TP2 : Conception d'un jeu intelligent

(activité d'équipe)

Travail Pratique : Concevoir d'un jeu en utilisant une technique d'IA

Ce travail compte pour 10 % de la note finale.
Échéance : 10 mars 2016 (23h59).

Objectifs d'apprentissage

Les objectifs d'apprentissage reliés à ce travail sont les suivants :

- identifier un problème d'intelligence artificielle ;
- analyser un problème de recherche dans un espace d'états ;
- choisir une technique de recherche appropriée dans un contexte donné ;
- implanter une solution en utilisant une approche déclarative.

Travail à faire

Le travail consiste à développer en équipes de 2 à 4 personnes un jeu avec une technique d'intelligence artificielle. La première chose à faire est de choisir un jeu quelconque, excepté les jeux du taquin et du tic-tac-toe, de préférence avec des règles simples. Voici quelques exemples de jeux qui peuvent être choisis : puissance 4, go, dames, solitaire, etc. Le jeu peut se jouer en duel ou en individuel. Le but est ensuite de programmer le jeu et de créer un joueur intelligent avec l'ordinateur. Pour cela, vous devez suivre la démarche de résolution présentée dans le cadre du cours (recherche intelligente dans un espace d'états). Les autres approches de résolution utilisées ne seront pas considérées.

Vous devez donc analyser le problème à résoudre en utilisant la résolution de problèmes par espace d'états et construire votre programme à l'aide du langage Prolog en utilisant une des techniques de recherche heuristiques vues en cours (Hill-Climbing, Meilleur d'abord, A*, Minimax et AlphaBêta).

1. **(10 %)** Tout d'abord, vous devez présenter le jeu choisi et expliquer les règles générales. Cette étape est importante pour que l'équipe de correction comprenne ce que vous avez choisi et réalisé.

2. **(30%)** Vous devez modéliser le problème, c'est-à-dire décrire ce que vous vous proposez de résoudre comme dans les exemples présentés en cours. Expliquez l'état initial, l'état final, les mouvements autorisés, la ou les fonctions heuristiques, la technique de recherche utilisée ainsi que les résultats attendus. Illustrez sur des exemples.

3. **(30%)** Vous devez maintenant implanter votre solution en langage Prolog, que vous avez proposée en (2). Les annexes vous donnent des exemples de ce qui est attendu. Cela vous donnera une idée du type d'interface que vous pouvez produire, sachant que cet aspect n'est pas primordial pour ce travail. L'accent est mis sur la technique de résolution de problèmes. Vous aurez peut-être à proposer une interface de jeu adaptée. Le programme doit être documenté (les prédicats doivent être définis dans le programme sous forme de commentaires).

4. **(20%)** Discutez les résultats obtenus (utilisez et donnez des jeux d'essais pour montrer les résultats obtenus par votre programme). Les buts que vous vous êtes fixés sont-ils atteints ? Discutez aussi des limites de l'heuristique implémentée. Vous pouvez proposer plusieurs heuristiques et expliquer leur performance.

5. **(10%)** Rédigez votre rapport en intégrant les 4 parties ci-dessus en soignant l'expression écrite et sa présentation. Ajoutez une page couverture (titre, auteurs, date, etc.), une introduction (but du travail, ce que vous avez voulu faire, ce que vous avez vraiment réalisé), une conclusion (résumé du travail accompli et ce que vous auriez pu ou aimé faire de plus) et, s'il y a lieu, une bibliographie.

Modalités de remise de ce travail

Avant de remettre votre travail, il est suggéré de vous autoévaluer à l'aide des deux grilles fournies dans cet énoncé et d'ajuster votre travail au besoin.

La date limite de remise du travail est le 10 mars 2016 (23h59). Tout travail remis en retard ne sera pas évalué et recevra la note 0, à condition qu'il y ait eu une entente préalable avec votre professeure.

La remise de ce travail se fait via le Portail des cours dans la rubrique Évaluation et résultats. Une boite de dépôt est prévue à cet effet. Cette boite accepte seulement les fichiers au format PDF et PL. N'oubliez pas de créer votre équipe de travail avant la date limite du 3 mars 2016 (23h59).

