Compléments d'informatique

Projet 3 : intelligence artificielle

26 novembre 2024

L'objectif concret de ce projet est d'implémenter deux agents intelligents pour le jeu du morpion. L'objectif pédagogique est de vous faire utiliser des structures de données de type liste et dictionnaire. Vous en apprendrez également un peu sur certaines techniques d'intelligence artificielle. Ce projet est à réaliser **seul ou à deux**. La date limite de remise est précisée sur eCampus et sur Gradescope.

1 Principe du jeu

Le morpion, aussi appelé Tic-tac-toe ou oxo en Belgique, est un jeu à deux joueurs, au tour par tour, basé sur une grille 3×3 dans sa version la plus simple qui sera considérée ici. À chaque joueur est attribué un symbole, `X` ou `O`. Chacun des joueurs à son tour place son symbole sur une case vide de la grille et le premier réalisant un alignement horizontal, vertical ou diagonal de son symbole a gagné. Si la grille est pleine sans aucun alignement, la partie se termine par une égalité (partie nulle). On considérera dans ce projet que le joueur `X` est toujours celui qui commence. A partie du contenu de la grille, il est donc possible de déterminer à qui est le tour [1]. Dans la suite, pour simplifier, on supposera que les positions dans la grille sont numérotées de 0 à 9, de haut en bas et de gauche à droite (par exemple, la case en bas à gauche est la case 6).

2 Agents intelligents

On vous fournit un moteur de jeu complet, incluant un agent humain et un agent autonome jouant de manière purement aléatoire. Votre objectif pour ce projet est d'implémenter deux agents autonomes plus intelligents que l'agent aléatoire, qui seront représentatifs de deux

1. Si le nombre de `X` est pair, c'est à `X` de jouer, sinon, c'est à `O`.

type d'intelligence artificielle. Le premier déterminera de manière exacte le coup optimal à effectuer en faisant l'hypothèse que l'autre joueur joue également de manière optimale. Le second agent incorporera un mécanisme lui permettant de s'améliorer au fur et à mesure des parties en s'adaptant au jeu de son adversaire. Le principe de ces deux agents est détaillé ci-dessous.

2.1 Algorithme minimax

Face à un grille, le premier agent déterminera le coup qui lui permettra d'obtenir le meilleur *score* possible en faisant l'hypothèse que son adversaire utilisera la même stratégie. Dans ce projet, le score associé à une partie pour un joueur donné vaudra $+1$ dans le cas où la partie est gagnée, -1 si la partie est perdue et 0 si la partie est nulle. Le score associé à une grille b, noté $S(b)$, est alors défini comme le score maximal que le joueur dont c'est le tour étant donné b peut espérer obtenir à partir de b sachant que son adversaire jouera lui aussi de manière optimale. Sous cette hypothèse, on peut calculer $S(b)$ pour toute grille de manière récursive par l'équation suivante :

$$S(b) = \max_{m \in \mathcal{M}(b)} -S(\texttt{next}(b, m)) \tag{1}$$

où $\mathcal{M}(b)$ est l'ensemble des coups possibles étant donné la grille b (c'est-à-dire l'ensemble des cases vides dans b) et $\texttt{next}(b, m)$ est la nouvelle grille obtenue après avoir joué m. Le signe moins vient du fait qu'on change de joueur à chaque coup joué. Le cas de base de la récurrence correspond à une partie terminée : $S(b)$ vaut -1 si b contient trois symboles identiques alignés (-1 parce que celui dont ça devrait être le tour a perdu), 0 si la grille est pleine.

Une fois les valeurs de $S(b)$ calculées, on peut déterminer le coup optimal m^* à partir d'une grille b pour le joueur dont c'est le tour, en calculant :

$$m^* = \arg \max_{m \in \mathcal{M}(b)} -S(\texttt{next}(b, m). \tag{2}$$

En général, une implémentation naïve de la récurrence (1) serait potentiellement très couteuse en temps de calcul. Cela demanderait en effet d'explorer toutes les parties possibles à partir de chaque grille b, ce qui amènera à recalculer $S(b)$ pour une même grille b plusieurs fois puisqu'une grille b peut être atteinte à partir d'une autre de plusieurs manières possibles en fonction de l'ordre des coups. Pour éviter de recalculer plusieurs fois chaque $S(b)$, on stockera ces valeurs dans une table de hachage, qui sera consultée avant de lancer la récursivité (voir les conseils d'implémentation à la section 4).

L'algorithme tel que décrit correspond à ce qu'on appelle en théorie des jeux et en IA l'algorithme minimax [2].

