INFORMATIQUE COGNITIVE

IFT-703

TP-1

ACT-R

Le Monde du *Wumpus*

Travail présenté à

M. André Mayers

Par

Julien Filion 07 177 770

Université de Sherbrooke, Département d'informatique

Octobre 2010

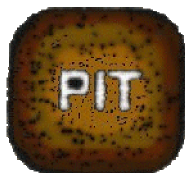
Table des matières

Problématique 3
Interface 4
  Touche clavier 5
Analyse préliminaire 5
But hiérarchique 6
  Analyse cognitive 6
  ACT-R 6
Analyser une pièce 7
  Analyse cognitive 7
  ACT-R 7
Sélectionner la prochaine pièce à visiter 8
  Analyse cognitive 8
  ACT-R 8
    Apprentissage 9
Trouver un chemin 9
  Analyse cognitive 9
  ACT-R 10
Suivre un chemin 11
  Analyse cognitive 11
  ACT-R 11
Simplifications 11
Conclusion 11

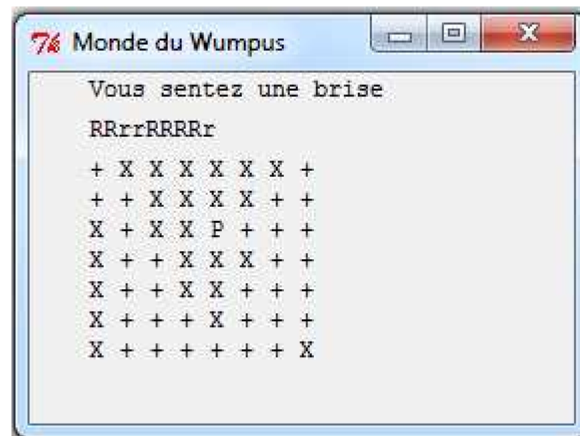
Problématique

Pour le projet, j'ai décidé de résoudre le problème du Wumpus tel que décrit dans le livre de Stuart Russel et Peter Norvig. Dans ce problème, il y a un joueur qui évolue sur une grille rectangulaire de taille variable. Il a comme but d'explorer la grille une cellule à la fois pour retrouver l'or. Le joueur peut se déplacer à gauche, à droite, en haut et en bas, mais pas en diagonale. La grille du jeu est composée de plusieurs cellules qui peuvent être libres, contenir un trou mortel, abriter le Wumpus ou contenir un lingot d'or. Il ne peut y avoir qu'un seul lingot d'or et un seul Wumpus par grille, mais il peut y avoir plusieurs trous mortels. Le joueur doit trouver le lingot d'or en évitant d'être dévoré par le Wumpus et de tomber dans un trou mortel. Afin d'éviter les trous mortels, ainsi que le Wumpus, le joueur se trouvant dans une pièce adjacente au Wumpus perçoit une odeur. Lorsque le joueur se trouve sur une pièce adjacente à un trou mortel, il perçoit un courant d'air. Le joueur a aussi la possibilité de tirer une flèche en direction du Wumpus pour le tuer.

Lors du développement de mon logiciel, quelques modifications ont été apportées au problème original. Premièrement, le joueur ne peut pas tirer de flèche pour tuer le Wumpus, le problème était bien assez complexe pour le projet sans cette option. Deuxièmement, plutôt que de sentir une odeur lorsqu'on est près du Wumpus, on entend un grognement, ce qui permet d'utiliser les capacités auditives d'ACT-R.



Interface



Interface du Wumpus

L'interface du logiciel est composée de trois éléments, soit deux zones de texte et une grille. La première zone de texte affiche le message « Vous sentez une brise » lorsqu'il y a un trou dans une pièce adjacente à la pièce courante. Cette zone possède une seconde utilité, elle permet aussi de savoir si la partie est terminée, le texte « Vous avez gagné » est affiché si vous avez trouvé l'or et le texte « Vous avez perdu » est affiché si vous avez croisé le Wumpus ou si vous êtes tombé dans un trou.
La deuxième zone de texte permet d'afficher le texte « RRrrRRRRr » lorsqu'un grognement est entendu, c'est-à-dire lorsque l'on se trouve près d'un Wumpus. Ce texte n'est utile que pour les joueurs humains, car le module audio d'ACT-R ne produit aucun son réel. Lorsqu'un modèle d'ACT-R joue au jeu, il n'utilise pas cette information, il se fie plutôt à son module auditif.

