Projet N°2 : Démineur [1]

1. Introduction

Écrivez un programme d'IA capable de jouer au démineur.

A rendre le : **30/10/2020**

2. Contexte

Le démineur est un jeu de puzzle qui consiste en une grille de cellules, dont certaines contiennent des "**mines**" cachées. En cliquant sur une cellule qui contient une mine, la mine explose et l'utilisateur perd le jeu. Cliquer sur une cellule "**sûre**" (c'est-à-dire une cellule qui ne contient pas de mine) révèle un nombre qui indique combien de cellules voisines - où un voisin est une cellule qui se trouve à un carré à gauche, à droite, en haut, en bas ou en diagonale de la cellule donnée - contiennent une mine.

Dans ce jeu démineur 3x3 suivant, par exemple, les trois valeurs 1 indiquent que chacune de ces cellules a une cellule voisine qui est une mine. Les quatre valeurs 0 indiquent que chacune de ces cellules n'a pas de mine voisine.

	0	**0**
0	**1**	**1**
0	**1**	

Compte tenu de ces informations, un joueur logique pourrait conclure qu'il doit y avoir une mine dans la cellule inférieure droite et qu'il n'y a pas de mine dans la cellule supérieure gauche, car c'est seulement dans ce cas que les étiquettes numériques sur chacune des autres cellules seraient exactes.

Le but du jeu est de signaler (c'est-à-dire d'identifier) chacune des mines. Dans de nombreuses implémentations du jeu, y compris celle de ce projet, le joueur peut marquer une mine en cliquant avec le bouton droit de la souris sur une cellule (ou en cliquant à deux doigts, selon l'ordinateur).

3. Logique propositionnelle

Votre but dans ce projet sera de construire une AI qui peut jouer au Démineur. Rappelez-vous que les agents basés sur la connaissance prennent des décisions en tenant compte de leur base de connaissances, et en faisant des déductions basées sur cette connaissance.

Une façon de représenter les connaissances d'une AI sur un jeu de Démineur est de faire de chaque cellule une variable propositionnelle qui est vraie si la cellule contient une mine, et fausse sinon.

À quelles informations l'AI a-t-elle accès ? Eh bien, l'AI le sait à chaque fois qu'on clique sur une cellule sécurisée et elle peut voir le numéro de cette cellule. Considérez le tableau suivant du Démineur, où la cellule du milieu a été révélée, et les autres cellules ont été étiquetées avec une lettre d'identification pour les besoins de la discussion.

[1] https://cs50.harvard.edu/ai/2020

A	**B**	**C**
D	**1**	**E**
F	**G**	**H**

Quelles sont les informations dont nous disposons actuellement ? Il semble que nous sachions maintenant que l'une des huit cellules voisines est une mine. Par conséquent, nous pourrions écrire une expression logique comme celle ci-dessous pour indiquer qu'une des cellules voisines est une mine.

```
Or(A, B, C, D, E, F, G, H)
```

Mais nous savons en fait plus que ce que dit cette expression. La phrase logique ci-dessus exprime l'idée qu'au moins une de ces huit variables est vraie. Mais nous pouvons faire une déclaration plus forte que cela : nous savons qu'exactement une de ces huit variables est vraie. Cela nous donne une phrase logique propositionnelle comme celle ci-dessous.

```
Or(
    And(A, Not(B), Not(C), Not(D), Not(E), Not(F), Not(G), Not(H)),
    And(Not(A), B, Not(C), Not(D), Not(E), Not(F), Not(G), Not(H)),
    And(Not(A), Not(B), C, Not(D), Not(E), Not(F), Not(G), Not(H)),
    And(Not(A), Not(B), Not(C), D, Not(E), Not(F), Not(G), Not(H)),
    And(Not(A), Not(B), Not(C), Not(D), E, Not(F), Not(G), Not(H)),
    And(Not(A), Not(B), Not(C), Not(D), Not(E), F, Not(G), Not(H)),
    And(Not(A), Not(B), Not(C), Not(D), Not(E), Not(F), G, Not(H)),
    And(Not(A), Not(B), Not(C), Not(D), Not(E), Not(F), Not(G), H)
)
```

C'est une expression bien compliquée ! Et c'est juste pour exprimer ce que cela signifie pour une cellule d'avoir un 1 en elle. Si une cellule a un 2 ou un 3 ou une autre valeur, l'expression peut être encore plus longue.

