance à l'arrivée, car pour la plupart des problèmes, connaître la distance exacte reviendrait à connaître un chemin entre un sommet quelconque et l'arrivée, or c'est le problème que l'algorithme essaye précisément de résoudre. Pour la distance au sommet de départ, comme le sommet donné fait partie d'un chemin construit par l'algorithme, on connaît tous ses ancêtres et il suffit de sommer les poids des arêtes qui constituent ce chemin.

Pour beaucoup de jeux, l'heuristique de distance employée sera soit la distance euclidienne (le « vol d'oiseau ») ou la distance de Manhattan pour les jeux qui se déroulent sur un échiquier.

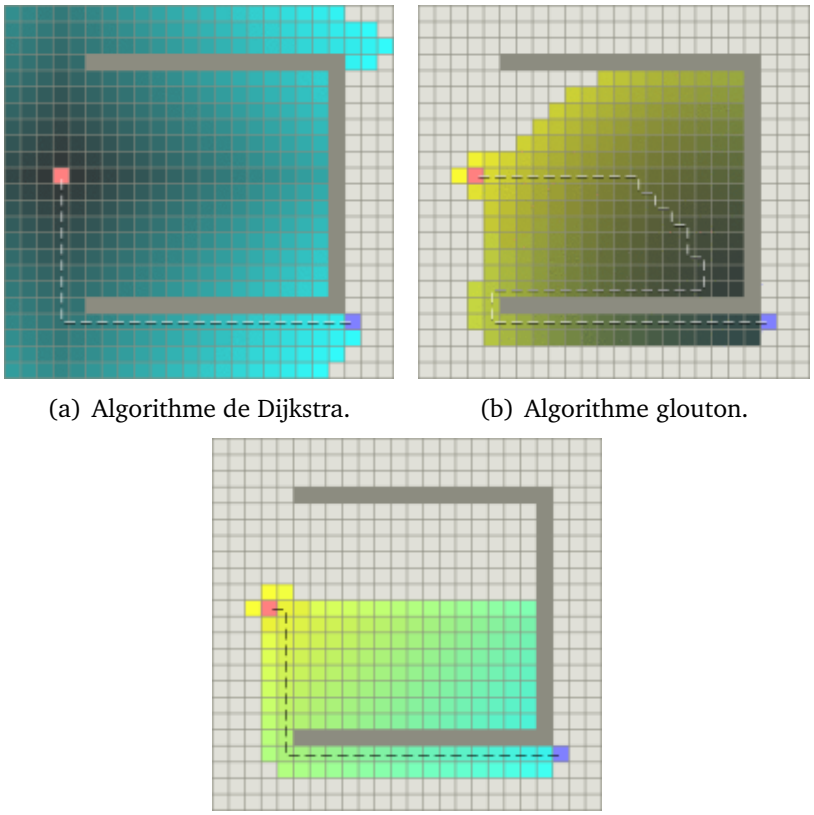
Cette fonction heuristique permet d'orienter la recherche en choisissant d'évaluer d'abord les sommets proches l'arrivée, ou les sommets proches du départ, ou bien des sommets proches à la fois du départ et de l'arrivée. Cette approche de la recherche de chemin ressemble davantage à celle qu'aurait un humain confronté au même problème.

L'idée de l'algorithme A* est de combiner deux approches préexistantes : la recherche gloutonne, qui consiste à minimiser la distance à l'arrivée à l'aide de l'heuristique , et l'algorithme de Dijkstra qui cherche à minimiser le distance du départ. On peut voir une comparaison des trois approches sur la figure 8. L'algorithme de Dijkstra et A* retournent un chemin optimal, mais Dijkstra explore beaucoup plus de cases qu'il n'est nécessaire. On voit qu'avec la recherche gloutonne les cases très éloignées de l'arrivée ne sont jamais considérées, mais le chemin est assez mauvais. A* retourne le bon chemin, tout en évitant les cases qui s'éloignent du but. Il est donc plus rapide.

