ECS7002P: Artificial Intelligence in Games

Set: 2nd October 2018

Due: Friday 2nd November 2018

Assignment I: μRTS

1 Summary

The first assignment for ECS7002P consists of writing and submitting a player for the game *μRTS* (a simplified version of *Starcraft*). This assignment is made in groups of **3 students**. You can join existing (empty) groups on QMPlus, by following these instructions (see Step 3): `https://elearning.qmul.ac.uk/guide/groups-student-self-select-a-group/`. If you can't make a group of 3, please send me (diego.perez@qmul.ac.uk) an email.

The repository with the code can be found in this link:

`https://github.com/GAIGResearch/microrts`

You can use **any** technique seen during the module. Given the nature of the framework and the fact that it provides a Forward Model, search methods (as seen in Lecture 2) are particularly suitable. However, you can also use (alone or in combination) other techniques seen during the module, such as decision and behaviour trees, blackboards, working memories or reinforcement learning methods.

Whatever you choose to implement, you must discuss the algorithm in the final report, together with a motivation for your choice and a study of its performance. The assessment of this assignment is thus composed of the following parts:

- Controller implemented (15%).
- Written report (15%).

2 Submission

The submission of the assignment must be done through **QMPlus**. Each group must submit two files: a zip file with the sources of your controller (without including the framework) and a PDF file with the final report.

Submission of bots and final rankings:

The main file containing your bot must be named with the first surname of each member of the group, preceded by the letters "QM" (*QMSurname1Surname2Surname3.java*, e.g. *QMWalterSmithHolmes.java*) and uploaded to QMPlus together with any other new files your bot requires to run (not already included in the framework). The code will be assessed attending its correctness and clarity, as well as the presence of comments that explain the most critical parts of the agent.

All bots will be entered into a **round robin tournament**, having **100ms** for making decisions at each game tick. The winner of the competition will gain a 10% **extra mark** on the `controller` part of the mark breakdown (assuming it's different enough to the sample bots provided; and always respecting a maximum of 100% in the final mark). Similarly, the two runner-up bots (second and third ranked in the league) will gain a 5% **extra mark** on the same part.

Written Report

A report must be written with the following suggested structure:

- Abstract: a summary in a 70-150 words paragraph. Motivate the research and explain your findings.
- Introduction: initial introduction to the problem, motivation and outline of the rest of the paper.
- microRTS: Brief description of the game, and literature review of what has been done in research before in this **and** similar games.
- Background: Explain here the algorithm that the submitted agent's technique is based on, including literature review.
- Techniques Implemented: Explanation and description of the agent submitted, and possible variants. How do you value a state? Have you used abstract/macro actions in your approach (and how)?
- Experimental Study: Describe the experimental study performed to obtain the final submitted agent (i.e., the procedure used to obtain the best combination of parameters, effect on the results of adding different functionality to the agent, empirical comparison against sample bots, etc).
- Analysis: Analysis of the development process (What things worked? What things didn't work? Did something surprise you?)
- Overall conclusions (problems and difficulties found, main takeaways) and potential future work. Note that conclusions **must not** be a reflection of what did the assignment mean for you as a student, but a **critical summary** of the work and findings of the project.
- References.

The report must be written using one of the templates (Word or LaTeX) provided in QMPlus and it should be approximately 3500 words long. The report will be assessed attending factors such as its contents, clarity, explanations, references, quality of the algorithm designed and the experimental process used to validate the implemented approach, etc.

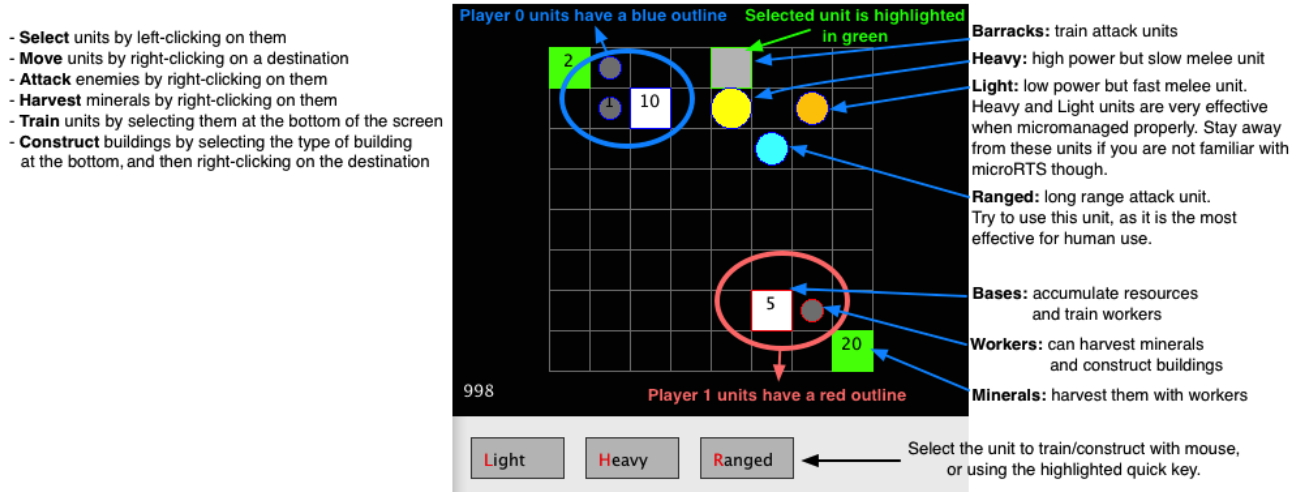
μRTS

The objective is to create a player that plays the game of *μRTS* as intelligently as possible.

