TestAI handbook

Introduction

The aim of this document is to give an overview about the ideas and classes of the AI gamer "TestAI" for TripleA. It was originally developed during an university course in 2015 (with TripleA version 1.8.0.9).

The main goal was to implement an AI for the movement of units based on the UCT algorithm. Other phases of the game are implemented too, but mostly for testing the AI in full game scenarios.

Basic concept

The overall idea for the TestAI was to create an versatile AI gamer, that is able to show different behaviours, and can be configured by the user to some extent.

E.g. the user should (at least) be able to choose between a "defensive", "normal" and "offensive" alignment of the AI before playing a map, and there are noticeable differences in the movement of AI units then.

Of course, it is a challenging task to translate this concept in a fully working and "strong" AI behaviour on different types of maps.

In theory, the current version of the TestAI should provide a solid core, that can be build upon in various directions.

Classes

The logical flow for a move (game round) by the TestAI:

1. TestAI
2. TestAnalyseBoardAI (getAlignmentForThisRound())
3. TestPurchaseAI
4. TestMoveAI (combatMove)
5. TestAnalyseBoardAI (setOverallStrategyForThisMove())
6. TestSetAndEvaluateGoals (setRelevantTerritoriesForGoals())
7. TestMoveAI (cM: simulate moves)
8. TestSetAndEvaluateGoals (evaluateGoal())
9. TestMoveAI (cM: perform best move-sequence)
10. TestMoveAI (nonCombatMove)
11. TestAnalyseBoardAI (setOverallStrategyForThisMove())
12. TestSetAndEvaluateGoals (setRelevantTerritoriesForGoals())
13. TestMoveAI (nCM: simulate moves)
14. TestSetAndEvaluateGoals (evaluateGoal())
15. TestMoveAI (nCM: perform best move-sequence)
16. TestPlaceAI

<u>TestAI</u>

Contains the logic for the different phases of the move (game round) and calls the according classes and methods.

<u>TestAnalyseBoardAI</u>

Analyses the current situation on the game board (units and territories of all players in relation to each other) and derives a (more or less) reasonable strategy for the move.

At the moment, the TestAI distinguishes three types of alignment ("defensive", "normal", "offensive") and applies it in the following phases:

- recalculation (of the alignment) based on the game data before "purchase"
- subsequent use in "combatMove"
- recalculation (of the alignment) based on the game data before "nonCombatMove"
- subsequent use in "place" (-> maybe should be recalculated here too, because "nonCombatMove" could lead to significant changes in the current strength of all players)

For example, in the present implementation it would be possible, that the TestAI purchases with "normal" behaviour, and places the units with "offensive" behaviour in the nonCombatMove, if the combatMove was successful prior to that.

Besides setting the alignment of the AI, this class also sets the strategy for the movement of the units:

Based on the alignment of the AI, several goals of the class TestSetAndEvaluateGoals are combined and specify thereby, which territories are targeted in the combatMove as well as nonCombatMove of this game round.

The target territories in turn affect the unitsThatCanMove: A list of units is provided, which can reach at least one of the target territories and therefore are relevant for the simulation of the move sequence in TestMoveAI.

TestSetAndEvaluateGoals

This class contains different "goals", which can be selected in TestAnalyseBoardAI. A goal defines, which territories are targeted by the AI, and which not.

The evaluation concentrates on the situation in each target territory after a simulated move: Would the territory be won, or lost?

I.e., raising the value of the overall move, keeping it unchanged or reducing it.

Obviously, the goals and respective evaluation formulas are decisive for the strength of the AI player. I.e., defining and combining goals would profit from a solid knowledge about the game and its tactics.

Otherwise, goals could overrule each other, or lead to a predictable behaviour of the AI player.

Some exemplary properties for the calculation of goals:

- strategic position of a territory
- count of enemy units near territory
- how much PUs a territory is worth

Theoretically, this "goal" approach should allow to create AI players with distinct behaviour, and could be tailored to single maps.

E.g., creating an AI player that operates on sea mostly by the usage of goals that rely on water-territories. Or an AI player that only defends its capital city.

Ideas for further improvement:

- subdivide goals in categories and classes for better clarity
- establish a reliable concept for evaluation formulas
  - e.g., it could be possible, that 3 out of 4 goals turn out well, but goal 4 of 4 fails and would be more important for the overall move

- develop a modular "goal editor": a GUI for simplifying the definition of goals
  - to some extent, this could be an approach for the creation of own AI players by "average users" too

TestPurchaseAI

This class allows the AI player to purchase units, which are placed via TestPlaceAI subsequently.

