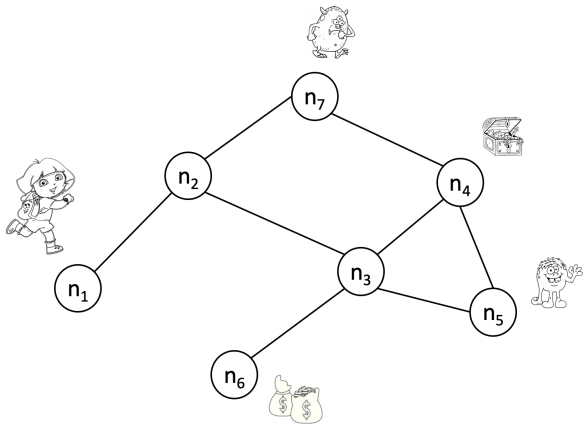
Licence MIASHS 2019-2020

Projet d'intelligence artificielle : le donjon



Ce projet consiste à programmer un agent qui se déplace dans un labyrinthe appelé "le donjon" (ici modélisé à l'aide d'un graphe). La première partie considère uniquement le déplacement de l'agent explorateur qui est entièrement "piloté" par l'utilisateur humain du système. La deuxième partie ajoute des éléments dans le labyrinthe : le trésor et des agents "monstre" statiques. La troisième partie apporte des premiers éléments d'autonomie pour l'agent aventurier. La quatrière partie donne quelques idées pour aller plus loin dans le développement de cette application.

Partie 1 - modéliser le labyrinthe et déplacer l'agent explorateur

Exo 1 : Pour commencer, l'important est de modéliser le labyrinthe dans la base de faits. Cela doit être réalisé par l'intermédiaire de faits construits sur un prédicat $laby(n_x, n_y)$ qui indique qu'un chemin existe entre le nœud n_x et n_y. A priori, le graphe est orienté et donc le chemin est à sens unique. Si vous voulez qu'un chemin existe dans le sens inverse, une solution consiste à ajouter le fait symétrique. Une autre solution est d'adapter les règles de déplacements qui suivent.

Exo 2 : Ajoutez à la base de faits l'emplacement initial de votre aventurier (représenté par un nœud de votre choix dans le graphe).

Exo 3 : Ajoutez une première règle d'interaction, appelée `demander`, qui demande à l'utilisateur de choisir un endroit (donc le nom d'un nœud) vers lequel il souhaite diriger l'aventurier (par ex. n_2 dans l'exemple de la figure).

Exo 4 : Ajoutez une première règle de déplacement qui vérifie que le chemin existe bien entre la localisation actuelle de l'aventurier et le lieu vers lequel il se rend. La règle doit mettre à jour le lieu où se trouve l'aventurier et un message doit s'afficher pour indiquer le nouveau lieu découvert. Dans tous les cas, vous pouvez considérer que l'utilisateur du programme possède une carte du labyrinthe sous les yeux. Il peut donc facilement choisir le nom du lieu où il compte se rendre.

Exo 5 : Ajoutez une deuxième règle de déplacement qui doit se déclencher si le lieu vers lequel se dirige l'aventurier est inaccessible. Un message doit s'afficher pour en informer l'utilisateur.

Exo 6 : Ajoutez une règle qui doit se déclencher si l'utilisateur ne rentre aucune réponse afin que le programme ne s'arrête pas.

Partie 2 - l'explorateur rencontre des monstres et trouve des trésors

Exo 7 : Ajoutez deux nouveaux types de faits dans la base : un fait correspondant aux trésors contenus dans le labyrinthe et un fait correspondant aux monstres que peut rencontrer l'aventurier. Un trésor est caractérisé par un montant en pièces d'or. Un monstre est caractérisé par un score de force. Ajoutez également une caractéristique de force à votre aventurier (un score entier supérieur à 0).

Exo 8 : Ajoutez une règle qui doit se déclencher si l'aventurier arrive sur un lieu comportant un trésor. Il doit permettre de récupérer le trésor en ajoutant le nombre de pièces au "magot" accumulé depuis le début de l'aventure. Il faut pour cela savoir additionner les pièces au total déjà trouvé par l'aventurier (pensez à ajouter un fait correspondant dans la base).

Exo 9 : Ajoutez une règle qui doit se déclencher si l'aventurier arrive sur un lieu comportant un monstre de force plus faible que lui. Dans ce cas, le monstre est retiré du jeu et la partie continue.

Exo 10 : Ajoutez une règle si la force du monstre est supérieure à celle de l'aventurier. Dans ce cas, l'aventurier perd tout le trésor qu'il a accumulé jusqu'ici et perd la partie.

Exo 11 : Ajoutez une règle qui termine la partie et affiche le total du trésor découvert lorsque l'utilisateur écrit le mot "fin" comme choix d'action.

Il vous faudra peut-être ajouter une ou plusieurs règles pour gérer la fin du jeu qui peut subvenir de différentes manières. N'oubliez pas que vous pouvez utiliser la commande `(declare (salience N))` avec N une valeur entière indiquant la priorité de la règle en cas de conflit dans l'agenda.

