Developing an agent for sge-risk

Lukas Grassauer

last changed: 2019-11-22

Please also refer to the "SGE-Manual"

Running a game of risk

```
java -jar sge-1.0.0.jar match \
 --file=sge-risk-1.0.0.jar \
 --directory=agents
```

If you want specific agents to play against each other add them via `--file` instead of `--directory` or use their Agent-Name like so:

```
java -jar sge-1.0.0.jar match \
 --file=sge-risk-1.0.0.jar \
 --directory=agents \
 -a MctsAgent AlphaBetaAgent
```

Note that the agents play in that exact order. To limit the calculation time to n seconds use `--computation-time` n or the equivalent short option `-c` n.

Running a tournament

```
java -jar sge-1.0.0.jar tournament \
 --file=sge-risk-1.0.0.jar \
 --directory=agents \
 --mode="Double_Round_Robin"
```

Implementing an agent

The agent has to implement the Java-Interface `at.ac.tuwien.ifs.sge.agent.GameAgent<? extends Game<A, ?>, A>` where `A` is the type of the action, provided by `sge`.

Before each turn the game is stripped from player-specific information and then submitted to the agent via the `computeNextAction()` method, alongside the allowed computational time as well as its unit.

Whatever this action returns is then applied to the canonical game via `doAction()`. Note however that `doAction()` is side-effect free, as it returns a new instance of the game, with that move applied. Risk-Agents ought to return objects of the class `RiskAction`.

To access the board `getBoard()` can be used. This method returns a copy of the games internal object of type `RiskBoard`. This class provides a number of helpful methods which can be used to determine the current state of the game. In a sense it can also be used to generate legal moves, or rather check if a move is legal. Check the specific javadoc for explanations.

Objects of type `RiskAction` can be generated via their classes static methods. For example `RiskAction.attack(src, dst, trp)` generates a `RiskAction` with the intent to attack `dst` from `src` with `trp` troops. The other actions are generated in a similar fashion. Check the javadoc for explanations.

The game is also able to generate all legal actions by itself. `getPossibleActions()` is precisely for that purpose. Note that in uncanonical games the number of generated actions might be larger than the actual legal ones, depending on the state of hidden information. However `getPossibleActions()` never generates illegal moves, and the actions of the submitted game are the only possible ones.

Besides generating moves the game also has a embedded utility evaluation. The utility value is used to determine which player has won. It can be accessed via `getUtilityValue()`. Moreover the game interface also provides a heuristic evaluation via `getHeuristicValue()` which is a rather simple way to predict general trend of the utility values throughout the game. In the case of Risk `getUtilityValue()` returns 1 if and only if the current player has won, and 0 otherwise. `getHeuristicValue()` returns the number of occupied territories of the current player without argument and with argument the number of occupied territories of that specified player.

`isValidAction()` only returns true if and only if the action would be a valid action for that game, and can be used to verify before returning the actual action.

Quickstart with RiskAgent

Simply fork from the attached `RiskAgent` to start from a simple agent that compiles, it might help in reducing headaches.

Other Tips

Canonical Games

A game is called canonical if all information is perfect and complete, which SGE models all games to. Indeterminateness are resolved by a "World"-Player which uses randomness to determine their moves. Uncanonical games are stripped of some information. These games are passed to the agents.

Dice Throws

In the board game risk parts of the game are determined by rolling dice. SGE does not model this directly, but rather the outcomes of repeated dice throws, which are how many troops are lost. Advancing the game in an agent can either be done by using a random number generator or by the `Game.determineNextAction()` method. Which chooses an outcome based on the probabilities.

Determinacy

An uncanonical game is uniquely identified by the list of moves it lead to. Canonical games can have initial randomness (shuffle of a deck of cards - modelling this would result in an enormous game tree) which is not shown to the agents.

Game tree

SGE assumes the game tree has a finite width. If there are multiple possible moves due to indeterminacy then all possible outcomes are given as next possible actions.

Default Agents

SGE provides three default agents:

- RandomAgent
  - An agent which chooses from all possible actions a random one.

- AlphaBetaAgent
  - An agent which uses (aggressive) alpha beta pruning to determine the next best action.
- MCTSAgent
  - An agent which uses Monte-Carlo-Tree-Search to find the most likely winning action.

Be wary however these agents are not implemented for Risk directly but rather use an abstract notion of games, so the heuristics are rather bad, i.e. they do not model the game particularly well. For example do they not consider that a territory is easier to be captured if they do not have a lot of troops in this territory (well they do, but not in an explicit sense). Hence beating them might not be too difficult.

Use fuzzy heuristics

The game tree for Risk is rather large. A simple suggestion to reduce the width would be to not make too much of a difference if a territory has for example 10 or 11 troops. There are charts on the probabilities and combining very similar probabilities might be worth it.

Additionally if a player only has a few territories left it might take a long time to finish the game. This can result in a deep game tree which is a computational feat. Reduction is possible by cutting off after winning/losing is improbable.