It is not very sophisticated yet, and may not worth to build upon.

The core idea would state a calculation formula that determines a "purchase attractivity" for each unit type based on the alignment of the AI player and the current enemy units on the game board.

I.e., selecting the most appropriate unit type, that could help in the specific situation.

Ideas for further improvement:
- maybe it is wiser to rest the purchase AI upon "expert knowledge" than weighting unity types and properties
  - i.e., a group of units suitable for defending a territory usually consists of x units of unit type 1 combined with y units of unit type 2 etc.
    - this approach could be depicted in a matrix perhaps, where all unit types and combinations are described in their efficiency opposed to each other, including the actual numbers of units on the game board
      - i.e., compiling the game (map) dependent matrix of unit types and evaluating it with the present numer of units before purchasing new units

TestMoveAI

This class is the key component of the TestAI gamer.

Put simply, the whole simulation and execution of the phases "combatMove" and "nonCombatMove" is computed here.

For all units, that have been selected as "moveable" in TestAnalyseBoardAI before, each possible move is simulated in accordance to the target territories selected in TestAnalyseBoardAI (via TestSetAndEvaluateGoals).

In detail:

If there a transports, that (or other land units with their help) can reach territoriesToMoveTo, the transports are moved first (because possible moves of transported land units can depend on it).

Likewise, carriers are moved next, because possible moves of "fighters" can depend on it.

Then all other units (land, air, sea) perform their moves (technically, it would be possible to define a further order, e.g., units with lower movement first).

One examplary simulation run:

transport1 has 5 possible moves -> 5 resulting states as children

carrier1 has 3 possible moves -> 3 resulting states as children

One of the children of transport1 is selected by the UCT algorithm. From this resulting state, one of the children of carrier1 is selected.

Based on the selected move of transport1, landunit1 has 3 children and selects a move, where it

boards transport1. Landunit2 has 2 children, and boards transport1 too.

Landunit3 could have reached a target territory with the help of transport1 as well, but the capacity of transport1 is already exhausted in this case. So landunit3 remains on its current territory, because no other target territory is in range.

Fighter1 could land on carrier1, but chooses a landing territory which it can reach on its own.

At the end of the simulation run, all mentioned units are on their "new" territories. The resulting "game board" is evaluated then according to the goals set in TestAnalyseBoardAI before.

If the resulting value is higher than the previous one, the current movement sequence is stored as new reference.

Depending on the map and number of units, the phases combatMove/nonCombatMove are finished after a predefined duration (15 seconds are set at the moment), or, what would be the ideal case, if an end node is reached via the UCT algorithm (-> e.g. possible on MiniMap with fewer units).

Concerning the UCT algorithm, the more simulation cycles are performed, the better the result should be (at least in theory).

Incorporating transports and carriers is quite time consuming (-> because there are much more possible moves for several units, and corresponding children/leaves to assess).

So one central question is, if the TestAI should have the opportunity to select from all possible single moves, or units are grouped and moved together in the simulation.

From a technical standpoint, the most reasonable solution maybe is to allow both variants.

Ideas for further improvement:

- separate the class in several classes for better readability
- everything, that speeds up the simulation runs, and increases the strength of the AI gamer
  - e.g., the calculation of transports and carriers could be more compact

TestPlaceAI

This class enables the TestAI to place the purchased units on territories with own factories or sea territories (sea units). Technically, it finishes the game round of the TestAI player.

Basically, the state of this class is similar to TestPurchaseAI: Everything is based on a calculation formula, which is not very sophisticated yet, and maybe more complicated than necessary.

The idea would be, that a purchased unit is placed on a territory, where it could make the most impact according to the alignment of the TestAI gamer.

For example:

An air unit should be placed on a territory nearest to the front line, if the alignment is "offensive".

On the contrary, a "defensive" alignment would propose to place the units in territories of the own "heartland".

+ several container-classes

Testing

First of all, TestAI was developed and tested with TripleA version 1.8.0.9. So it is hard to say, how operable it is in newer versions without further adjustments.

Testing and developing was done with the help of the maps "MiniMap" and "World War II Revised", as well as a logger (class "SimpleLogger") in a text file. Additionally, the ingame history was used to trace the movement of the units, and comparing it with the data of the test log.