Partie 3 - l'agent aventurier devient autonome

Exo 12 : Ecrivez une nouvelle règle `choix_direction` qui va remplacer la règle `demander` écrite précédemment : il n'est plus question de demander son choix à l'utilisateur (donc, plus d'appel à `readline`). Cette règle utilise comme motif d'entrée les informations du labyrinthe afin de connaître les nœuds voisins accessibles à partir de la position courante de l'aventurier. La limitation de cette règle est que le système d'inférence risque de choisir systématiquement le *premier* fait de la base correspondant au même chemin (ie. le même arc ou la même arête du graphe). Votre agent va donc être vite bloqué dans un cul de sac ou bien tourner en boucle. Nous allons corriger ça par la suite.

Exo 13 : Indiquez au moteur d'inférence que vous utilisez la stratégie aléatoire dans la sélection des règles actives de l'agenda. Pour cela, il faut utiliser la commande `(set-strategy random)` directement dans l'interpréteur de CLIPS. De manière alternative, il est souvent possible d'indiquer cette option dans les préférences en passant par le menu de l'IDE CLIPS. Testez à nouveau votre programme et indiquez en commentaires le changement que vous observez dans son exécution. Cela répond-il au problème soulevé dans l'exercice précédent ?

Exo 14 : Ajoutez une règle permettant de mettre fin au jeu dès lors qu'une condition est rencontrée (par ex. si le magot dépasse une certaine valeur maximale précisée dans la base de faits). Cette règle est indispensable car l'utilisateur ne peut plus indiquer qu'il souhaite terminer le programme.

N'oubliez pas que vous pouvez exécuter le code pour un nombre limité de règles afin d'éviter un programme qui tourne à l'infini, en utilisant la commande `(run N)` où N est le nombre de règles à exécuter au maximum.

Partie 4 - quelques développements possibles

Les exercices précédents permettent d'avoir une version très simple du jeu avec un agent autonome mais pas très malin. Voilà un certain nombre de développements possibles, à vous de choisir ce que vous souhaitez faire et jusqu'où vous comptez aller. Vous vous rendrez compte que certaines pistes sont naturellement plus difficiles à réaliser que d'autres. Elles sont données par ordre de difficulté croissant. Il n'est bien sûr pas question de réaliser *toutes* ces pistes.

- Ajouter comme nouveau type de trésor des objets magiques qui permettent d'améliorer les compétences de l'aventurier. Vous pouvez ajouter de nouvelles compétences en plus de la force (agilité, ruse...) et ajouter des épreuves/pièges de différents types dans le labyrinthe.
- Ajouter une part d'aléatoire dans le traitement des rencontres avec les monstres. La commande `(random 1 N)` retourne un valeur entière aléatoire entre 1 et N. Vous pouvez l'utiliser dans la résolution des combats ou pour d'autres raisons que vous imaginerez.
- Ajouter un système de clefs ouvrant certaines portes permettant de passer (ou non) d'un lieu à un autre. Il devient alors nécessaire de trouver les bonnes clefs dans le bon ordre pour progresser dans le labyrinthe.

— Ajouter une mémoire : un agent qui revient constamment sur ces pas sans raison n'a pas un comportement très rationnel à priori. Vous pouvez résoudre ce problème en partie en ajoutant une forme de mémoire à l'aventurier. Ainsi, vous pouvez ajouter des faits `memoire` qui enregistre le dernier lieu visité et empêcher qu'il soit choisi parmi les destinations possibles dans la règle `choix_direction`. Il ne faut bien sûr pas oublier de supprimer ces faits au fur et à mesure sous peine de totalement bloquer l'agent.
— Ajouter un système d'indices permettant à l'aventurier de mieux se déplacer dans le labyrinthe. Comme nous l'avons vu en cours, les monstres peuvent ainsi faire du bruit perceptible dans les lieux proches de celui où il se trouve. Ces faits `indices` peuvent être utilisés pour inciter l'aventurier à rebrousser chemin vers des lieux plus sûrs. Le problème est de ne pas finir bloqué, incapable de bouger. On peut aussi imaginer pouvoir développer un mécanisme de déduction analogue à ce qui a été vu en cours : si plusieurs indices concordent pour indiquer qu'un lieu pourrait contenir un monstre, alors cela confirme la dangerosité du lieu (et donc rend plus sûrs les autres lieux).
— Vous pouvez bien entendu rendre les autres agents (monstres) autonomes et donc capables de se déplacer, rendant le jeu totalement dynamique et donc moins prévisible et probablement plus difficile.

Ce qu'il faut rendre

Le projet doit être rendu au plus tard le samedi **9 mai à 12h**. Le rendu se fait via moodle mais un envoi par email est possible si vous ne pouvez faire autrement. Il faut rendre deux documents :

1. Le fichier source en CLIPS dont le nom est : `votrenom_projetIA.clp` (par ex. : `velcin_projetIA.clp`). Votre code doit être commenté.
2. Un rapport au format .pdf de *quelques pages* qui décrit votre projet. Il s'agit donc de reformuler brièvement les consignes, expliquer votre mise en œuvre, les résultats obtenus (avec des exemples d'exécution) et une conclusion où vous pouvez expliquer les problèmes rencontrés et les solutions que vous y avez trouvées. N'oubliez pas d'indiquer explicitement votre nom en tête du document.

Le travail est **individuel** mais, dans le cas où vous auriez été aidé ou si une partie du travail a été réalisé avec un autre étudiant, il faut l'indiquer explicitement. Si vous vous êtes aidé de ressources sur Internet, il faut également le mentionner.

Bonne chance !