Même si l'algorithme A* est satisfaisant pour beaucoup d'applications, la complexité croissante des jeux vidéos, qui se veulent toujours plus réalistes, engendrée par la simulation d'environnements de plus en plus dynamiques (destruction du terrain et du décor), et le nombre important d'entités (qui doivent toutes trouver leur chemin), requiert des algorithmes adaptés pour maintenir l'illusion de l'intelligence parmi ses agents.

2.2 Variantes d'A*

Il y a plusieurs chemins possibles pour optimiser A*. D'abord, les besoins du jeu sont prioritaires : si la carte est très dynamique, alors pendant qu'une entité effectue la recherche de chemin, il se peut que le graphe sur lequel elle travaille ait changé, et il faut recommencer. En pratique, A* s'exécute rapidement sur les machines modernes (en particulier sur les machines des joueurs, qui doivent être souvent renouvelées pour profiter des dernières technologies), mais si le graphe est très large, ou si le nombre d'entités est important, alors une partie significative du temps processeur peut y être consacrée. C'est pourquoi plusieurs



(a) Algorithme de Dijkstra. (b) Algorithme glouton.

(c) A*.

FIG. 8 – Le case rose représente le départ, et la violette l'arrivée. Le cyan indique la distance de la case colorée à la case de départ (les plus claires sont les plus éloignées), et le jaune indique la valeur de la fonction heuristique (les cases jaune clair sont les plus éloignées de l'arrivée). Le chemin retourné est tracé en pointillés. (Mitchell, 2008)

variantes dynamiques d'A* ont été développées.

L'algorithme D* (Stentz, 1994) est capable de redéfinir un chemin de manière efficace lorsque l'agent rencontre un obstacle. L'algorithme met à jour sa représentation locale du graphe en répercutant un changement du coût de passage d'une arête sur les sommets voisins. Comme seule une partie du graphe est changée en cas d'obstacle (celle nécessaire au chemin de l'agent), il y a un net gain d'efficacité en temps d'exécution, proportionnel à la taille du graphe. En revanche, pour la même raison, si l'agent n'a aucune connaissance du graphe à l'avance, le chemin emprunté ne sera pas optimal au sens d'A*, mais localement optimal (c'est le plus court envisageable avec les informations dont dispose l'agent). On peut cependant améliorer ces résultats en fournissant à l'agent une carte grossière (un graphe regroupant des sommets proches en un seul) des obstacles connus à l'avance.

Une autre optimisation possible concerne l'espace mémoire pris par l'exécution de l'algorithme de recherche. Si la représentation du monde du jeu est large, comme A* maintient une liste des sommets du graphe en mémoire, un trop grand espace mémoire est alors utilisé pour la recherche de chemin. La mémoire étant souvent réduite sur les consoles de jeux, c'est une optimisation souhaitable sur ces machines.

L'algorithme IDA* (pour *Iterative Deepening A**) (Korf, 1985) économise la mémoire en n'explorant qu'un seul chemin à la fois (recherche en profondeur), parmi les chemins dont le score (somme de la distance du sommet de départ et de l'heuristique) est inférieur à une valeur donnée. Tant que la solution n'est pas trouvée, cette valeur est incrémentée au

palier suivant, et l'algorithme teste ainsi des chemins plus longs. Comme un seul chemin est conservé en mémoire à la fois, la mémoire totale utilisée pendant l'exécution d'IDA* est proportionnelle à la longueur du chemin qui mène à l'arrivée. Cependant, l'incrémentation du rayon de la « sphère de recherche » conduit à revisiter des sommets déjà explorés lors d'une itération précédente, ce qui gâche des cycles processeurs. Il s'agit donc d'un compromis à faire, suivant l'importance accordée par le jeu à la mémoire et au temps processeur.

