FACULTÉ DES SCIENCES
DÉPARTEMENT D'INFORMATIQUE

UNIVERSITÉ LIBRE DE BRUXELLES ULB

INFO-F-311: Intelligence Artificielle

Projet 2: Recherche adversariale

Yannick Molinghen **Pascal Tribel** **Tom Lenaerts**

1. Préambule

Dans ce projet, vous allez implémenter des techniques d'intelligence articifielle basées sur de la recherche dans des graphes en considérant un ou plusieurs adversaires. On vous fournit des fichiers de base pour le projet que vous pouvez les télécharger sur l'université virtuelle.

1.1. Rust

L'environnement dans lequel vous allez travailler est implémenté en Rust. Pour pouvoir compiler le projet, vous aurez besoin d'installer le compilateur adapté avec la commande ci-dessous issue du site officiel.

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Si vous utilisez Windows, vous pouvez utiliser Windows Subsystem for Linux (WSL) ou vous référer à la procédure décrite dans la documentation.

1.2. Poetry

Le projet nécessite Python $\geq$ 3.10 et utilise Poetry pour gérer les dépendances Python et les environnements virtuels. Ouvrez un terminal dans le répertoire du projet, installez Poetry puis installez les mises dépendances du projet:

```
poetry shell
poetry install
```

Ces commandes vont automatiquement créer un environnement virtuel puis installer les dépendances du projet.

1.3. Tests automatiques

Vous pouvez tester vos méthodes avec la commande `pytest`. Pour lancer uniquement les tests d'un fichier en particulier, vous pouvez le donner en paramètre:

```
pytest tests/tests_bfs.py
```

Il y a un fichier de test pour chacune des tâches que vous devez remplir.

1.4. Fichiers existants

On vous donne les fichiers `mdp.py`, `adversarial_search.py` et `world_mdp.py`, mais vous ne devrez travailler que dans les deux derniers. `mdp.py` contient la déclaration de l'interface d'un Markov Decision Process, `adversarial_search.py` contient les algorithmes de recherche que vous allez implémenter tandis que `world_mdp.py` contient l'adaptation du `lle.World` en `MDP` dans lequel chaque agent agit chacun à son tour.

Important: Vous pouvez modifier **TOUT** ce que vous voulez dans le code qui vous est donné à l'exception des tests unitaires. Ce code vous est uniquement donné comme guide.

1.5. Laser Learning Environment

Comme lors du premier projet, vous allez utiliser la librairie `lle` illustrée dans la Figure 1 pour ce projet. Il existe un jupyter notebook sur le git de du projet qui vous montre les fonctionnalités de l'environnement et qui vous permet de vous familiariser à son utilisation.

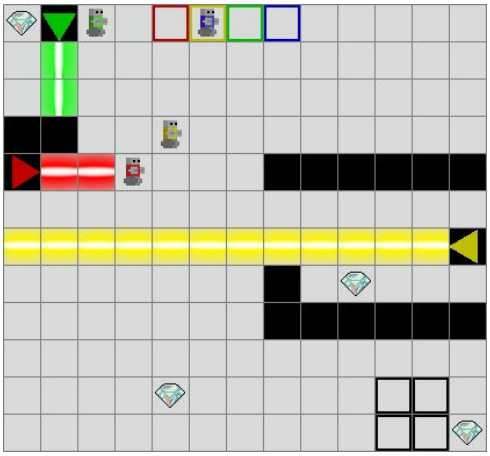

Figure 1: Rendu de l'environnement

Rapport de bugs: L'environnement LLE est encore récent, et quelques bugs pourraient subsister. Dans le cas où vous pensez en avoir trouvé un, n'hésitez pas à le signaler par email à l'adresse yannick.molinghen@ulb.be en joignant le code minimal nécessaire à le reproduire. Chaque première personne à rapporter un bug (et qui est confirmé) donnera lieu à un point supplémentaire avec un maximum de 2 points sur le projet.