Grille d'auto-évaluation

Description du jeu choisi (10%)				
Présentation du jeu	Le but du jeu est présenté ainsi que des exemples.	Le but du jeu est présenté ou des exemples sont donnés.	Les explications et/ou les illustrations données sont incorrectes ou incomplètes.	Le but du jeu n'est pas présenté et aucun exemple n'est donné.
Description des règles	Les règles sont expliquées et illustrées.	Les règles sont expliquées ou illustrées.	Les explications et/ou les illustrations données sont incorrectes ou incomplètes.	Les règles ne sont ni expliquées, ni illustrées.
Modélisation du problème et de la solution (30%)				
Description du problème	Tous les éléments du problème sont expliqués et illustrés en utilisant l'approche par espace d'états.	Les éléments du problème sont expliqués ou illustrés en utilisant l'approche par espace d'états.	Les explications et/ou les illustrations données sont incorrectes ou incomplètes.	Les éléments du problème ne sont ni expliqués, ni illustrés.
Description de la solution	Tous les éléments de la solution sont expliqués et illustrés.	Les éléments de la solution sont expliqués ou illustrés.	Les explications et/ou les illustrations données sont incorrectes ou incomplètes.	Les éléments du problème ne sont ni expliqués, ni illustrés.
Implantation (30%)				
Fonctionnement	Le programme compile sans erreur et joue correctement.	Le programme compile sans erreur et joue correctement.	Le programme compile sans erreur mais joue incorrectement.	Le programme ne compile pas sans erreur.
Documentation	Les prédicats importants sont expliqués et illustrés.	Les prédicats importants sont expliqués ou illustrés.	L'explication et/ou l'illustration donnée est incorrecte.	Les prédicats importants sont ni expliqués, ni illustrés.
Résultats et discussion (20%)				
Discussion des jeux d'essais	Plusieurs jeux d'essais sont présentés et les résultats sont discutés.	Plusieurs jeux d'essais sont présentés mais les résultats ne sont pas discutés.	Un seul jeu d'essai est présenté et discuté.	Pas de jeu d'essai ou un seul jeu d'essai est présenté mais non discuté.
Avantages et limites	Plusieurs avantages et limites sont proposés.	Un avantage et une limite sont proposés.	Aucun avantage ou aucune limite n'est proposé.	Aucun avantage, ni aucune limite n'est proposé.
Améliorations possibles	Une amélioration possible est proposée et expliquée.	Une amélioration possible est proposée mais non expliquée.	Aucune amélioration n'est proposée.	
Appréciation globale (10%)				
Expression écrite	Le rapport ne contient aucune faute (vocabulaire, grammaire, syntaxe, etc.).	Le rapport ne contient pas plus d'une dizaine de fautes (vocabulaire, grammaire, syntaxe, etc.).	Le rapport ne contient pas plus de 5 fautes par page (vocabulaire, grammaire, syntaxe, etc.).	Le rapport contient plus de 5 fautes par page (vocabulaire, grammaire, syntaxe, etc.).
Présentation du rapport	Le format proposé est respecté. Le rapport est paginé.	Le rapport n'est pas paginé, il manque un élément du format.	Le rapport n'est pas paginé. Il manque 2 éléments du format.	Il y a plus de 2 éléments du format qui ont été oubliés.

ANNEXES : EXEMPLES DE PROGRAMMES PROLOG

EXEMPLE #1 : JEU DU TIC-TAC-TOE

https://www.youtube.com/watch?v=LCJB3h0dmR0

EXEMPLE #2 : JEU DU TAQUIN

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% resolveur du taquin (Technique du Hill-Climbing)
%
% La question a poser est : ?- resolveur( Deplacements ).
% Deplacements donne tous les deplacements a faire pour atteindre la situation finale
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

resolveur(Deplacements ) :-
        configuration_initiale( CI),
        configuration_finale(CF),
        resoudre_hill_climbing(CI, CF, [CI], Deplacements ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Hill Climbing
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
resoudre_hill_climbing( Configuration, Configuration, _, [] ).
resoudre_hill_climbing( Configuration, ConfigurationFinale, Deja_generees, [Deplacement | Deplacements] ) :-
        trouver_deplacements_legaux( Depls_Possibles, Configuration ),
        generer_configurations( Depls_Possibles, Configuration, ListeConfigurations ),
        valider( ListeConfigurations, Depls_Possibles, Deja_generees, ListeConfsValides, DeplsValides ),
        valeur_HC_tous( ListeConfsValides, ConfigurationFinale, ListeValeurs ),
        faire_liste( ListeValeurs, DeplsValides, ListeConfsValides, ListeConfsValuees ),
        trier( ListeConfsValuees, ListeConfsTriees ),
        member( [ V, Deplacement, NouvelleConf ], ListeConfsTriees ),
        resoudre_hill_climbing( NouvelleConf, ConfigurationFinale, [NouvelleConf | Deja_generees], Deplacements ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Predicats necessaires au jeu du taquin
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Situation de depart et situation a atteindre
configuration_initiale([1,*,4,8,3,5,7,2,6]).
configuration_finale([1,4,5,8,3,*,7,2,6]).

% Les deplacements possibles de l'asteristique
deplacer( deplacer_vers_gauche, De, A ) :- A is De - 1.
deplacer( deplacer_vers_haut, De, A ) :- A is De - 3.
deplacer( deplacer_vers_droite, De, A ) :- A is De + 1.
deplacer( deplacer_vers_bas, De, A ) :- A is De + 3.