Finalement, la grille représente l'état du jeu, chaque case représente une pièce. Voici la signification de chaque symbole :

P	Position du joueur
+	Pièce visitée
X	Pièce non visitée
T	Trou
W	Wumpus
G	Or

Touche clavier

Afin d'interagir avec le jeu, le joueur doit appuyer sur des touches claviers. Voici la liste des touches et leurs effets :

w	Déplace le joueur d'une case vers le haut.
a	Déplace le joueur d'une case vers la gauche
s	Déplace le joueur d'une case vers le bas.
d	Déplace le joueur d'une case vers la droite.
r	Démarre une nouvelle partie.

Analyse préliminaire

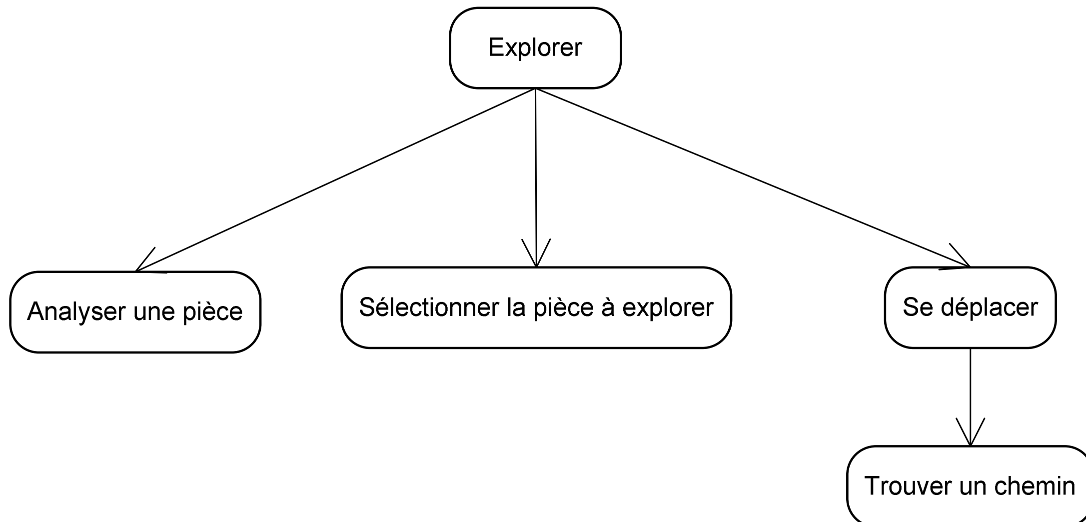
Le problème du Wumpus est un problème qui demande à l'humain de résoudre différentes tâches pour arriver à ses fins. Son but principal est de trouver l'or, pour ce faire il doit explorer le plus de cases possible tout en évitant le Wumpus et les trous.
La première tâche qu'il doit effectuer est de choisir une pièce inexplorée et accessible à visiter. Cette tâche peut impliquer de faire du résonnement probabiliste pour limiter les risques de perdre.
Par la suite, le joueur doit être en mesure de se déplacer vers la pièce sélectionnée, il doit donc trouver un chemin de la pièce courante à la nouvelle pièce.
Une fois que le chemin est trouvé, on doit fournir à l'interface les bonnes commandes pour suivre ce chemin.

De plus, le joueur évolue dans un environnement dynamique, la grille change au fur et à mesure que le joueur se déplace. Il doit pouvoir modéliser l'environnement dans lequel il évolue et analyser les informations qui lui sont fournies. Le joueur va donc se fier à ses sens et à sa mémoire déclarative pour conserver les informations sur la grille de jeu. Par contre, puisqu'il y a une grande quantité d'information à retenir, le joueur a de fortes chances d'oublier certaines informations.

But hiérarchique

Analyse cognitive

Lors de mon analyse du problème, j'ai remarqué qu'un humain avait tendance à structurer ses buts ou ses tâches de manière hiérarchique. Par exemple, pour le problème du Wumpus, nous cherchons un chemin dans le but de nous déplacer et on se déplace dans le but d'explorer la grille. J'ai donc appliqué ce principe à mon modèle et j'ai créé une hiérarchie de but simple.