Lorsque la représentation du monde fournit un espace de recherche trop large pour permettre au nombre désiré d'agents de s'exécuter rapidement, on peut faire appel aux abstractions. Au lieu d'effectuer la recherche au niveau atomique, on peut créer un niveau d'abstraction qui remplace les zones semblables (forêts, ville, désert) par un sommet « bloc », en conservant les connexions entre les blocs ainsi créés. Au besoin, le processus d'abstraction peut être répété pour réduire l'espace de recherche, tant que sa résolution reste signifiante pour le problème.

C'est l'approche hiérarchique de la recherche de chemin implémentée par des algorithmes comme HPA* (Botea *et al.*, 2004). HPA* commence par créer un niveau d'abstraction qui étudie les frontières des blocs délimités pour y déceler les « transitions », qui deviennent les arêtes du graphe construit par l'abstraction. Ensuite A* est utilisé pour trouver le chemin du bloc qui contient le sommet de départ au bloc qui contient le sommet d'arrivée. Comme ce chemin est optimal pour l'abstraction, mais pas nécessairement pour le graphe atomique, il faut alors utiliser A* dans chaque bloc pour obtenir le chemin local optimal. Les avantages sont multiples : ayant obtenu un chemin abstrait, l'agent peut commencer à se déplacer après avoir seulement calculé le chemin du bloc dans lequel il se trouve ; si le monde change, ou si l'agent est bloqué pour une certaine raison (attaqué par un joueur par exemple), il n'aura pas eu à calculer de chemin détaillé pour les autres blocs qui le séparent du but, qui ne lui aurait pas servi.

2.3 Représentation du monde

Nous l'avons vu, les algorithmes de recherche de chemin opèrent sur un graphe abstrait. Dans un jeu vidéo, il est rare de représenter le monde au joueur sous forme d'un graphe, ou même de s'en servir comme structure de donnée interne.

Actuellement, la grande majorité des jeux commerciaux sont en 3D, et on peut fabriquer le graphe abstrait à partir des polygones qui constituent le monde. Le nombre de ces polygones étant généralement très élevé pour rendre le monde visuellement riche et détaillé, on partitionne les zones où les agents doivent pouvoir se déplacer en un treillis de polygones convexes (pour qu'il soit facile de trouver un chemin entre deux points d'un même polygone) (Tozour, 2002). Ce treillis utilisé pour la recherche de chemin est invisible pour le joueur. Son élaboration peut être automatisée à l'aide d'algorithmes : si les niveaux sont statiques, où ne sont pas sujets à de modifications majeures pendant le jeu, on peut se permettre d'utiliser des algorithmes de pavage optimaux, qui ont un temps d'exécution prohibitif pour une utilisation répétée pendant l'action. Néanmoins, le plus souvent les personnes chargées de l'élaboration des niveaux définissent elles-mêmes ce treillis, ou au moins contrôlent celui généré par l'algorithme de pavage, pour s'assurer que les personnages non joueurs se déplacent aux endroits voulus.

Une autre méthode, plus adaptée pour les jeux se déroulant en deux dimensions, consiste à plaquer un échiquier sur le monde et considérer chaque case comme un sommet du graphe, et relier deux sommets par une arête s'il y a un chemin entre les cases qu'ils représentent (Smed et Hakonen, 2006). Comme pour la recherche hiérarchique, la taille des

