Heimlich & Co for the Strategy Game Engine

Clemens Anzinger, 11904646
Vienna University of Technology

March 2023

Contents

1 Introduction

Heimlich & Co is a strategy board game from 1984 for two to seven players. The game features a zero sum reward system, is only partially observable and has elements of chance. The game concepts and rules are rather simple, but they allow for various different strategies when playing the game.
This project aimed at making Heimlich & Co playable with the Strategy Game Engine (SGE), and furthermore, make it easy for students to implement their own agents for the game.
Therefore, the project consisted of three main steps:

- Extending the SGE
- Implementing Heimlich & Co according to the rules and the interfaces given by the SGE
- Implementing three different agents to act as a reference for students implementing their own agents.

A more precise game description including the rules and the general gameplay can be found in *documentation/game_instructions.md* in the repository of the game.
All code is written with the Java SDK 11.

2 Strategy Game Engine Extension

`https://github.com/canzinger/Strategy-Game-Engine-Disqualification-Extension`

The Game interface in the SGE is extended by two methods to allow for disqualifying an agent when it times out. After disqualification, the match will continue until it normally ends (or until too many players time out and get disqualified).
The option to disqualify agents is turned off by default, but can be enabled with the following argument:

- **-dq** or **--disqualify**: Enables disqualification of agents that time out. Available for both the *match* and *tournament* command.

The *Game* interface of the SGE was extended by the following two methods:

```
/**
   * Disqualifies the current player. Can be invoked after a player
   * times out for example. The new game is "as identical
   * as possible" to this game, but with the player disqualified.
   * The state of the game changes as little as possible (meaning the
   * state of the game, board, other players are only changed if
   * absolutely necessary). The utility of the disqualified player is -1
```

```
 * (when calling getUtility(playerId)). The player never becomes
 * the current player again.
 *
 * For some games this might not be possible/feasible, in this case an
 * UnsupportedOperationException is thrown.
 *
 * @return a new game with the given player disqualified
 * @throws IllegalStateException if the player cannot be disqualified
 * because not enough players would remain
 * @throws UnsupportedOperationException if the game does not support
 * disqualification
 */
default Game<A, B> disqualifyCurrentPlayer() {
  throw new UnsupportedOperationException("Disqualifying a player is
      not supported.");
}

/**
 * Returns whether the game supports the disqualification of a player.
 *
 * @return true if the game supports disqualification
 */
default boolean supportsDisqualification() {
  return false;
}
```

All other commands, rules, etc. of the SGE are still valid and can be found in the corresponding manual (with the exception of *-b*, see 3).

3 Heimlich & Co Game

`https://github.com/canzinger/HeimlichAndCo`

The exact rules and gameplay of the game are explained in *documentation/game_instructions.md*.
As Heimlich & Co implements the *Game* interface of the SGE, it also provides all methods described in the SGE manual. Additionally, the methods are documented well via Javadoc comments (Javadoc can be generated with the gradle command *javadoc*). The most important additional methods and variables are repeated here:

- **currentTurnPlayer**: This variable always determines which players turn it is. This is not always equal to *currentPlayer*. Generally, the *currentTurnPlayer* is always the player who last rolled the die. This can be different from *currentPlayer* when playing with cards during the card phase.
- **playersToAgentsMap**: Saves which in-game agent is assigned to which

player. This information is secret. Therefore, when receiving an instance of *HeimlichAndCo* from the SGE, the map will only have one entry corresponding to the current player.