Essayer d'effectuer une vérification de modèle sur ce type de problème aussi deviendrait rapidement insoluble : sur une grille 8x8, la taille utilisée par Microsoft pour son niveau Débutant, nous aurions 64 variables, et donc 2^64 modèles possibles à vérifier - beaucoup trop pour qu'un ordinateur puisse les calculer dans un délai raisonnable. Nous avons besoin d'une meilleure représentation des connaissances pour ce problème.

4. Représentation des connaissances

Nous allons plutôt représenter chaque phrase de notre connaissance de l'AI comme ci-dessous.

```
{A, B, C, D, E, F, G, H} = 1
```

Chaque phrase logique de cette représentation comporte deux parties : un ensemble de cellules du tableau qui sont impliquées dans la phrase, et un nombre, représentant le nombre de ces cellules qui sont des mines. La phrase logique ci-dessus indique que parmi les cellules A, B, C, D, E, F, G et H, une seule est une mine.

Pourquoi cette représentation est-elle utile ? En partie, elle se prête bien à certains types d'inférences. Examinons le jeu ci-dessous.

A	**B**	**C**
D	**E**	**F**
0	**G**	**H**

En utilisant les connaissances du nombre en bas à gauche, nous pourrions construire la phrase `{D, E, G} = 0` pour signifier que sur les cellules D, E et G, exactement 0 d'entre elles sont des mines. Intuitivement, nous pouvons déduire de cette phrase que toutes les cellules doivent être sûres. Par extension, chaque fois que nous avons une phrase dont le compte est égal à 0, nous savons que toutes les cellules de cette phrase doivent être sûres.

De même, considérez le jeu ci-dessous.

A	**B**	**C**
D	**E**	**F**
G	**H**	**3**

Notre AI construira la phrase `{E, F, H} = 3`. Intuitivement, on peut en déduire que tous les E, F et H sont des mines. Plus généralement, chaque fois que le nombre de cellules est égal au compte, nous savons que toutes les cellules de cette phrase doivent être des mines.

En général, nous voulons que nos phrases ne concernent que les cellules dont on ne sait pas encore si elles sont sûres ou si elles sont minées. Cela signifie qu'une fois que nous savons si une cellule est une mine ou non, nous pouvons mettre à jour nos phrases pour les simplifier et éventuellement tirer de nouvelles conclusions.

Par exemple, si notre AI connaissait la phrase `{A, B, C} = 2`, nous n'avons pas encore assez d'informations pour conclure quoi que ce soit. Mais si on nous disait que `C` est sans danger, nous pourrions supprimer C de la phrase, ce qui nous laisserait la phrase `{A, B} = 2` (ce qui, soit dit en passant, nous permet de tirer de nouvelles conclusions).

De même, si notre AI connaissait la phrase `{A, B, C} = 2`, et qu'on nous disait que C est une mine, nous pourrions retirer C de la phrase et diminuer la valeur du compte (puisque C est une mine qui a contribué à ce compte), ce qui nous donnerait la phrase `{A, B} = 1`. C'est logique : si deux de A, B et C sont des mines, et que nous savons que C est une mine, alors il doit y avoir un cas où, parmi A et B, exactement un d'entre eux est une mine.*

Si nous sommes encore plus intelligents, il y a un dernier type d'inférence que nous pouvons faire.

1	**1**	1
A	**B**	**C**
D	**2**	**E**

Considérez seulement les deux phrases que notre AI connaîtrait en se basant sur la cellule du milieu supérieure et la cellule du milieu inférieure. À partir de la cellule centrale supérieure, nous avons `{A, B, C} = 1`. Dans la cellule du milieu inférieure, nous avons `{A,`

`B, C, D, E} = 2`. Logiquement, nous pourrions alors déduire un nouvel élément de connaissance, à savoir que `{D, E} = 1`. Après tout, si deux des éléments A, B, C, D et E sont des mines, et qu'un seul des éléments A, B et C est une mine, il est logique qu'un des éléments D et E soit l'autre mine.

Plus généralement, chaque fois que nous avons deux phrases set1 = compte1 et set2 = compte2 où set1 est un sous-ensemble de set2, nous pouvons alors construire la nouvelle phrase set2 - set1 = compte2 - compte1. Considérez l'exemple ci-dessus pour vous assurer que vous comprenez pourquoi c'est vrai.