In general, the TestAI gamer should work on all maps with the following phases and sequence:

- Purchase
- Combat
- NonCombat
- Place

Regarding the operability in further phases or other sequences, some parts of the logic would have to be adapted (-> e.g. the behaviour of the AI player should not be set in the phase "purchase").

Known bugs

Altogether, the TestAI gamer should be tested sufficiently to work in TripleA version 1.8.0.9 on the maps "MiniMap" and "World War II Revised" with one or more instances.

Observed bugs:

- sometimes own units (of all types) are not found by TripleA and not moved in consequence
  - maybe a problem with getUnits(), that is not present anymore
- earlier in development, there occasionally were problems with duplicated units, that also started battles in nonCombat too (e.g. duplicated japanese units in Buryatia on the map "World War II Revised")
  - was not observed in later stages, but maybe is still present in specific cases

Potential bugs:

- it is possible, that rarely occurring constellations are not considered in the classes "TestMoveAI" or "TestPlaceAI" yet, and could lead to problems

Development

The code of "TestAI" can be used under the terms of GNU General Public License version 2 or later.

Building the project

Note: This guide describes the original setup with TripleA version 1.8.0.9 on Windows 7 and Eclipse Juno as development environment.

-> download:

TripleA installer version 1.8.0.9:

https://github.com/triplea-game/triplea/releases/download/1.8.0.9/triplea_1_8_0_9_windows_installer.exe

TripleA source code version 1.8.0.9:

https://github.com/triplea-game/triplea/archive/refs/tags/1.8.0.9.zip

-> install game with the installer

-> unpack source code

-> insert folder "TestAI" in the folder with the source code besides the other AI-folders: ...\triplea_1_8_0_9\src\games\strategy\triplea\TestAI

-> create a new Java Project in Eclipse:
- "File"->"New"->"Java Project"
- choose a name for the project
- the selected JRE should be JavaSE-1.6 or higher
- click on "Finish"

-> delete the empty folder "src" in the project folder

-> right-klick on project folder and open "Build Path->Link Source"

-> select folder with the source code ...\triplea_1_8_0_9\src and click on "Finish"

-> right-klick on the project folder and open "Build Path->Configure Build Path"
- go to the "Libraries"-tab and klick on "Add External JARs"
- navigate to the installed TripleA-folder, and select the existing "triplea.jar" in the folder "bin" there: ...\triplea_1_8_0_9\bin\triplea.jar
- the JAR should now be listed under "Libraries": expand "triplea.jar" by klicking the arrow then, and double-klick on "Source attachment". Select "External location" and "External folder" next, and choose the folder with the source code ...\triplea_1_8_0_9\src again.
- confirm by klicking on "OK" two times

-> select "Run" in the menu bar, and open "External Tools->External Tools Configurations“

-> choose "Ant Build->New launch configuration" (Ant Build should be installed by default in Eclipse)

-> select the build file from the folder with the source code there: ...\triplea_1_8_0_9\build.xml

-> switch to the tab "JRE", and set the folder with the source code ...\triplea_1_8_0_9 as "working directory" for "Other" (select it via "File System")

-> confirm by klicking on "Apply", and click "Run" to start the build

-> "triplea.jar" should be built in the folder with the source code now correctly for the linked source code as ...\triplea_1_8_0_9\bin\triplea.jar

-> using this "triplea.jar", the game can be started and tested with the code of the Eclipse-project included

-> to integrate "TestAI" in the game:
- open the class `games.strategy.triplea.TripleA` and add the „new“ TestAI: `public static final String TEST_COMPUTER_PLAYER_TYPE = "Test (AI)";`
- go to method `createPlayers()` and append `else if (type.equals(TEST_COMPUTER_PLAYER_TYPE)){players.add(new TestAI(name, type));}`
- Eclipse should display a message now, advising to import "TestAI": confirm this, resulting in `import games.strategy.triplea.ai.TestAI.TestAI;`
- finally, go to method `getServerPlayerTypes()` and add `TEST_COMPUTER_PLAYER_TYPE`

-> use the previously created "New launch configuration" to build "triplea.jar" and run the game

-> "Test (AI)" can be selected within the process of "Start Local Game" now, and plays in

accordance to the implementation of the different phases in the code (as described in the section "Classes" in this handbook)