SGE 1.0.0 - Manual

Draft

Lukas Grassauer

last changed: 2019-11-22

About

`sge` is a game engine primarily designed for autonomous agent playing discrete and sequential games.

Prerequisites

Java Installation

sge is written for Java 11 which is available under https://jdk.java.net/. Verify the version with

```
java -version
```

The version should be at least 11.

Engine Installation

The engine itself is bundled with all it's dependencies in a fat-jar, usually named `sge-1.0.0-exe.jar` and can be executed with

```
java -jar sge-1.0.0-exe.jar
```

It does not have to be installed in a classical sense.

Vocabulary

Match A certain number (can also be only one) of agents play a single game.
Round consists of every players **turn**.

Turn consists of all the **actions** a player makes until he ends his **turn**. If a turn consists only of a single action it is also called a **move**.

Action something the player can do which can end the **turn** or lead into another **action**.

Synopsis

`sge`

The engine is a executable jar file which loads other jar files and the usage follows the POSIX principle (note `java -jar sge-1.0.0-exe.jar` is substituted by `sge`):

```
sge [ENGINE-OPTIONS]... [COMMAND [COMMAND-OPTIONS...] [COMMAND-ARGUMENTS]...]
```

It is important to note that straying from that principle might result in unexpected behaviour.

The engine has, as of version 1.0.0, two working commands:

1. match
2. tournament

Generally the engine either takes paths or strings as input from the user. The engine automatically determines if a string is a path to a file, and if not it falls back to interpreting it as a string.

`sge match`

A match is a single game that is played by agents. There are several ways to allow agents to play matches against each other.

Firstly the engine will create a list of agents that are going to play in this match. This list is called the agent configuration. Should the user not have given any, it adds unused agents, and finally human players should those run out as well.

The agents are then instantiated and the match is played in this order. This implicitly sets the number of players.

Arguments

`sge match` can optionally take paths to a game file, agent file(s) and agent configurations. The engine will determine what is what automatically. For example:

```
sge match gameJar agent1Jar agent2Jar
```

will start a match with the game provided by `gameJar` with the agent provided by `agent1Jar` vs the agent provided `agent2Jar`.

Or if the order of the players matters (provided `agent1Jar` provides `agent1` and `agent2Jar` provides `agent2`):

```
sge match gameJar agent1Jar agent2Jar agent1 agent2 agent1
```

will start a match with the game provided by `gameJar`. The players are in that order an instance of `agent1`, `agent2` and another instance of `agent1`.

Options

1. `--debug`

   This option is a flag.

   Starts the engine in debug mode. No timeouts and verbose is turned on once (Log level is reduced by one).

2. `-a`, `--agent`

   This option has an arity of '1..*'.

   This is a more explicit variant to give configuration of agents. This needs to be terminated by another option or `--`.

3. `-b`, `--board`

   This option has an arity of '1'.

   Use a different board instead of the default. This can be a path or a string, depending on the game, one or both is allowed.

4. `-c`, `--computation-time`

   This option has an arity of '1'.

   Determine how long an agent is allowed to compute before a timeout. Humans cannot timeout. The unit is per default seconds, however it can be controlled by `-u` or `--time-unit`.

5. `-d`, `--directory`

   This option has an arity of '1..*'.

   This is a more explicit variant to give jars of game and agents. Every subdirectory will be considered. This needs to be terminated by another option or `--`.

6. `-f`, `--file`

   This option has an arity of '1..*'.

This is a more explicit variant to give jars of game and agents. This needs to be terminated by another option or `--`

7. `-h`, `--help`

   This option is a flag.

   Gives an usage overview.

8. `-p`, `--number-of-players`

   This option has an arity of '1'.

   Either set implicitly by the agent-configuration, the minimum required to play or explicitly by this option.

9. `-q`, `--quiet`

   This option is a flag.

   Increases the log level by one. These flags can be used cumulatively. `-qqq` therefore turns off any logging.

10. `-r`, `-s`, `--shuffle`

    This option is a flag.

    Shuffles the agent configuration before starting the match.

11. `-u`, `--time-unit`

    This option has an arity of '1'.

    This allows to scale the computation time.

12. `-v`, `--verbose`

    This option is a flag.

    Decreases the log level by one. These flags can be used cumulatively. `-vv` therefore turns on all logging.

`sge tournament`

A tournament are one or more matches which determine the outcome of a tournament.

Per default all agents which are loaded are included in the tournament. Via the agent-configuration it is possible to limit the contestants.