Structure de but hiérarchique pour le problème du Wumpus

ACT-R

Pour implanter cette structure dans le modèle d'ACT-R, j'ai dû ajouter trois slots aux chunks qui modélisent des buts.

Parent	Contient une copie du but qui a créé ce but.
Parent-state	Contient l'état que doit prendre le but parent lorsqu'il sera réactivé.
Subtask-state	Permet à un but enfant de transmettre de l'information au but parent, par exemple indiquer si le but a été atteint ou non.

Lorsque le modèle doit créer un sous-but, celui-ci est créé dans le tampon *imaginal* avec ses valeurs initiales et le tampon *goal* est copier dans le slot *Parent* du sous but. Le tampon *imaginal* est par la suite copiée dans le tampon *goal*. À l'inverse, lorsque l'on veut reconstruire un but parent, on copie d'abord celui-ci dans le tampon *imaginal*, puis on modifie sont slot *State* par la valeur du *Parent-state* et on modifie son slot *Subtask-state* par la valeur à transmettre (par exemple *erreur* ou *succès*). Par la suite le tampon *imaginal* est copié dans le tampon *goal*.

Analyser une pièce

Analyse cognitive

L'un des aspects importants du jeu est de pouvoir modéliser son environnement et de se souvenir de l'état des pièces. Le joueur doit se servir de sa vision pour voir si une pièce contient une brise et de son ouïe pour entendre les grognements. De plus, il doit pouvoir estimer les risques d'explorer une nouvelle pièce. Par exemple, puisqu'il n'y a qu'un seul Wumpus, on peut déterminer où il se trouve si on entend deux grognements et ainsi diminuer les risques sur les cases adjacentes aux grognements. Évidemment, toutes ces informations doivent être stockées dans la mémoire déclarative du joueur. Puisqu'il peut y avoir une grande quantité de pièces, il n'est pas rare qu'un joueur oublie certaines cases et qu'il doive les réévaluer.

ACT-R

Mon modèle utilise le module auditif et le module visuel pour analyser l'état d'une nouvelle pièce. L'information est ensuite stockée dans la mémoire déclarative sous la forme d'un chunk *tile*. Les chunks *tile* contiennent l'information nécessaire pour identifier la pièce (sa position) ainsi que les éléments qu'elle contient (brise et grognement).

Faute de temps, j'ai dû simplifier grandement le calcul des risques, mon modèle n'est donc pas représentatif du comportement humain pour ce point. Plutôt que de calculer plusieurs niveaux de risque pour une pièce, je ne distingue que deux états, sécuritaire ou risqué. Une pièce est sécuritaire s'il y a au moins une pièce adjacente sans brise ni grognement.

J'ai fait face à un problème lorsque j'ai voulu conserver l'information des pièces non explorées, car l'état de la pièce peut changer lorsque l'on décide de les explorer. Or ACT-R n'offre aucune fonction pour modifier, supprimer ou inhiber un chunk déjà en mémoire déclarative. La seule alternative est d'ajouter un nouveau chunk, par contre cette alternative pose problème lorsque l'on fait une requête pour trouver une pièce qui n'a pas été visitée. Pour contrer ce problème, j'ai décidé de ne conserver que l'information statique, donc les pièces déjà visitées. On vérifie si une pièce inexplorée est sécuritaire seulement lorsque nécessaire, l'information n'est pas conservée.

Sélectionner la prochaine pièce à visiter

Analyse cognitive

La sélection d'une pièce à explorer est la partie la plus importante du jeu, en explorant la bonne pièce on gagne la partie, mais avec la mauvaise, on perd la partie. C'est pourquoi un joueur va chercher la pièce la plus sécuritaire en fouillant dans sa mémoire. De plus, le joueur aura tendance à choisir les pièces plus près de sa position que les pièces éloignées. Par contre, le joueur doit apprendre quand prendre un risque et quand chercher une pièce sécuritaire. Si le joueur est incapable de retrouver une pièce sécuritaire, il peut tenter d'en retrouver une en retournant sur des cases déjà visitées pour se rappeler l'information oubliée. Mais il arrive parfois que le joueur n'ait d'autre choix que de prendre un risque. Dans cette situation, le joueur cherche la pièce qui pose le moins de risque. Tout au long de ce processus, le joueur utilise sa vision pour chercher et valider de l'information.