Ainsi, en utilisant cette méthode de représentation des connaissances, nous pouvons écrire un agent d'AI qui peut rassembler des connaissances sur le tableau du Démineur, et avec un peu de chance, sélectionner des cellules qu'il sait être sûres !

5. Dossier degrees.zip

Considérez le fichier minesweeper.zip. Dézippez le.

Une fois dans le répertoire du projet, lancez `pip3 install -r requirements.txt` pour installer le paquet Python requis (`pygame`) pour ce projet si vous ne l'avez pas déjà installé.

6. Compréhension

Il y a deux fichiers principaux dans ce projet : `runner.py` et `minesweeper.py`. `minesweeper.py` contient toute la logique du jeu lui-même et pour l'AI de jouer le jeu. `runner.py` a été implémenté pour vous, et contient tout le code pour exécuter l'interface graphique du jeu. Une fois que vous avez rempli toutes les fonctions requises dans `minesweeper.py`, vous devriez pouvoir exécuter `python runner.py` pour jouer au Démineur (ou laisser votre AI jouer pour vous) !

Ouvrons le fichier `minesweeper.py` pour comprendre ce qui est fourni. Il y a trois classes définies dans ce fichier, `Minesweeper`, qui gère le gameplay ; `Sentence`, qui représente une phrase logique qui contient à la fois un ensemble de `cellules` et un `compte` ; et `MinesweeperAI`, qui gère l'inférence des mouvements à effectuer en fonction de la connaissance.

La classe `Minesweeper` des dragueurs de mines a été entièrement mise en place pour vous. Notez que chaque cellule est une paire `(i, j)` où `i` est le numéro de la ligne (allant de `0` à la `hauteur - 1`) et `j` est le numéro de la colonne (allant de `0` à la `largeur - 1`).

La classe `Sentence` sera utilisée pour représenter des phrases logiques de la forme décrite dans la 4. `Représentation des connaissances.` Chaque phrase comporte un ensemble de cellules et un compte du nombre de ces cellules qui sont des mines. La classe contient également des fonctions `known_mines` et `known_safes` permettant de déterminer si certaines des cellules de la phrase sont connues pour être des mines ou pour être sûres. Elle contient également les fonctions `mark_mine` et `mark_safe` pour mettre à jour une phrase en réponse à de nouvelles informations sur une cellule.

Enfin, la classe `MinesweeperAI` va mettre en place une AI qui peut jouer au Démineur. La classe AI garde la trace d'un certain nombre de valeurs. `self.moves_made` contient l'ensemble de toutes les cellules sur lesquelles on a déjà cliqué, de sorte que l'AI sait qu'il ne faut pas les

sélectionner à nouveau. `self.mines` contient l'ensemble de toutes les cellules connues comme étant des mines. `self.safes` contient l'ensemble de toutes les cellules connues comme étant sûres. Et `self.knowledge` contient une liste de toutes les phrases que l'AI sait être vraies.

La fonction `mark_mine` ajoute une cellule à `self.mines`, de sorte que l'IA sait qu'il s'agit d'une mine. Elle passe également en boucle toutes les phrases dont l'IA a connaissance et informe chaque phrase que la cellule est une mine, de sorte que la phrase peut se mettre à jour en conséquence si elle contient des informations sur cette mine. La fonction `mark_safe` fait la même chose, mais pour les cellules sûres.

Les autres fonctions, `add_knowledge`, `make_safe_move` et `make_random_move`, sont à votre disposition !

7. Travail à faire

Compléter les mises en œuvre de la classe `Sentence` et de la classe `MinesweeperAI` dans `minesweeper.py`.

Dans la classe `Sentence`, complétez les implémentations de `known_mines`, `known_safes`, `mark_mine` et `mark_safe`.