2. Voir par exemple https://fr.wikipedia.org/wiki/Algorithme_minimax si vous voulez en savoir plus.

2.2 Apprentissage par renforcement

L'idée du second agent est de partir d'un agent choisissant ses coups au hasard et d'améliorer progressivement ses performances sur base de parties réalisées contre un autre joueur. Pour ce faire, cet agent maintiendra pour chaque grille b une liste de paires $(m, \hat{S}(b,m))$, où m est une position dans $\mathcal{M}(b)$ et $\hat{S}(b,m)$ sera la moyenne des scores obtenus par l'agent lorsqu'il a joué le coup m à partir de b. Ainsi, plus le score $\hat{S}(b,m)$ sera élevé, et plus il sera intéressant de jouer le coup m avec la grille b. Initialement, toutes les valeurs de $\hat{S}(b,m)$ seront fixées à 0 et on supposera que chaque coup possible aura été joué une seule fois.

L'agent pourra être dans deux modes lors d'une partie : en mode apprentissage ou en mode exploitation.

En mode *exploitation*, l'agent ne mettra pas à jour les valeurs de $\hat{S}(b,m)$ et jouera pour une grille b le coup donné par :

$$m^* = \arg \max_{m \in \mathcal{M}(b)} \hat{S}(b,m). \tag{3}$$

Dans le cas où plusieurs coups auraient le même score (ce qui est le cas initialement), l'agent devra choisir un coup au hasard parmi ceux de score maximal.

En mode *apprentissage* par contre, l'agent mettra à jour les valeurs de $\hat{S}(b,m)$ à chaque fin de partie et il jouera ses coups selon la stratégie suivante :

- avec une probabilité ϵ, il choisira un coup au hasard dans $\mathcal{M}(b)$,
- avec une probabilité $1-\epsilon$, il choisira son coup selon l'équation (3).

Cette stratégie permet de diriger les coups de l'agent en priorité vers ceux qui ont déjà permis d'obtenir un score élévé tout en maintenant une exploration de tous les coups pour éventuellement en identifier de meilleurs. On fixera la valeur de ϵ à 0.25.

L'algorithme décrit ici est une version très simplifiée de ce qu'on appelle en IA l'apprentissage par renforcement [3].

3 Implémentation

Nous vous fournissons une implémentation de table de hachage dans les fichiers `Dict.c/.h` et une implémentation de liste liée dans les fichiers `LinkedList.c/.h` que vous êtes libres d'utiliser, sans néanmoins les modifier. Les fonctions de ces deux implémentations sont documentées dans les fichiers `.h` respectifs et ne seront pas décrites en détail ci-dessous.

Nous vous fournissons également un module `board.c/.h` pour gérer les grilles, un module `agent.c/.h` pour gérer des agents et un fichier `main.c` pour faire des tests. Vous devrez vous

3. Voir par exemple https://fr.wikipedia.org/wiki/Apprentissage_par_renforcement si vous voulez en savoir plus.

implémenter les modules `aiagent.c/.h` et `rlagent.c/.h` implémentant respectivement les deux agents décrits ci-dessus. Tous ces fichiers sont décrits en détails ci-dessous.

3.1 Fichiers `board.c/.h`

Les fichiers `board.c/.h` fournissent un type `Board` pour représenter une grille. Il correspond simplement à une chaîne de caractères de longueur 9 (terminée par un `\0`) reprenant dans l'ordre les symboles présents dans la grille (`' '` pour une case vide). Un type `Player` est également défini, correspondant aux constantes `X`, `O`, et `E`, ce dernier servant à identifier un match nul ou une case vide dans la suite. Un type `Move` représentant les entiers de 0 à 9 est utilisé pour représenter un coup.

Fonctions de l'interface. Les fonctions fournies par l'interface sont les suivantes :
`Board boardMakeEmpty(void)` : renvoie une grille vide.

`void boardFree(void)` : libère la grille de la mémoire.

`Board boardCopy(Board b)` : renvoie une copie de la grille b.

`void boardPrint(Board b)` : affiche la grille dans le terminal.

`Player boardWin(Board b)` : renvoie le joueur gagnant si la grille est gagnante, `E` sinon.

`bool boardIsFull(Board b)` : renvoie `true` si la grille est pleine, `false` sinon.

`Board boardNext(Board b, Move m, Player p)` : attribue la case `m` au joueur `p` dans la grille `b` qui est modifiée. Arrête le programme si la case n'est pas vide.

`bool boardValidMove(Board b, Move m)` : renvoie `true` si la case `m` est vide, `false` sinon.

`bool boardGetPlayer(Board b)` : renvoie le joueur dont c'est le tour (`E` si la grille est pleine).