ACT-R

Mon modèle possède trois méthodes différentes pour choisir la prochaine pièce. Il peut choisir une case adjacente à sa position, se souvenir d'une case ou choisir la case le plus près.

Lorsqu'il choisit de prendre une pièce adjacente, il doit d'abord évaluer le risque des cases adjacentes. Pour rester simple, on considère que les cases sont risquées si notre position courante possède une brise ou un grognement, sinon la case est sécuritaire. Lorsqu'une case est risquée, le modèle doit faire un deuxième choix, changer de méthode ou prendre un risque et visiter une pièce adjacente. Si le modèle ne change pas de stratégie, il analyse visuellement les cases qui l'entourent pour savoir quelle case n'a pas été visitée. Finalement, on sélectionne une des cases adjacentes inexplorées grâce au module procédural.

La deuxième méthode consiste à retrouver une pièce sécuritaire dans la mémoire déclarative. Par contre, pour des raisons mentionnées plus tôt, le modèle ne conserve pas d'information sur les pièces inexplorées. On ne peut donc pas chercher directement dans la mémoire déclarative une pièce inexplorée sécuritaire. Pour résoudre le problème, on cherche une pièce visitée qui ne possède pas de brise ni de grognement. Lorsque l'on trouve une telle pièce, on vérifie visuellement s'il y a une pièce adjacente inexplorée, si oui on la sélectionne, sinon on choisit à nouveau une méthode. Lorsque l'on cherche une pièce, on utilise la condition « :recently-retrieved nil » qui limite la recherche aux chunks qui n'ont pas été retrouvés récemment pour éviter d'évaluer deux fois la même pièce. Par contre, il y a une quantité limitée de marqueurs « recently-retrieved » ce qui limite l'utilisation de cette méthode à un nombre fixe de tentatives par itération. Dans le modèle courant, on ne peut l'exécuter que sept fois.

Apprentissage

Pour réaliser cette tâche, le joueur a des choix à faire et c'est pour cette raison que j'ai décidé d'ajouter de l'apprentissage à cette section. J'ai donc utilisé le système de récompense d'ACT-R pour permettre au modèle d'apprendre quelles procédures sont prioritaires. Le modèle reçoit donc une récompense positive lorsqu'il explore une nouvelle case et une récompense négative lorsqu'il perd la partie.

J'ai éprouvé un problème avec le système de récompense, car mon modèle possède plusieurs tâches et la récompense pour la résolution d'un problème ne devrait pas influencer l'utilité de procédure des autres tâches. Malheureusement, au moment d'attribuer la récompense pour la sélection d'une pièce, d'autres tâches ont dû être exécutées. Donner une récompense à ce moment influencerait l'utilité de ces autres tâches, ce qui ne semble pas cognitivement correct. Idéalement, la récompense serait donnée au moment où l'on voit la nouvelle pièce, mais seulement aux bonnes procédures. J'ai dû tricher un peu pour arriver à un résultat similaire. Je donne la récompense lorsque la pièce est sélectionnée en testant ce qui se cache derrière la pièce sélectionnée (ce que le joueur ne voit pas).

Trouver un chemin

Analyse cognitive

Trouver un chemin est une tâche qui peut être accomplie de plusieurs manières. En informatique, on utilise généralement A* ou une recherche en largeur. Par contre, je ne crois pas qu'un humain utilise de telles techniques. Un humain aura tendance à diriger son attention en direction de son objectif et de revenir à des chemins qu'il n'a pas analysés lorsque son chemin est bloqué, d'une manière similaire à une recherche en profondeur.

De plus, un humain à tendance à voir la grille comme un ensemble de « zones » qui représente un ensemble de pièces adjacentes de même type. Ces zones permettent de faciliter la recherche de chemin, mais elles doivent être modifiées au fur et à mesure que le joueur se déplace.

Aussi, lorsqu'un humain cherche un chemin, il utilise principalement ses capacités visuelles pour voir rapidement quels sont les chemins disponibles.

ACT-R

La recherche de chemin a été complexe à implanter dans ACT-R, et le modèle décrit dans l'analyse cognitive n'a pas été complètement respecté. Tout d'abord, la gestion de zone semblait trop compliquée pour être implantée dans ce modèle, la recherche est donc faite case par case. Par contre, le modèle se base principalement sur les informations visuelles pour atteindre son but.

