Instruction for GVGAI Single-Player Learning track

Useful links

GVGAI competition

GVGAI framework

GVGAI single-learning track framework, branch *singleLearning2017*

GVGAI wiki (planning tracks and level generation track)

Overview

The Single-Player Learning track is based on the GVGAI framework. Different from the planning tracks, no forward model is given to the agent, thus, no simulation of game is possible. It is notable that the agent still has the access to the current game state (objects in the current game state), as in planning tracks.

Main procedure

For a given game, each agent will have **10 minutes** for training on levels 0,1,2 of the game, the level 3 and 4 will be used for validation.

Main steps during training

1. Playing once levels 0, 1 and 2 in a sequence: Firstly, the agent plays once levels 0,1,2 sequentially. At the end of each level, whatever the game has terminated normally or the agent forces to terminate the game, the `results()` method will be called and send the results of the (possibly unfinished) game to the agent.
2. (Repeat until time up) Level selection: After having finished step 1, the agent is free to select the next level to play (from levels 0, 1 and 2) by calling the method `nextLevel`. If the selected level id $\notin \{0, 1, 2\}$, then a random level id $\in \{0, 1, 2\}$ will be passed and a new game will start. This step is repeated until **10min** has been used.

Main steps during validation

During the validation, the agent plays once levels 4 and 5 sequentially.

Remark: Playing each level once or several times is to be decided.

Methods to implement and time control

Constructor of the agent class

```
public Agent(SerializableStateObservation sso, ElapsedCpuTimer elapsedTim
er){...}
```

The constructor receives two parameters:

- `SerializableStateObservation sso`: The `StateObservation` is the observation of the current state of the game, which can be used in deciding the next action to take by the agent (see doc for planning track for detailed information). The `SerializableStateObservation` is the serialised `StateObservation` **without forward model**, which is a `String`.
- `ElapsedCpuTimer elapsedTimer`: The `ElapsedCpuTimer` is a class that allows querying for the remaining CPU time the agent has to return an action. You can query for the number of milliseconds passed since the method was called (`elapsedMillis()`) or the remaining time until the timer runs out (`remainingTimeMillis()`). The constructor has **1 second**. If `remainingTimeMillis()` ≤ 0, this agent is **disqualified** in the game being played.

Initialise the agent

```
public Types.ACTIONS init(SerializableStateObservation sso, ElapsedCpuTim
er elapsedTimer){...}
```

The `init` method is called once after the constructor, before selecting any action to play. It receives two parameters:

- `SerializableStateObservation sso`.
- `ElapsedCpuTimer elapsedTimer`: (see previous section) The `act` has to finish in **40ms**, otherwise, the `NIL_ACTION` will be played.

Select an action to play

```
public Types.ACTIONS act(SerializableStateObservation sso, ElapsedCpuTime
r elapsedTimer){...}
```

The `act` method selects an action to play at every game tick. It receives two parameters:

- `SerializableStateObservation sso`.
- `ElapsedCpuTimer elapsedTimer`: The `act` has to finish in **1 second**, otherwise, this agent is **disqualified** in the game being played.

Select the next level to play

```
public int nextLevel() {...}
```

During the step 2 of training, after terminating a game, the agent is supposed to select the next level to play. If the return level id $\notin \{0, 1, 2\}$, then a random level id $\in \{0, 1, 2\}$ will be passed and a new game will start.

Abort the current game

The agent can abort the current game by calling the method `abort()`. The agent will receive the results of the unfinished game. Then the agent is supposed to select the next level to play using the method `nextLevel(int nextLevelId)`.