Arguments

`sge tournament` can optionally take paths to a game file, agent file(s) and agent configurations. The engine will determine what is what automatically. For example:

```
sge tournament gameJar agent1Jar agent2Jar
```

will start a tournament with the game provided by `gameJar` with the agent provided by `agent1Jar` vs the agent provided `agent2Jar`.

If only a select number of agents are to play in a tournament append the their agent names:

```
sge tournament gameJar agent1Jar agent2Jar agent3Jar agent1 agent2 agent1
```

will start a tournament with the game provided by `gameJar`. The players are in that order an instance of `agent1`, `agent2` and another instance of `agent1`, but not `agent3`.

Options

1. `--debug`

   This option is a flag.

   Starts the engine in debug mode. No timeouts and verbose is turned on once (Log level is reduced by one).

2. `-a`, `--agent`

   This option has an arity of '1..*'.

   This is a more explicit variant to give configuration of agents. This needs to be terminated by another option or `--`.

3. `-b`, `--board`

   This option has an arity of '1'.

   Use a different board instead of the default. This can be a path or a string, depending on the game, one or both is allowed.

4. `-c`, `--computation-time`

   This option has an arity of '1'.

   Determine how long an agent is allowed to compute before a timeout. Humans cannot timeout. The unit is per default seconds, however it can be controlled by `-u` or `--time-unit`.

5. `-d`, `--directory`

   This option has an arity of '1..*'.

This is a more explicit variant to give jars of game and agents. Every subdirectory will be considered. This needs to be terminated by another option or `--`.

6. `-f`, `--file`

   This option has an arity of '1..*'.

   This is a more explicit variant to give jars of game and agents. This needs to be terminated by another option or `--`

7. `-h`, `--help`

   This option is a flag.

   Gives an usage overview.

8. `-m`, `--mode`

   This option has an arity of '1'.

   As of version 1.0.0 `sge tournament` supports the following tournament modes:

   (a) Round Robin

   Default. Valid value: `Round_Robin`

   Requires at least 2 agents, but has no upper limit. Matches can be played with 2 agents, but at most as many as tournament contestants.

   Every combination of agent is played once.

   (b) Double Round Robin

   Valid value: `Double_Round_Robin`

   Requires at least 2 agents, but has no upper limit. Matches can be played with 2 agents, but at most as many as tournament contestants.

   Every permutation of agent is played once.

9. `-p`, `--number-of-players`

   This option has an arity of '1'.

   Implicitly the minimum required to play or explicitly by this option. Note that this does not change the number of involved agents in a tournament but rather how many are playing in a single match.

10. `-q`, `--quiet`

    This option is a flag.

    Increases the log level by one. These flags can be used cumulatively. `-qqq` therefore turns off any logging.

11. `-r`, `-s`, `--shuffle`

    This option is a flag.

    Shuffles the agent configuration before starting the tournament.

12. `-u`, `--time-unit`

    This option has an arity of '1'.

    This allows to scale the computation time.

13. `-v`, `--verbose`

    This option is a flag.

    Decreases the log level by one. These flags can be used cumulatively. `-vv` therefore turns on all logging.

Writing for sge

Writing an Agent

Build environment

Through the build tool make sure that following attributes are ensured:

- Source Compatibility: 1.11
- Following Manifest attributes
  - 'Sge-Type': 'agent'
  - 'Agent-Class': path.to.actual.agent
  - 'Agent-Name': The name of the agent
- Engine is in classpath
- Recommended: Game is in classpath

To achieve this in gradle:

```
sourceCompatibility = 1.11

repositories {
 jcenter()
}

dependencies {
  compile group: 'at.ac.tuwien.ifs.sge', name: 'sge', version: '1.0.0'
  //also consider to add the game in the same manner
}

jar {
 manifest {
  attributes 'Sge-Type': 'agent'
```

```
  attributes 'Agent-Class': 'path.to.actual.agent'
  attributes 'Agent-Name': 'The name of the agent'
 }
}
```

Development Environment

1. Intellij IDEA

   First create a new Gradle project, by selecting *File*, then *New* and then *Project...* (see Figure 1).

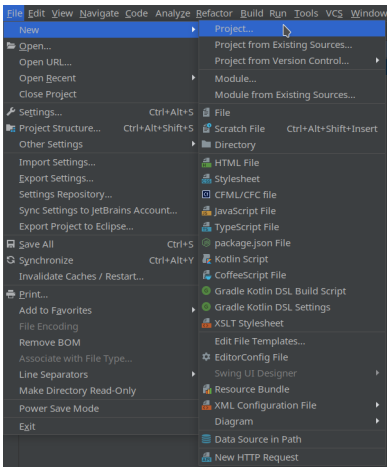

Figure 1: Create a new project in Intellij IDEA.