% Les contraintes du jeu (ne pas sortir des 9 positions possibles
contraintes( deplacer_vers_gauche, [1, 4, 7] ).
contraintes( deplacer_vers_haut, [1, 2, 3] ).
contraintes( deplacer_vers_droite, [3, 6, 9] ).
```

```
contraintes( deplacer_vers_bas, [7, 8, 9] ).

% Determine le premier deplacement possible (gauche, haut, droit, bas)
trouver_deplacement_legal( Deplacement, Configuration ) :-
        position_element( Configuration, '*', P),
        contraintes( Deplacement, C),
        not( member( P, C ) ).

% Determine tous les deplacements legaux qui s'appliquent a une configuration
trouver_deplacements_legaux( Deplacements, Configuration ) :-
        findall( X, trouver_deplacement_legal( X, Configuration ), Deplacements ).

% Permet de generer une nouvelle configuration (C2) a partir d'une autre (C1) en appliquant Deplacement
generer( Deplacement, C1, C2 ) :-
        position_element( C1, '*', P ),
        deplacer( Deplacement, P, P1 ),
        element_position( C1, Element, P1 ),
        remplacer( P1, '*', C1, C3 ),
        remplacer( P, Element, C3, C2 ).

% Determine toutes les configurations obtenues a partir d'une configuration en appliquant la liste des deplacements
generer_configurations( [], Configuration, [] ).
generer_configurations( [ Deplacement | Deplacements ], Configuration, [ Config | Configs ] ) :-
        generer( Deplacement, Configuration, Config ),
        generer_configurations( Deplacements, Configuration, Configs ).

% Si une configuration a deja ete generee, on la rejette ainsi que le deplacement associe
valider( [], _, _, [], [] ).
valider( [ C | Configurations ], [ D | Deplacements ], Deja_generees, [C | Configs], [D | Depls] ) :-
        not( member( C, Deja_generees ) ),!,
        valider( Configurations, Deplacements, Deja_generees, Configs, Depls ).
valider( [ C | Configurations ], [ D | Deplacements ], Deja_generees, Configs, Depls ) :-
        valider( Configurations, Deplacements, Deja_generees, Configs, Depls ).

% Calcul de la valeur de la fonction heuristique
valeur_HC_tous( [], _, [] ).
valeur_HC_tous( [C|Configs], CF, [V|Valeurs] ) :-
        valeur_HC( C, CF, V ),
        valeur_HC_tous( Configs, CF, Valeurs ).

valeur_HC( [], [], 0 ) :- !.
valeur_HC( [X | Xs], [X | Ys], V_HC ) :-
        valeur_HC( Xs, Ys, V_HC ), !.
valeur_HC( [X | Xs], [Y | Ys], V_HC ) :-
        valeur_HC( Xs, Ys, V_HC1 ), V_HC is V_HC1 + 1.

% faire une seule liste de triplet a partir de trois listes
faire_liste( [], [], [], [] ).
faire_liste( [X | Xs], [Y | Ys], [Z | Zs], [[X,Y,Z] | Reste] ) :-
        faire_liste( Xs, Ys, Zs, Reste ).

% trier (algorithme de tri QuickSort)
trier( [], [] ).
trier( [X|Reste], ListeTriee ) :-
        partitionner( Reste, X, Les_Petits, Les_Grands ),
        trier( Les_Petits, Ps ),
```

```
        trier( Les_Grands, Gs ),
        append( Ps, [X | Gs], ListeTriee ).

partitionner( [], _, [], [] ).
partitionner( [ [X1,X2,X3] | Xs], [Y1,Y2,Y3], [ [X1,X2,X3] | Ps], Gs ) :-
        X1 < Y1, !,
        partitionner( Xs, [Y1,Y2,Y3], Ps, Gs).
partitionner( [ [X1,X2,X3] | Xs], [Y1,Y2,Y3], Ps, [ [X1,X2,X3] | Gs] ) :-
        partitionner( Xs, [Y1,Y2,Y3], Ps, Gs).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Predicats de service
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% position_element/3 : recherche la position d'un element dans une liste
position_element( [ Element | _ ], Element, 1 ) :- !.
position_element( [ _ | Reste ], Element, P ) :- position_element( Reste, Element, NP ), P is NP + 1.

% element_position/3 : recherche l'element a une position donnee dans une liste
element_position( [ Element | _ ], Element, 1 ) :- !.
element_position( [ _ | Reste ], Element, P ) :- NP is P - 1, element_position( Reste, Element, NP).

% remplacer/3 : remplace un enieme element d'une liste par un autre
remplacer( 1, Nouveau, [ Element | Reste ], [ Nouveau | Reste ] ) :- !.
remplacer( N, Nouveau, [ Element | Reste ], [ Element | Final ] ) :-
            NP is N - 1, remplacer( NP, Nouveau, Reste, Final ).
```