- **cards**: Saves which players have which cards. Similar to *playersToAgentsMap*, this is secret information and the map will only have on entry when receiving a *HeimlichAndCo* instance from the SGE.
- **phase**: Describes which phase the game currently is in. There are four (self-explaining) phases:
  - DIE_ROLL_PHASE
  - AGENT_MOVE_PHASE
  - CARD_PLAY_PHASE
  - SAFE_MOVE_PHASE
- **applyAction(action)**: Has the same purpose as the standard SGE function *doAction(action)*, but does not create a copy of the game (useful when doing simulation in MCTS for example because it is faster).

```
/**
 * Applies an action to this game, DOES NOT create a copy of
 * this game (in contrast to doAction).
 *
 * More performant than doAction (as the game does not have to
 * be copied), but changes THIS instance.
 *
 * @param action action to take
 */
public void applyAction(HeimlichAndCoAction action);
```

- **getUtilityValue(i)**: The utility function for the game. Returns -1 if the player was disqualified. If the game is not over, it returns the current score of a player's in-game agent. If the game is over, either returns the score of a player's in-game agent (in case a "real" in-game agent won), or returns 0 (in case a dummy agent won).
  Only works if the object variable *playersToAgentsMap* contains an entry for the player.

```
/**
 * Returns the utility (usually the current score) of the player
 *
 * @param i player for which utility is wanted
 * @return utility value
 * @throws IllegalArgumentException if there is no entry for
 * the player in the playersToAgentsMap
 */
@Override
```

```
public double getUtilityValue(int i);
```

Running a game of Heimlich & Co without cards and 3 agents and disqualification:

```
sge-1.0.4-dq-exe.jar match libs/HeimlichAndCo-1.0.0.jar
build/libs/HeimlichAndCoDepthSearchAgent-1.0.jar
build/libs/HeimlichAndCoMCTSAgent-1.0.jar
build/libs/HeimlichAndCoRandomAgent-1.0.jar
-p 3
-b 0
-c 30
-dq
```

Running a game of Heimlich & Co with cards and 5 agents and disqualification:

```
sge-1.0.4-dq-exe.jar match libs/HeimlichAndCo-1.0.0.jar
build/libs/HeimlichAndCoDepthSearchAgent-1.0.jar
build/libs/HeimlichAndCoMCTSAgent-1.0.jar
build/libs/HeimlichAndCoRandomAgent-1.0.jar
HeimlichAndCoDepthSearchAgent
HeimlichAndCoMCTSAgent
HeimlichAndCoRandomAgent
HeimlichAndCoMCTSAgent
HeimlichAndCoMCTSAgent
-p 5
-b 1
-c 30
-dq
```

The general structure to run a game is the same as for any game using the SGE. However, there is one option which is different to the normal SGE options:

- **-b**: This option controls whether Heimlich & Co is played with or without cards (in contrast to being an option to play with a custom board with the SGE; a custom board is not supported for Heimlich & Co). The game is played with cards, if the *-b* option is present and the value is either *1*, *cards* or *Cards*. In all other cases, the game is player without cards.

4 Heimlich & Co Agents

`https://github.com/canzinger/HeimlichAndCoAgents`

The agents are a starting point for students implementing their own agents. They show how to interact with the game and display certain approaches to handle random events or the fact that Heimlich & Co is only partially observable.

4.1 Random Agent

The random agents simply takes a random action out of all possible ones.

4.2 Depth Search Agent

The Depth Search Agent showcases a classic MiniMax depth search without any big improvements. It does not feature good time management, but rather the depth can be set before compilation (**TERMINATION_DEPTH**). Because the tree explodes in complexity, the default value is 3.
The agent builds a tree with all possible actions with a depth of 3. Then, the agent evaluates a game state by checking how far ahead or behind the player's in-game agent is to other in-game agents. These game state evaluations are calculated for every leaf node and backpropagated to the root node.

To even be able to build the tree, the agent "determinizes" the game, meaning it sets which agent belongs to which player (done randomly) and if playing with cards, also sets which cards belong to which player (other players are assumed to have no cards), as that information is removed by the game (for more information, see section 4.4).

In terms of strength, this agent is weaker than the MCTS agent, but occasionally might win against it.