   Select *Gradle* (see Figure 2) and then follow the wizard.

   After that replace the contents of the `build.gradle` file with that given in *Build Environment*.

2. Eclipse

   First create a new Gradle project, by selecting *File*, then *New* and then *Project...* (see Figure 3)

   Select *Gradle*, then *Gradle Project* (see Figure 4) and then follow the wizard. After that replace the contents of the `build.gradle` file with that given in *Build Environment*.

Implementing the GameAgent Interface

In order to write an agent for sge a class has to implement the interface `GameAgent`. It is also highly recommended to extend from

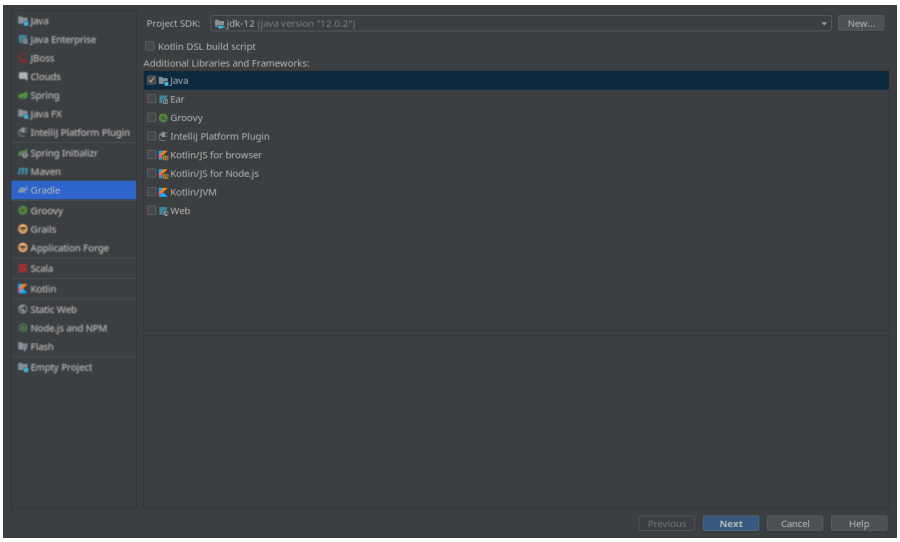

Figure 2: Select the Gradle project template.

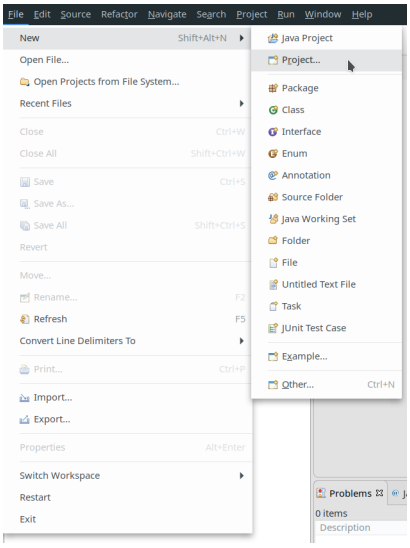

Figure 3: Create a new project in Eclipse.

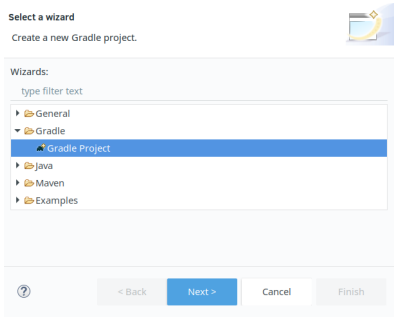

Figure 4: Select the Gradle project template.

`at.ac.tuwien.ifs.sge.agent.AbstractAgent`. It provides comparators which allow to compare games by utility and heuristic value and a method `shouldStopComputation()` which checks if the a certain part (per default half) of the computation time was already used.

Here an minimal working example that chooses the first available option of any game:

```
import at.ac.tuwien.ifs.sge.agent.*;
import at.ac.tuwien.ifs.sge.engine.Logger;

public class FirstAgent<G extends Game<A, ?>, A> extends AbstractGameAgent<G, A>
  implements GameAgent<G, A> {

 public FirstAgent(Logger log){
  super(log);
 }

 @Override
 public A computeNextAction(G game,
                            long computationTime,
                            TimeUnit timeUnit){
  //optionally set AbstractGameAgent timers
  super.setTimers(computationTime, timeUnit);
  //choose the first option
  return List.copyOf(game.getPossibleActions()).get(0);
 }

}
```

Note that there has to exist at least a constructor with `at.ac.tuwien.ifs.sge.engine.Logger` as argument. This logger does not have to be used though.