La recherche de chemin utilise deux types de chunk. Les chunk *Explored-tile* qui représentent des pièces déjà analysées par ce processus et les chunk *way-node* qui représentent des pièces du chemin trouvé. Ces deux types possèdent des slots qui permettent d'identifier leurs pièces ainsi qu'un slot *Iteration* qui permet au joueur de ne pas se mélanger avec les chunk d'un appel précédent de la recherche de chemin. Voici le processus que suit le modèle pour trouver un chemin. L'algorithme est initialisé à la case où le joueur se trouve.

1. On crée un chunk *Explored-tile* dans le buffer *imaginal*.
2. Si le but est une pièce adjacente, l'algorithme termine et on génère le chemin.
3. On choisit une direction (gauche, droite, haut, bas).
4. On regarde la pièce dans cette direction avec le module visuel.
5. Si la pièce n'a pas été visitée par le joueur, elle est marquée comme invalide et on passe à l'étape 8.
6. On vérifie dans la mémoire déclarative que la pièce n'a pas été explorée par l'algorithme.
7. Si la pièce a été explorée, on marque la pièce comme invalide.
8. Si le but se trouve dans une direction qui est valide, on choisit cette direction et on passe à l'étape 12.
9. Si une direction est valide et que les directions vers le but ne sont pas valides, on choisit cette direction et on passe à l'étape 12.
10. Si aucune des directions n'est valide, on cherche une autre pièce à explorer
    a. On cherche dans la mémoire déclarative une pièce déjà explorée.
    b. On déplace l'attention visuelle du modèle vers cette case.
    c. On copie le chunk dans le buffer *imaginal*.
    d. On reprend à l'étape 3
11. Si nous n'avons pas assez d'information, on analyse une nouvelle direction (on retourne à l'étape 4).
12. On déplace l'attention visuelle du modèle vers la pièce choisie.
13. On inscrit le chunk *Explored-tile* dans la mémoire procédurale.
14. On recommence à l'étape 1 en prenant note de la direction utilisée dans le nouveau chunk *Explored-tile*.

Une fois que le but est trouvé, on doit construire un ensemble de chunk *way-node* qui représente le chemin à suivre depuis la case initiale jusqu'à la case finale. Pour ce faire, on construit le chemin en partant du but et en suivant les directions contenues dans les *Explored-tile* pour trouver le prochain nœud à ajouter.

Suivre un chemin

Analyse cognitive

Lorsque le joueur veut se déplacer, il doit d'abord créer un chemin pour atteindre son but comme décrit précédemment. Par la suite il va tenter d'exécuter ce chemin avec les touches clavier. Il peut arriver que le joueur se trompe et perde son chemin, dans cette situation, il devra refaire un plan pour se rendre au but.

ACT-R

Dans l'implantation de mon modèle, je ne gère pas encore les erreurs qui peuvent se produire si le joueur se perd. Le modèle ne fait que suivre le chemin dicté par les *way-node* construits lors de la recherche de chemin. Pour ce faire, il utilise le module moteur d'ACT-R qui lui permet d'appuyer sur les touches claviers.

Simplifications

Pour réaliser ce projet, j'ai dû tricher à quelques endroits pour simplifier le modèle. Premièrement, j'ai dû utiliser les opérations d'adition et de soustraction de LISP pour incrémenter les numéros d'itération et les numéros de monde ainsi que pour calculer les positions des pièces. Deuxièmement, j'ai utilisé la fonction *!eval!* pour créer des conditions logiques « ou » et « et » dans certaine fonction. À mon avis, un humain est capable de raisonner avec des conditions logiques simples sans devoir faire de calcul additionnel.

Conclusion

Pour conclure, ce projet est fonctionnel, mais loin d'être terminé, il y a encore beaucoup à faire pour modéliser complètement un humain qui joue au Monde du Wumpus. ACT-R semble être un bon système pour modéliser le comportement humain, mais pourrait tout de même être amélioré sur certains points. Par exemple, j'ai éprouvé des difficultés à travailler dans un environnement dynamique, j'ai l'impression qu'il manquait certaine fonctionnalité pour modifier où inhiber des donné présente dans la mémoire déclarative. Une autre alternative aurait été d'inclure un module de mémoire de travail, une mémoire flexible, mais qui n'est pas persistante contrairement à la mémoire déclarative.