3.2 Fichiers `agent.c/.h`

Ces fichiers contiennent la définition d'un type (opaque) `Agent` représentant un agent (c'est-à-dire un joueur potentiel), ainsi que la fonction permettant de lancer une partie entre deux agents (`agentPlayGame`). La fonction `agentCreate`, que vous utiliserez pour créer vos deux agents intelligents, simule un mécanisme d'héritage de classe. Elle permet de définir un agent avec ses propres méthodes `play`, `end` et `freeData`. Un agent peut incorporer des

données, qui sont accessibles via les fonctions `agentGetData` et `agentSetData`.

Fonctions de l'interface. Les fonctions fournies par l'interface sont les suivantes :

```
Agent *agentCreate(char *name, Move (*play)(Agent *, Board),
                   void (*end)(Agent *, Board, Player),
                   void (*freeData)(void *))
```

crée un agent. `name` est une chaîne de caractère contenant le nom de l'agent qui sera utilisée pour l'affichage. `play` est une fonction prenant en argument l'agent, une grille et un joueur (`X` ou `O`) et renvoyant le coup joué par l'agent. `end` est une fonction appelée lorsque la partie est terminée. Ses arguments sont l'agent, la grille finale et le joueur gagnant s'il y en a un, `E` sinon. `freeData` est une fonction qui sera appelée pour libérer les données stockées dans l'agent via la fonction `agentSetData`. Par défaut, l'agent correspond au joueur `X`.

`void agentFree(Agent *agent)` : libère l'agent de la mémoire (la fonction `freeData` sera appelée sur les données).

`void agentSetPlayer(Agent *agent, Player p)` : change le joueur (`X` ou `O`) que joue l'agent.

`Player AgentGetPlayer(Agent *agent)` : renvoie le joueur que joue l'agent.

`char *agentGetName(Agent *)` : renvoie le nom du joueur.

`void agentSetData(Agent *agent, void *data)` : ajoute des données à l'agent.

`void *agentGetData(Agent *agent)` : renvoie les données de l'agent.

`Agent *createHumanAgent(void)` : crée un agent humain. La fonction `play` de cet agent demande la position à jouer sur le terminal.

`Agent *createRandomAgent(void)` : crée un agent aléatoire. Cet agent joue un coup aléatoire à chaque tour.

`Player agentPlayGame(Agent *agentX, Agent *agentO, bool verbose)` : joue une partie entre `agentX` et `agentO` (`agentX` commençant la partie) et renvoie le gagnant (`E` en cas d'égalité). Si `verbose` est `true`, des messages sont affichés qui montrent l'évolution de la partie. Sinon, aucun message n'est affiché.

3.3 Fichiers `aiagent.c/.h`

Ce fichier devra implémenter l'agent intelligent utilisant l'algorithme minimax (voir ci-dessus). Aucune nouveau type n'est défini et l'interface ne contient que deux fonctions :
`Agent *createAiAgent(void)` : crée un agent utilisant l'algorithme minimax vu plus haut.

`int getMinimaxScore(Agent *agent, Board b)` : renvoie la valeur de $S(b)$ telle que définie plus haut. Cette fonction a pour seul but de faciliter le test de votre code. Elle ne sera appelée qu'avec des grilles de jeu valide, c'est-à-dire qui peuvent être générée lors d'une partie.

3.4 Fichiers `rlagent.c/.h`

Ce fichier devra implémenter l'agent intelligent utilisant l'apprentissage par renforcement (voir ci-dessus). Aucune nouveau type n'est défini et l'interface ne contient que trois fonctions :
`Agent *createRlAgent(void)` : crée un agent utilisant l'apprentissage par renforcement vu plus haut. Initiallement, l'agent créé devra être en mode apprentissage.

`void setTrainingModeRlAgent(Agent *agent, bool training)` : permet de mettre l'agent est mode apprentissage (`training` à `true`) ou en mode exploitation (`training` à `false`).

`float getMoveScoreRlAgent(Agent *agent, Board b, Move m)` : renvoie la valeur de $\hat{S}(b, m)$ telle que définie plus haut. Cette fonction a pour seul but de faciliter le test de votre code. Elle ne devrait être appelée à aucun endroit du code. Elle ne sera appelée qu'avec des grilles b et des mouvements m valides.

3.5 Fichier `main.c`

Le fichier `main.c` et le `Makefile` fournis créeront un exécutable appelé `tictactoe`. L'option `-s agent1 agent2` permet de lancer une seule partie entre deux agents, choisis parmi `human`, `random`, `ai` (pour l'algorithme minimax) et `rl` (pour l'apprentissage par renforcement). Si l'un des deux est `human`, vous pourrez ainsi affronter l'autre en rentrant vos coups sur le terminal. L'option `-t N agent1 agent2` permet d'exécuter N parties mettant en compétition les deux agents et d'afficher le nombre de parties gagnées par chaque joueur. Si l'un des agents (ou les deux) sont du type `rl`, celui-ci sera en mode apprentissage et devrait donc améliorer ses performances au cours des parties. Il est également possible de fournir une option `-p M [trainingagent]`. Si elle est fournie, l'agent `rl` sera confronté à `trainingagent` s'il est fourni ou bien l'agent qu'il affrontera ensuite sinon, pendant `M` parties en mode apprentissage avant d'exécuter les autres parties en mode exploitation.