Every instance of the agents is created via this constructor. This also means that if the same agent plays against itself two instances of it are created.

Every agent also has the methods `setUp(numberOfPlayers, playerNumber)` called before every match, `tearDown()` called after every match, and `destroy()` called before shutting down. These methods can be used to get resources in place or to destroy them. Note that the same instance is used for multiple matches.

Game API

Every game follows the `Game<A, B>` API, where `A` is an action and `B` is the board.

The javadoc explains every method and their contracts in detail, however here are the most important relisted.

```
/**
 * Checks whether the game is over yet. Once this state is reached it can
 * not be left.
 *
 * @return true if and only if game over
 */
boolean isGameOver();

/**
 * Checks which player's move it is and returns the id of the player.
 * A negative number indicates some indeterminacy which is resolved by
 * the game itself.
 *
 * @return the id of the player
 */
int getCurrentPlayer();

/**
 * Applies the (public) utility function for the given player. The
 * utility function is the final measure which determines how
 * "good" a player does. The player with the highest value is
 * considered the winner. On equality it is considered a tie.
 *
 * @param player - the player
 * @return the result of the utility function for the player
 */
double getUtilityValue(int player);

/**
 * Applies the heuristic function for the given player. This function
```

```
     * is a more lax measure in how "good" a player does, it is not used
     * to determine the outcome of a game. Per default the same as
     * getUtilityValue().
     *
     * @param player - the player
     * @return the result of the heuristic function for the player
     */
    default double getHeuristicValue(int player) {
      return getUtilityValue(player);
    }

    /**
     * Collects all possible moves and returns them as a set. Should the
     * game be over an empty set is returned instead.
     *
     * @return a set of all possible moves
     */
    Set<A> getPossibleActions();

    /**
     * Returns a copy of the current board. Notice that only in non-canonical
     * games some information might be hidden.
     *
     * @return the board
     */
    B getBoard();

    /**
     * Checks whether doAction(action) would not throw an exception.
     *
     * @param action - the action
     * @return true - iff the action is valid and possible
     */
    boolean isValidAction(A action);

    /**
     * Does a given action.
     *
     * @param action - the action to take
     * @return a new copy of the game with the given action applied
     * @throws IllegalArgumentException - In the case of a non-existing action or null
     * @throws IllegalStateException    - If game over
     */
    Game<A, B> doAction(A action);

    /**
```

```
     * Returns the record of all previous actions and which player has done it.
     *
     * @return the record of all previous actions
     */
    List<ActionRecord<A>> getActionRecords();

    /**
     * If the game is in a state of indeterminacy, this method will return an
     * action according to the distribution of probabilities, or hidden
     * information. If the game is in a definitive state null is returned.
     *
     * @return a possible action, which determines the game
     */
    A determineNextAction();
```

Logging

The standard logger implementation provides five levels of logging.

1. Trace (level −2)
2. Debug (level −1)
3. Info (level 0)
4. Warn (level 1)
5. Error (level 2)

A logger can be configured with pre and post strings which are pre- and appended to some of the printed strings.

An API-abiding agent is passed a logger which has the same level as the engine. This can be useful as repeated printing is suboptimal for the performance, however some debug information is sometimes useful.

Every level of logging has a couple of variants. Using `debug` as example:

- `debug` (prints pre, the message, post and newline)
- `deb` (same as `debug` but without newline in the end)

Those two now have multiple variants again:

- `debugf` (prints a formatted string, behaving like `String.format`)
- `debugEnum` (prints a message and a number, mostly used for indicating that something is counted)
- `debugProcess` (prints a message and a progress percentage, as well as the explicit values, mostly used for indicating that something is processed)

Every variant of these have variants again

- `_debug` (Print no pre)
- `debug_` (Print no post)

- `_debug_` (Print no pre and post)

This can be double checked in the javadoc.

Debugging

To effectively debug (in JUnit for example). You can create a new instance of the game with the constructor and an instance of your agent.

```
@Test
public void text_example(){
 ExampleGame exampleGame = new ExampleGame();
 FirstAgent agent = new FirstAgent();

 // Bring game and agent to the required state


 ExampleAction action = agent.determineNextAction(exampleGame, 30, TimeUnit.SECONDS);
 ExampleGame next = (ExampleGame) exampleGame.doAction(action);

 //Test if agent behaves as expected

}
```