μRTS a small implementation of an RTS game developed by Santiago Ontañón, designed to perform AI research. The advantage of using μRTS with respect to using a full-fledged game like Wargus or Starcraft (using BWAPI) is that μRTS is much simpler, and can be used to quickly test theoretical ideas, before moving on to full-fledged RTS games.

microRTS GitHub Repository

μRTS is a Real Time Strategy game in which two players compete to eliminate one another before the time runs out. As a player, you control a group of units of different types in real time, each one of them with different actions and characteristics. Your agent will be asked for an action for all your units at every game tick, which must be provided in no more than 100ms. All actions are *durative* (i.e. their execution spans over several game ticks) and cannot be cancelled - hence the importance of planning efficiently.



For a complete description of the game mechanics and rules, please check the instructions here:

`https://github.com/santiontanon/microrts/wiki/Game-Definition`

You can also see the game in action in this video:

`https://youtu.be/ZsKKAoiD7B0`

µRTS featured in an international AI competition in the IEEE-CIG 2017 and 2018 conferences. You are welcome to submit your agent to the next edition of the competition[1] but you can find information on the competition website about these editions, which includes results and approaches attempted in the past:

`https://sites.google.com/site/micrortsaicompetition`

The games used for this assignment will be fully observable and deterministic.

3 The Framework

Your are recommended to follow the exercises proposed for the labs before attempting the assignment. Below you can find a summary of the framework and coding instructions - but these are also covered in the exercises (in greater detail and step by step).

The code:

The code structure is described here:

`https://github.com/santiontanon/microrts/wiki/Code-Guide`

Creating bots:

In order to create a bot:

1. Implement a new class that extends from the class **ai.core.AIWithComputationBudget**. You **only** modify content in this class, **do not** alter the other classes of the framework.
2. Implement the **constructor**. This will take as a minimum two arguments: a **time budget** and an **iteration budget** (if the iteration budget is set to -1, this is infinite). The constructor can additionally take other arguments (you **must** specify these in your report and they **must** be reasonable and easily accessible, e.g. a pathfinding algorithm, the UnitTypeTable etc.). If you are not sure about this, ask!
3. Implement (i.e., overwrite from the parent class) these four necessary functions: reset, getAction, clone and getParameters.

[1] Not announced yet, and not a requirement for this assignment.

- **reset()**: This method should return your bot to its initial state, ready for a new game.
- **getAction(int player, GameState gs)**: This method is called at every game cycle for every bot in the game. It receives your player ID and the current game state and must return a **PlayerAction** object for the current game tick. A **PlayerAction** object is a list of pairs (**Unit**, **UnitAction**) - each unit can be assigned one (valid) action per game tick. This is where the main logic of your bot should be.
- **clone()**: This method should return a copy of the current state of your bot.
- **getParameters()**: This method should return a list of the parameters used in your bot. It *can* return an empty list. This method can be used to show the values of internal variables in your agent in the GUI mode (see below).

4. Optionally, implement the function **preGameAnalysis(GameState gs, long milliseconds, String readWriteFolder)**, which allows you to perform some pre-game reasoning. This method receives the initial game state, a time limit to execute and a path to the directory which it can write to or read from (e.g. reading a trained model in the case of learning methods). It will be called only if analysis before game starts is allowed.

5. When creating your AI, it's advisable that you use the Forward Model provided in *µRTS* to test your actions before issuing them in the game. There are two main procedures that you can use for this, which belong to the class **GameState**. For a **GameState gs**:

   - Cloning: **gs.clone()** creates a copy of the object **GameState** and returns it.
   - Rolling the state forward: You need to execute two steps:
     - 1) **gs.issue(PlayerAction pa)**: it sets the actions[2] described in **pa** to be executed in the game state **gs**; and
     - 2) **gs.cycle()**: executes the actions issued in the previous command in the state **gs**, which is modified internally to represent the future state of the game after applying the actions.

6. For testing your bot, you can follow the same instructions as shown in exercise 1 of this module.

Sample bots:

The framework comes with a few sample (and some not simple) bots to help you get started. These are located in the *ai* package. Details of all sample agents are provided here:

`https://github.com/santiontanon/microrts/wiki/Artificial-Intelligence`

Abstract actions: The action set of microRTS is very low level (for example, units have to move cell by cell, rather than using path-finding, as in standard RTS games). If you want to create an AI at a higher level of abstraction, the package **ai/abstraction** provides such abstraction by defining the actions: Move, Attack, Harvest and Train, with the standard behaviour that these actions have in games like Wargus or Starcraft.

Pathfinding: If you need to provide an algorithm for path-finding, in the package **ai/abstraction/pathfinding** you will find a few implementations you can use, including A* and greedy search.

Evaluation functions: Many AI techniques (such as game tree search techniques) require an evaluation function. The package **ai/evaluation** provides some simple evaluation functions that you can use. In particular SimpleEvaluationFunction is the most simple and fast one. EvaluationFunctionForwarding is more elaborate and takes into account the effect of the actions currently under execution for performing the game state assessment. All of them return a float number that is positive if the game state is good for the maximising player (you), and negative if the game state is good for the minimising player (your opponent).

[2] **pa** can include your actions, and also those (potentially) from your opponent