Instructions for GVGAI Single-Player Learning track

Thanks for using the GVGAI Single-Player Learning framework. All suggestions are welcome.

Contact: Jialin Liu, University of Essex, UK

Email: *jialin.liu@essex.ac.uk* or *jialin.liu.cn@gmail.com*

Useful links

GVGAI competition

GVGAI framework

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 games is possible. The agent does still have access to the current game state (objects in the current game state), as in the planning tracks.

Main steps

For a given game, each agent will have **5 minutes** for training on levels 0,1,2 of the game, the levels 3 and 4 will be used for validation. The communication time is not included by Timer.

Main steps during training

1. **Training phase 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, whether the game has terminated normally or the agent forces to terminate the game, the server will send the results of the (possibly unfinished) game to the agent.
2. **Training phase 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 `int result()` (detailed later). 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 **5 minutes** has been used.

In case that 5 minutes has been used up, the results and observation of the game will still be sent to the agent and the agent will have no more than **1 second** before the validation.

Main steps during validation

During the validation, the agent plays once levels 3 and 4 sequentially.

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

Methods to be implemented and time control

This section details the methods to be implemented in Agent. A sample random agent is given in the framework.

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 timer with maximal time **40ms** for the whole training. The `act` has to finish in **40 ms**, otherwise, this agent is **disqualified** in the game being played.

Abort the current game

The agent can abort the current game by returning the action `ACTION_ESCAPE`. The agent will receive the results and serialised state observation `sso` of the unfinished game, timer and returns the next level to play using the method `int result(...)`.

Select the next level to play

```
public int result(SerializableStateObservation sso, ElapsedCpuTimer elaps
edTimer) {...}
```

During the step 2 of training, after terminating a game and receiving the results and final game state, 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. The `result` method receives two parameters:

- `SerializableStateObservation sso`: the serialised observation of final game stat at termination.
- `ElapsedCpuTimer elapsedTimer`: The global timer with maximal time 5 mins for the whole training. If there is no time left (`remainingTimeMillis()` ≤ 0), an extract timer with maximal time=1 second will be passed.

Structure of framework

The framework can be downloaded here (master). There are two main projects to work on:

- `gvgai/clients/GVGAI-JavaClient`: It contains the all the necessary classes for client, including the communication and a sample random agent. You can create an agent for the Single-Player Learning Track of the GVG-AI competition just by creating a Java class that inherits from `utils.AbstractPlayer.java`. This class must be named `Agent.java` and its package must be the same as the username you used to register to the website (this is in order to allow the server to run your controller when you submit). During the testing phase, the package can be located in `gvgai/clients/GVGAI-JavaClient/src/agents`.
- `gvgai/src/tracks/SingleLearning`: This package contains `LearningMachine.java`, a test `TestSingleLearning.java`, `ServerComm.java` and the necessary `runClient_nocompile.bat` and `runClient_nocompile.sh` files for compiling the client and building the communication.

How to test

Step 1: Import the projects

- Launch the IDE, open the project `gvgai`.

Step 2: (Optional) Advanced settings

- Set the port for communication: If you don't want to use the default setting with port 8000, please edit `TestSingleLearning.java`.
- You may need to edit `gvgai/src/tracks/SingleLearning/runClient_nocompile.bat` to specify the path to the sdk.
- If you move the `GVGAI-JavaClient` project to another folder, please change the path to the build and source folders in `runClient_nocompile.bat` or `runClient_nocompile.sh`. Attention: this will affect the location of the output file ClientLog.txt (see `GVGAI-JavaClient/src/utils/IO.java`).

Step 3: Set your agent

If you want to use your own agent, you can pass the name of agent as a parameter to `JavaClient.java`. Otherwise, the sample random agent `gvgai/clients/GVGAI-JavaClient/src/agents/random/Agent.java` will be used.

Step 4: Launch the server and client

Run `TestSingleLearning.java` in `gvgai/src/tracks/SingleLearning`. It will compile the client, build the communication and launch the program.

Possible issues and actions

If gson not found

Please open *Project structure/Libraries*, add the `gvgai/lib/gson-2.8.0.jar`.

FileNotFound exception for logd/ClientLog.txt

Please check the location of the output file ClientLog.txt (see `GVGAI-JavaClient/src/utils/IO.java`).