4.3 Monte Carlo Tree Search Agent

The Monte Carlo Tree Search Agent features basic Monte Carlo Tree Search. It showcases some options to deal with randomness and partial observability. There are the following options in the *HeimlichAndCoMCTSAgent.java* class which can be adjusted before compilation:

- **TERMINATION_DEPTH**: Controls the depth of random playouts (in the simulation phase). A lower value leads to the agent being able to do more search iterations in the given computation time, but decreases the confidence of the simulation. The default value is 64, a value of -1 means the random playouts are played out until the game ends.
- **SIMULATE_ALL_DIE_OUTCOMES**: Controls how to deal with randomness during the selection phase of the MCTS. The default value for this is true (as it is the stronger setting).
  If set to true, means that all possible outcomes will be added to the tree, and then selected randomly (meaning after all outcomes are added to the tree, nodes for further exploration are chosen randomly). The idea behind this is that in the real game it is not possible to choose the outcome of a die roll, meaning this should make the MCTS more robust.
  If set to false, means that during selection the die is rolled once, and that outcome will be added to the tree. Other possible outcomes are ignored. This setting means the MCTS ignores 5 out of 6 possible outcomes, which

means it does not get a good overview about how good the current branch is.

For dealing with partial observability, the MCTS does the same as the Depth Search Agent (see Section 4.2). Before MCTS even takes place, it "determinizes" the game, meaning assigning agents to players and cards to players. This happens in the following method:

```
/**
 * Adds information that was removed by the game (i.e. hidden
 * information). Therefore, adds entries to the map which maps
 * agents to players and entries to the map mapping the cards of
 * players.
 * The agents are randomly assigned to players. And players are
 * assumed to have no cards.
 * @param game to add information to
 */
private void addInformationToGame(HeimlichAndCo game);
```

This is a very basic approach to this problem with lot of room for improvements and other strategies. However, it is necessary to do, as otherwise the game can not be played out in the simulation phase.

4.4 Writing your own Agents and Setup

Any agent for Heimlich & Co has to implement the *GameAgent<HeimlichAndCo, HeimlichAndCoAction>* interface from the SGE, additionally it is recommended to also extend the *AbstractGameAgent* class for additional functionality. For further information and the general setup of agents, consult the manual of the SGE. The *Game* interface is also explained there.

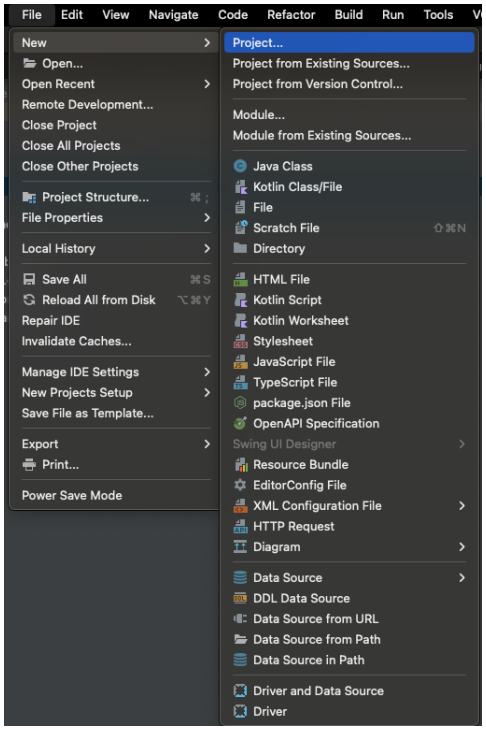
During play, the agent receives an instance of *HeimlichAndCo* from the engine in the *computeNextAction(...)* method. The received instance has some information removed which is supposed to remain secret. More specifically, the *playersToAgentsMap* only has one entry, which is for the current player (this way the agent can also find out which player it is). The same holds for the *cards* map (when playing with cards). When the agent does Depth Search or MCTS it is necessary to assign players their cards (or at least an empty list), otherwise the game cannot be played out and will throw an exception. It is also recommended to add additional entries to the *playersToAgentsMap* to assign agents to players, otherwise some methods (like *getUtilityValue()*, will throw exceptions). An example of how to do this can be seen in the MCTS and DepthSearchAgent.