2. Adaptation du `World` en `MDP`

Comme LLE est initialement un environnement cooperatif, vous allez adapter l'environnement de sorte qu'il fonctionne comme un Markov Decision Process compétitif dans lequel un agent agit à la fois. **Votre but est de maximiser le nombre de gemmes collectées par l'agent 0**, les autres agents étant considérés comme des adversaires. Pour ce faire, implémentez les classes `WorldMDP` et `MyWorldState` dans le fichier `world_mdp.py`.

`MyWorldState`

Comme il s'agit d'un MDP à plusieurs agents et à tour par tour, chaque état doit retenir à quel agent c'est le tour d'effectuer une action.

`WorldMDP`

- Après avoir fait `reset()`, c'est à l'agent 0 d'effectuer une action. Ainsi, `world.transition(Action.NORTH)` fera uniquement bouger l'agent 0 vers le nord tandis que tous les autres resteront sur place. Ensuite, c'est à l'agent 1 de bouger et ainsi de suite.
- La méthode `world.available_actions(state)` renvoie les actions accessibles à l'agent courant.
- La valeur d'un état correspond à la somme des *rewards* obtenues par les actions de l'agent 0 (c'est-à-dire les gemmes collectées + arriver sur une case de fin).
- Si l'agent 0 meurt lors d'une transition, la valeur de l'état tombe à la valeur `lle.REWARD_AGENT_DIED` (-1) immédiatement, sans prendre en compte les gemmes déjà collectées.

3. Algorithmes de recherche

3.1. Minimax

Dans le fichier `adversarial_search.py` implémentez l'algorithme du minimax dans la fonction `minimax` qui renvoie l'action à effectuer par l'agent 0 dans l'état donné. Cette fonction n'accepte que des états pour lesquels c'est à l'agent 0 de jouer et lève une `ValueError` dans le cas contraire. On vous suggère de diviser cet algorithme en une fonction `_min` et une fonction `_max`. N'oubliez pas qu'il peut y avoir plus d'un adversaire.

3.2. Alpha-beta pruning

De manière similaire à la tâche précédente, implémentez l'algorithme $\alpha\beta$-pruning dans la fonction `alpha_beta`. Cette fonction renvoie l'action à effectuer pour l'agent 0 dans l'état donné.

3.3. Expectimax

Dans minimax et $\alpha\beta$-pruning, nous avons supposé que l'adversaire agissait de manière optimale. Cependant, ce n'est pas toujours le cas pour des humains. L'algorithme « expectimax » permet de modéliser le comportement probabiliste d'humains qui pourraient prendre des choix sous-optimaux. La nature d'expectimax demande que nous connaissions la probabilité que l'adversaire prenne chaque action. Nous allons ici supposer que les autres agents prennent des actions uniformément aléatoires.

3.4. Meilleure évaluation

Comme dernier exercice, on vous demande d'implémenter une sous-classe de `WorldMDP` dans laquelle la valeur d'un état est calculée de manière plus intelligente que simplement en considérant le score de l'agent 0. Ecrivez cette sous-classe et vérifiez que pour une même carte, le nombre de noeuds étendus est en effet plus bas qu'avec la fonction d'évaluation de base. Il n'y a pas de test pour cet exercice.

4. Rapport

On vous demande d'écrire un court rapport (environ 2 pages) sur deux sujets:

- Expliquez l'idée derrière votre fonction d'évaluation utilisée en Section 3.4
- Comparez le nombre d'états étendus dans différents algorithmes pour un même niveau. Pour ce faire, inventez trois cartes (documenation dans le jupyter notebook et sur le wiki du git) que vous illustrez dans le rapport. Pour chacune des cartes, reportez graphiquement le nombre d'états étendus dans la recherche d'action pour
  - `minimax`,
  - `minimax` avec votre meilleure fonction d'évaluation,
  - `alpha_beta`,
  - `alpha_beta` avec la meilleure fonction d'évaluation.

  Veillez à choisir une méthode de représentation graphique adaptée.

Remise

Le livrable de ce projet se présente sous la forme d'un fichier zip contenant les fichiers: `adversarial_search.py`, `world_mdp.py` et votre rapport au format PDF.

Le travail est **individuel** et doit être rendu sur l'Université Virtuelle pour le 29/10/2023 à 23:59.