Quelques exemples de tests intéressantes :

`./tictactoe -s human human` : pour jouer à deux. C'est la commande lancée par `make run`.
`./tictactoe -s human ai` : permet d'affronter le premier agent intelligent. Au mieux, vous pourrez faire une égalité face à lui.
`./tictactoe -s human rl` : permet d'affronter l'agent `rl` sans entraînement (revient à affronter l'agent aléatoire).
`./tictactoe -s human rl -p 10000 ai` : permet d'affronter l'agent `rl` après qu'il se soit entraîné en faisant 10000 parties avec l'agent `ai`.
`./tictactoe -t 1000 random ai` : met l'agent `ai` en compétition avec l'agent aléatoire sur 1000 parties.
`./tictactoe -t 1000 random rl -p 100000` : met l'agent `rl` en compétition avec l'agent aléatoire après qu'il se soit entraîné contre lui sur 100000 parties. Fera-t-il mieux que l'agent `ai` face à l'agent `random` ?
`./tictactoe -t 1000 random rl -p 100000 ai` : met l'agent `rl` en compétition avec l'agent aléatoire après qu'il se soit entraîné contre l'agent `ai` sur 100000 parties. Est-ce que cet agent `rl` fait aussi bien que l'agent `ai` contre l'agent `random` ?

4 Conseils d'implémentation

Pour chacun des agents, vous devez définir les fonctions `play`, `end` et `freeData` à donner en argument à la fonction `agentCreate` et décider ce que vous allez placer dans les données en utilisant les fonctions `agentSetData` et `agentGetData`.

Dans le cas de `aiagent.c`, les données devraient contenir une table de hachage qui contiendra pour chaque grille b la valeur du score associé $S(b)$. Il peut être intéressant également pour facilier l'implémentation de `play` de stocker en plus également le coup optimal m^* dans la même table. Pour calculer $S(b)$, il suffit d'implémenter l'équation récursive (1) avec la modification importante suivante : avant de calculer $S(b)$, vous devez d'abord vérifier que sa valeur n'est pas déjà stockée dans la table. Si c'est le cas, vous pouvez renvoyer cette valeur directement. Si ce n'est pas le cas, vous calculez cette valeur en faisant les appels récursifs requis et vous stockez cette valeur dans la table de hachage avant de la renvoyer. Ce mécanisme permet de ne pas récalculer plusieurs fois la même valeur et devrait diminuer fortement les temps de calcul.

Dans le cas de `rlagent.c`, les données devraient contenir également une table de hachage qui stockera pour chaque grille b la liste des coups possibles avec l'estimation actuelle des scores $\hat{S}(b, m)$. Pour pouvoir mettre à jour cette moyenne, vous devrez probablement également stocker pour chaque coup combien de fois il a été utilisé dans le passé. En mode apprentissage, la fonction `play` devra utiliser les données pour stocker l'historique des coups joués par l'agent pour pouvoir mettre à jour les scores une fois la partie terminée. Cette mise à jour se fera dans la fonction `end`. Nous vous encourageons à utiliser le module

`LinkedList.c/.h` pour stocker la liste des coups dans la table de hachage et également pour stocker l'historique des coups précédemment joués.

Pour les deux agents, vous pouvez utiliser directement les grilles comme clés dans la table de hachage, ces dernières étant des chaînes de caractères. Vous pourriez être tenté de stocker les tables de hachage et l'historique des coups passés dans des variables globales, plutôt que d'utiliser les données de l'agent. Ce n'est cependant pas une bonne idée parce qu'on pourrait mettre en compétition deux agents du même type, ce qui entraînerait des problèmes de conflit s'ils utilisent tous les deux les mêmes variables globales.

5 Soumission

Vous devez soumettre (uniquement) les trois fichiers suivants sur Gradescope : `rapport.txt`, `aiagent.c`, et `rlagent.c`. Le rapport devra contenir les contributions des membres du groupes et le résultat de l'application de certaines commandes.

Le fait de soumettre votre projet suppose que vous attestez avoir respecté les règles en vigueur dans le cours en terme de plagiat rappelées dans le fichier rapport.txt et reprises aux slides 9 et 10 du cours théorique. Tout non respect de ces règles sera sanctionné.

Bon travail !