- La fonction `known_mines` doit retourner un ensemble de toutes les cellules de `self.cells` qui sont connues pour être des mines.
- La fonction `known_safes` doit renvoyer un ensemble de toutes les cellules de `self.cells` qui sont connues pour être sûres.
- La fonction `mark_mine` doit d'abord vérifier si la cellule est l'une des cellules incluses dans la phrase.
    - Si la cellule est dans la phrase, la fonction doit mettre à jour la phrase de sorte que la cellule ne soit plus dans la phrase, mais qu'elle représente toujours une phrase logiquement correcte étant donné que cette cellule est connue pour être une mine.
    - Si la cellule n'est pas dans la phrase, alors aucune action n'est nécessaire.
- La fonction `mark_safe` doit d'abord vérifier si la cellule est l'une des cellules incluses dans la phrase.
    - Si la cellule est dans la phrase, la fonction doit mettre à jour la phrase de sorte que la cellule ne soit plus dans la phrase, mais qu'elle représente toujours une phrase logiquement correcte étant donné que la cellule est connue pour être sûre.
    - Si la cellule n'est pas dans la phrase, aucune action n'est nécessaire.

Dans la classe `MinesweeperAI`, complétez les implémentations de `add_knowledge`, `make_safe_move` et `make_random_move`.

- `add_knowledge` doit accepter une cellule (représentée par un tuple `(i, j)`) et son compte correspondant, et mettre à jour `self.mines`, `self.safes`, `self.moves_made`, et `self.knowledge` avec toute nouvelle information que l'AI peut déduire, étant donné que cette cellule est connue pour être une cellule sûre avec un compte de mines voisin.
    - La fonction doit marquer la cellule comme l'un des mouvements effectués dans le jeu.

    - La fonction doit marquer la cellule comme une cellule sûre, en mettant à jour toutes les phrases qui contiennent la cellule également.
    - La fonction doit ajouter une nouvelle phrase à la base de connaissances de l'AI, basée sur la valeur de la cellule et du compte, pour indiquer que les comptes des voisins de la cellule sont des mines. Veillez à n'inclure que les cellules dont l'état est encore indéterminé dans la phrase.
    - Si, sur la base d'une des phrases de `self.knowledge`, de nouvelles cellules peuvent être marquées comme sûres ou comme mines, alors la fonction devrait le faire.
    - Si, sur la base de l'une des phrases de `self.knowledge`, de nouvelles phrases peuvent être déduites (en utilisant la méthode du sous-ensemble décrite dans 4. **Représentation des connaissances**), alors ces phrases doivent être ajoutées à la base de connaissances également.
    - Notez qu'à chaque fois que vous modifiez les connaissances de votre AI, il est possible de tirer de nouvelles conclusions qui n'étaient pas possibles auparavant. Assurez-vous que ces nouvelles déductions sont ajoutées à la base de connaissances si cela est possible.
- `make_safe_move` doit renvoyer un coup `(i, j)` dont on sait qu'il est sûr.
    - Le coup retourné doit être sûr et qu'il ne s'agit pas d'un coup déjà effectué.
    - Si aucun coup sûr ne peut être garanti, la fonction doit retourner Aucun.
    - La fonction ne doit pas modifier `self.moves_made`, `self.mines`, `self.safes`, ou `self.knowledge`.
- `make_random_move` doit retourner return un coup aléatoire `(i, j)`.
    - Cette fonction sera appelée si un coup sûr n'est pas possible : si l'AI ne sait pas où se déplacer, elle choisira plutôt de se déplacer au hasard.
    - Le coup ne doit pas être un coup déjà effectué.
    - Le coup ne doit pas être un coup sur une cellule connue comme étant une mine.
    - Si aucun mouvement de ce type n'est possible, la fonction doit renvoyer Aucun.

8. Indications

a) Vous pouvez ajouter de nouvelles méthodes à l'une des classes si vous le souhaitez, mais vous ne devez modifier les définition ou arguments des fonctions existantes.
b) Lorsque vous lancez votre AI (comme en cliquant sur "AI Move"), notez qu'elle ne gagnera pas toujours ! Dans certains cas, l'AI devra deviner, car elle ne dispose pas de suffisamment d'informations pour effectuer un déplacement en toute sécurité. Il faut s'y attendre. `runner.py` indiquera si l'AI effectue un mouvement qu'elle croit sûr ou si elle effectue un mouvement aléatoire.
c) Faites attention à ne pas modifier un ensemble en le parcourant. Cela pourrait entraîner des erreurs !

9. Soumission

Le dépôt se fera en ligne dans la plateforme https://fad.univ-thies.sn/. Vous êtes enrôlé dans l’espace du cours d’IA. Vous pouvez aussi partager avec moi votre espace Git contenant le projet.