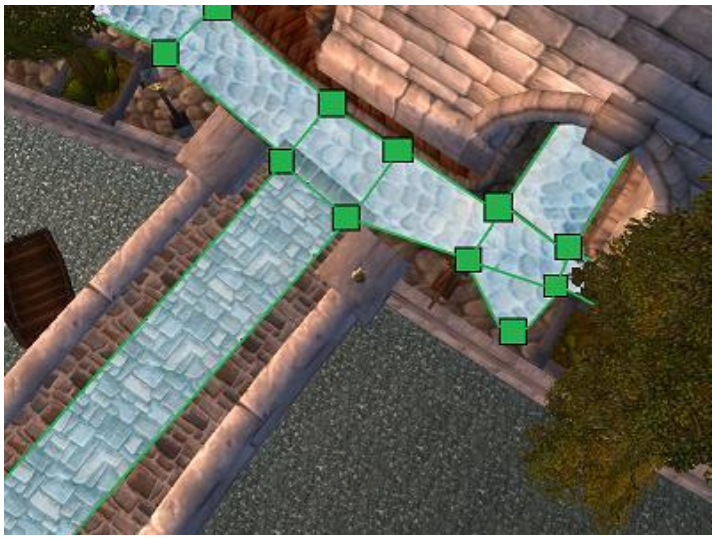


FIG. 9 – Un exemple de treillis de polygones dans *World Of Warcraft* (Blizzard Entertainment, 2004). (Tozour, 2008)

cases est un facteur important pour la performance de l'algorithme de recherche, aussi bien pour le temps qu'il prendra à s'exécuter, et pour la pertinence du résultat. Si les cases sont trop larges, un chemin sera facile à obtenir, mais il ne prendra peut être pas en compte des obstacles d'échelle inférieure. Une division hiérarchique peut être facilement utilisée pour ce genre de représentations. Le chemin étant établi de cases en cases, une procédure de raffinement de ce chemin est souvent mise en œuvre pour éviter à l'agent d'avoir une démarche trop carrée (voir les chemins de la figure 8).

Conclusion

Nous avons présenté des techniques utilisées dans les jeux vidéos pour donner l'illusion au joueur que les agents du jeu agissent de façon intelligente. Il existe bien sûr d'autres méthodes pour donner aux agents des comportements réalistes, employées avec plus ou moins du succès dans les jeux modernes.

Notons par exemple l'emploi de techniques de computation inspirée de la nature, telles que les réseaux de neurones artificiels dans *Black & White* (Lionhead Studios, 2001) et les algorithmes génétiques dans *Creatures* (Millennium Interactive, 1996) pour simuler des créatures vivantes qui interagissent entre elles et avec le joueur.

Malheureusement, même si ces jeux ont ravi une certaine partie du public, les résultats n'ont pas encouragés d'autres compagnies à innover dans ce domaine, et encore aujourd'hui, l'argument de vente principal reste l'image. Cependant, il est probable qu'une fois que les graphismes auront atteint leur limite de réalisme, l'élément immersif majeur encore manquant sera l'IA des personnages non joueurs, et les efforts des compagnies se tourneront alors vers ce domaine.

Références

BOTEA, A., MÜLLER, M. et SCHAEFFER, J. (2004). Near optimal hierarchical path-finding. *Journal of Game Development*, 1(1):7–28.

BOURG, D. M. et SEEMANN, G. (2004). *AI for game developers*. O'Reilly Media, Inc.

HART, P., NILSSON, N. et RAPHAEL, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107.

KORF, R. (1985). Depth-first iterative-deepening : An optimal admissible tree search. *Artificial intelligence*.

MILLINGTON, I. (2006). *Artificial Intelligence for Games*. Morgan Kaufmann.

MITCHELL, J. (2008). A* comparison. `http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html`. Accédé le 21 mai 2009.

MUNOZ-AVILA, H. (2005). Finite state machines in games. `http://www.cse.lehigh.edu/~munoz/ComputerGameDesignClass/classes/FSMandScripts.pptx`. Accédé le 18 mai 2009.

RUSSEL, S. J. et NORVIG, P. (2003). *Artificial Intelligence : A Modern Approach*. Prentice Hall, seconde édition.

SMED, J. et HAKONEN, H. (2006). *Algorithms and Networking for Computer Games*. Wiley.

STENTZ, A. (1994). Optimal and efficient path planning for partially-known environments. *In Proceedings of the IEEE International Conference on Robotics and Automation*, pages 3310–3317.

TOZOUR, P. (2002). Building a near-optimal navigation mesh. *AI Game Programming Wisdom*, pages 171–185.

TOZOUR, P. (2008). Fixing pathfinding once and for all. `http://www.ai-blog.net/archives/000152.html`. Accédé le 21 mai 2009.