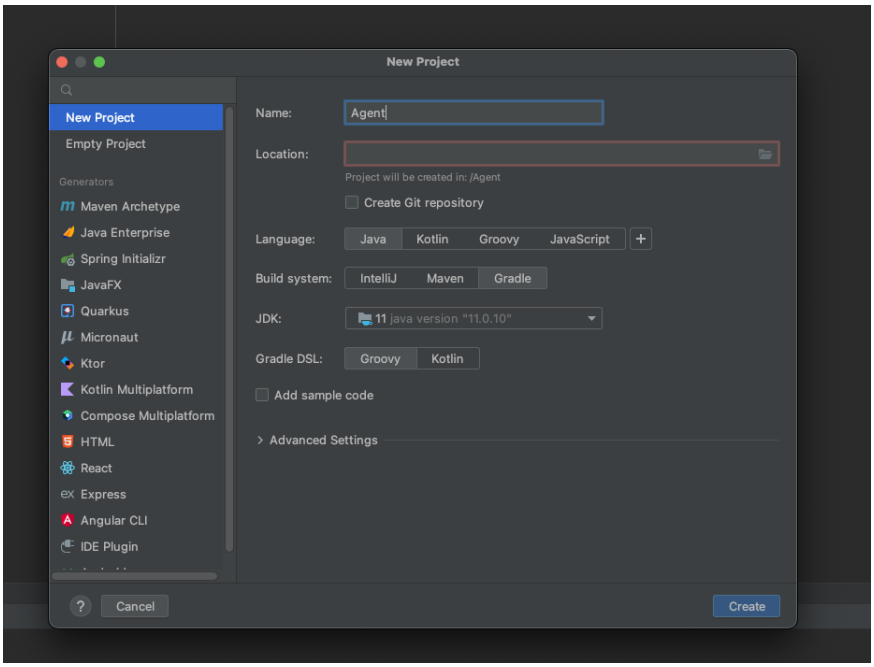
4.4.1 Setup

Here, an example of how to setup the project in IntelliJ is given. Note that this example provides the option to develop multiple agents in the same project

while keeping them separated, which might not be needed (but it can certainly be useful for trying different things).

First, create a new project and configure it with IntelliJ using the following actions:



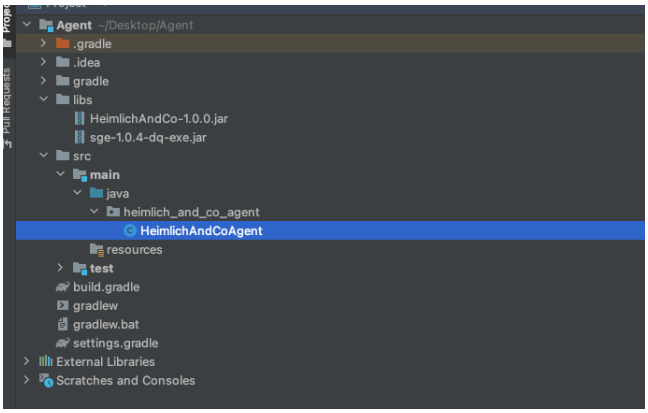


With the two actions above completed, you should now have a new, empty project. In this project, you should see a *build.gradle* file. Substitute the content with the following (can also be found in *documentation/build_example.gradle*):

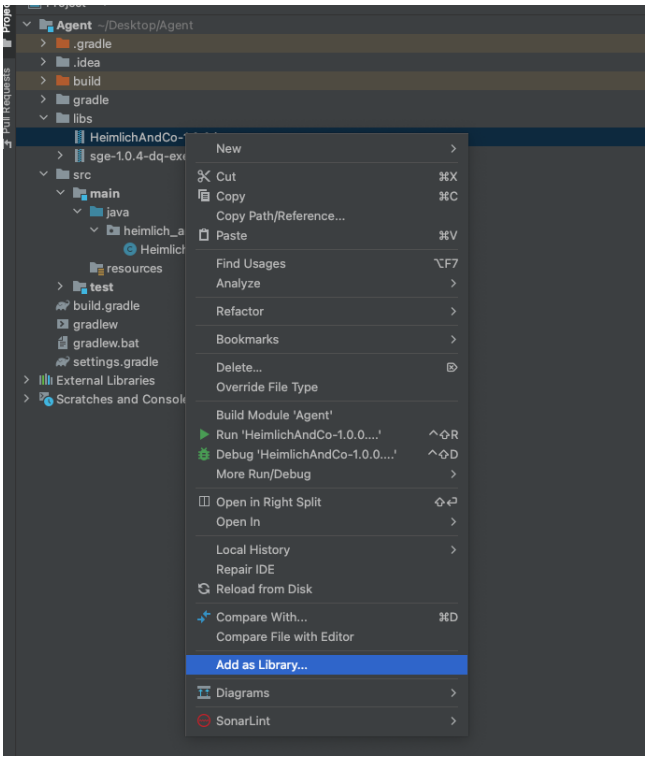
```
plugins {
id 'java'
}

sourceCompatibility = 1.11

def agent_name = 'HeimlichAndCoAgent'
def agent_package = 'heimlich_and_co_agent'

//def agent_name = 'Agent2'
//def agent_package = 'heimlich_and_co_agent2'

group 'org.example'
version '1.0-SNAPSHOT'

repositories {
    mavenCentral()
}

dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.8.1'
```

```
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.8.1'
    implementation files('libs/HeimlichAndCo-1.0.0.jar')
    implementation files('libs/sge-1.0.4-dq-exe.jar')
}

test {
    useJUnitPlatform()
}

jar {
    manifest {
        attributes 'Sge-Type': 'agent'
        attributes 'Agent-Class': agent_package +'.'+agent_name
        attributes 'Agent-Name': agent_name
    }
    archivesBaseName = agent_name
    includes = [agent_package +'/*']
}
```

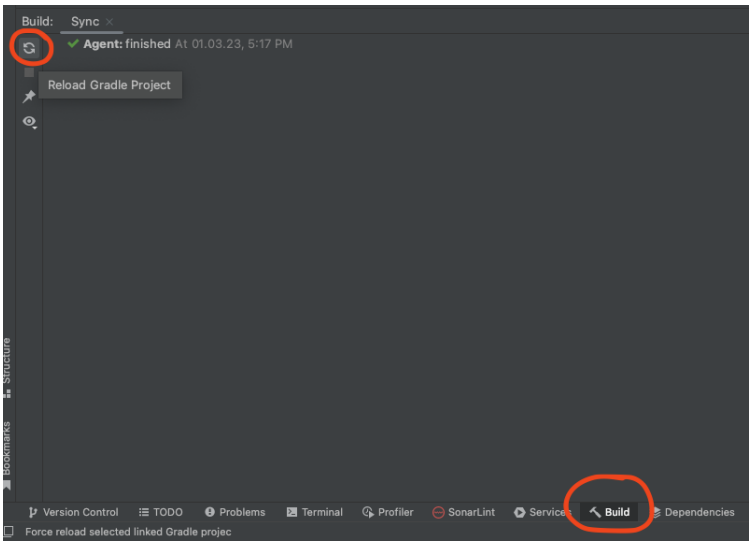
Then, create a package in *source/main/java* for the first agent with the same name as specified in the *build.gradle* (you can of course change the name). Inside the newly created package, create a java class with the same name as specified in the *build.gradle*.
Then, create a folder in the top-level of the project called *libs*, and copy the SGE and Heimlich&Co libraries (.jar files) into it (you can find the jars in the HeimlichAndCo repository).
The current state of the project should be as shown below:



Then add the two libraries as libraries in IntelliJ (the dialog that pops up can just be confirmed with ok):



The last step is to reload the gradle project:



With the steps showed above, you should now be good to go to start programming an agent. To confirm everything works correctly, you can take a look at

the code of the RandomAgent and copy it into your agent. An agent can then be built with:

```
./gradlew jar
```

It will be saved as a .jar file in *build/libs*. A game of Heimlich&Co can then be run as described above in Section 3.

4.4.2 Tips

- There are lots of strategies to deal with partial observability, the one showed in the example agents are very basic and simple. One could for example count how many cards each player has left and learn which in-game agent belongs to which player, to improve on the given implementation. There is also plenty of literature available on how to generally deal with partial information. For example, Maciej Świechowski and Tomasz Tajmaje show a practical solution on how to deal with imperfect information in practice [1] and Maciej Świechowski and others describe a few different approaches in a recent survey paper about MCTS [2].

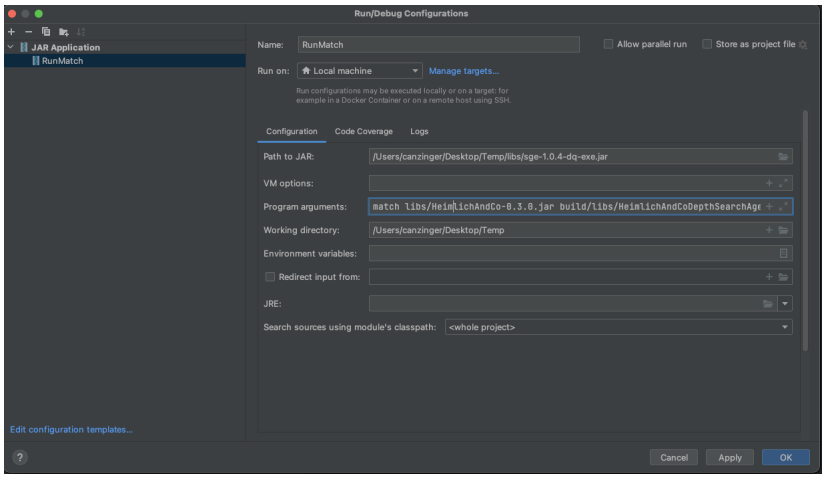
- When doing a tree-based strategy, the tree quickly explodes in width. Therefore, trimming the tree, focussing on promising actions, grouping together similar actions might be able to provide improvements.

- As explained in Section 3, it might happen that an in-game agent wins, that does not belong to a real player (i.e. a dummy agent wins). This makes some interesting strategies possible, like trying to make a dummy agent win to force a tie when the own agent is too far behind and has no chance of winning.

- Many useful functions for agents are already implemented in Heimlich & Co, so before implementing a function, checking if a function already exists which does what is needed might save some time (there are especially some useful functions in the *HeimlichAndCoBoard* class).

- Generally, the *HeimlichAndCo* class and other classes allow you to do what you want with an instance (e.g. changing variable values, moving agents, ...). This makes it possible to easily set up and try out scenarios, which might provide insight on what an AI agent will do in a given situation.

- You can use IntelliJ to run games and also to debug your agent as you are used to with other programs. To be able to do this, add a "Run/Debug Configuration" in IntelliJ describing a Jar Application, the configuration may look like this:



In the field "Program arguments" you have to provide the arguments as described in Section 3 and/or in the SGE (path to game, path to agents, other arguments).

References

[1] Maciej Świechowski and Tomasz Tajmajer. "A Practical Solution to Handling Randomness and Imperfect Information in Monte Carlo Tree Search". In: *2021 16th Conference on Computer Science and Intelligence Systems (FedCSIS)*. 2021, pp. 101–110. DOI: 10.15439/2021F3.

[2] Maciej Świechowski et al. "Monte Carlo Tree Search: a review of recent modifications and applications". In: *Artificial Intelligence Review* 56.3 (2023), pp. 2497–2562. DOI: 10.1007/s10462-022-10228-y. URL: https://doi.org/10.1007/s10462-022-10228-y.