

e-print <http://www.gm.fh-koeln.de/ciopwebpub/Kone17a.d/TR-GBG.pdf>

# The GBG Class Interface Tutorial: General Board Game Playing and Learning

Wolfgang Konen

Cologne Institute Computer Science,  
TH Köln,  
University of Applied Sciences,  
Germany

[wolfgang.konen@th-koeln.de](mailto:wolfgang.konen@th-koeln.de)

Last update: March 2019

(Initial version: June 2017)

## **Abstract**

This technical report introduces GBG, the general board game playing and learning framework. It is a tutorial that describes the set of interfaces, abstract and non-abstract classes which help to standardize and implement those parts of board game playing and learning that otherwise would be tedious and repetitive parts in coding. GBG is suitable for arbitrary 1-player, 2-player and  $N$ -player board games. It provides a set of agents (AI's) which can be applied to any such game. This document describes the main classes and design principles in GBG. GBG is written in Java.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Motivation	4
1.2	Related Work	5
1.3	Introducing GBG	6
<b>2</b>	<b>Class and Interface Overview</b>	<b>7</b>
<b>3</b>	<b>Classes in Detail</b>	<b>7</b>
3.1	Interfaces StateObservation and StateObsNondeterministic	7
3.2	Interface PlayAgent and class AgentBase	8
3.2.1	List of Agents implemented in GBG	9
3.3	Some Remarks on the Game Score	10
3.4	Difference between Game Score and Game Value	12
3.5	Interface Feature	12
3.6	Interface XNTupleFuncs	13
3.7	Interface GameBoard	13
3.8	Human interaction with the board and with Arena	14
3.9	Abstract Class Evaluator	14
3.10	Abstract Class Arena	15
3.11	Abstract Class ArenaTrain	16
3.12	The Param Classes	17
<b>4</b>	<b>Use Cases and FAQs</b>	<b>17</b>
4.1	I have implemented game XYZ and want to use AI agents from GBG – what do I have to do?	17
4.2	How to train an agent and save it	18
4.3	Which AI's are currently implemented for GBG?	18
4.4	How to write a new agent (for all games)?	19
4.5	What is the difference between TDAgent and TDNTuple3Agt?	19
4.6	How to specialize the TDAgent to a new game?	20
4.7	How to specialize the TDNTuple3Agt agent to a new game?	20
4.8	What is a ScoreTuple?	22
<b>5</b>	<b>Open Issues</b>	<b>22</b>
<b>A</b>	<b>Appendix: Interface Summary</b>	<b>23</b>
A.1	Interfaces StateObservation and StateObsNondeterministic	23
A.2	Interface GameBoard	24
A.3	Interface PlayAgent	25
A.4	Classes ACTIONS, ACTIONS_VT and ACTIONS_ST	26
A.5	Interface Feature	26
A.6	Interface XNTupleFuncs	27
A.7	Abstract class Evaluator	27

A.8	Abstract class Arena	28
A.9	Abstract class ArenaTrain	29
<b>B</b>	<b>Appendix: Other Game Value Functions</b>	<b>29</b>
<b>C</b>	<b>Appendix: N-Tuples</b>	<b>32</b>
C.1	Board Cell Numbering	32
C.2	N-Tuple Creation	32
C.3	N-Tuple Training and Prediction	34
<b>D</b>	<b>Appendix: Multi-core threads</b>	<b>35</b>
<b>E</b>	<b>Appendix: String representations of agents</b>	<b>35</b>

# 1 Introduction

## 1.1 Motivation

General board game (GBG) playing and learning is a fascinating area in the intersection of machine learning, artificial intelligence and game playing. It is about how computers can learn to play games not by being programmed but by gathering experience and learning by themselves (self-play). The learning algorithms are often called AI agents or just „AI“s (AI = artificial intelligence). There is a great variety of learning algorithms around, e.g. reinforcement learning algorithms like  $TD(\lambda)$ , Monte Carlo tree search (MCTS), different neural network algorithms, Minimax, ... to name only a few.

Even if we restrict ourselves to board games, as we do in this paper (and do not consider other games like video games), there is a plethora of possible board games where an agent might be active in. The term „General“ in GBG refers to the fact that we want to have in the end agents or AIs which perform well on a large variety of games. There are quite different games: 1-person games (like Solitaire, 2048, ...), 2-person games (like Tic-Tac-Toe, Othello, Chess, ...), many-person games (like Settlers of Catan, Poker, ...). The game environment may be deterministic or it may contain some elements of chance (like rolling the dices, ...).

A common problem in GBG is the fact, that each time a new game is tackled, the AI developer has to undergo the frustrating and tedious procedure to write adaptations of this game for all agent algorithms. Often he/she has to reprogram many aspects of the agent logic, only because the game logic is slightly different to previous games. Or a new algorithm or AI is invented and in order to use this AI in different games, the developer has to program instantiations of this AI for each game.

Wouldn't it be nice if we had a framework consisting of classes and interfaces which abstracts the common processes in GBG playing and learning? If someone programs a new game, he/she has just to follow certain interfaces described in the GBG framework, and then can easily use and test on that game *all* AIs in the GBG library.

Likewise, if an AI developer introduces a new learning algorithm which can learn to play games, she has only to follow the interface for agents laid down in the GBG framework. Then she can test this new agent on *all* games of GBG. Once the interface is implemented she can directly train her agent, inspect its move decisions in each game, test it against other agents, run competitions, enter game leagues, log games and so on.

The rest of this document introduces the class concept of GBG. After a short (and probably incomprehensive) summary of related work in Sec. 1.2, Sec. 2 gives an overview of the relevant classes and Sec. 3 discusses them in detail. Sec. 4 discusses some use cases and FAQs for the GBG class framework. Appendix A lists the methods of the important classes and interfaces, Appendix B gives more details on game value functions, Appendix C introduces n-tuples, and Appendix D describes the tasks in GBG which are multi-core parallelized.

## 1.2 Related Work

One of the first general game-playing systems was Pell's METAGAMER [Pell, 1996]. It played a wide variety of simplified chess-like games.

Later, the discipline **General Game Playing (GGP)** [Genesereth and Thielscher, 2014, Mańdziuk and Świechowski, 2012] became a wider coverage and it has now a long tradition in artificial intelligence: Since 2005, an annual GGP competition organized by the Stanford Logic Group [Genesereth et al., 2005] is held at the AAAI conferences. Given the game rules written in the so-called *Game Description Language (GDL)* [Love et al. [2008]], several AIs enter one or several competitions. As an example for GGP-related research, Mańdziuk and Świechowski [2012] propose a universal method for constructing a heuristic evaluation function for any game playable in GGP. With the extension *GDL-II* [Thielscher, 2010], where *II* stands for „Incomplete Information“, GGP is able to play games with incomplete information or nondeterministic elements as well.

GGP solves a tougher task than GBG: The GGP agents learn and act on previously unknown games, given just the abstract set of rules of the game. This is a fascinating endeavour in logic reasoning, where all information about the game (game tactics, **game symmetries** and so on) is distilled from the set of rules at *run time*. But, as Świechowski et al. [2015] has pointed out, arising from this tougher task, there are currently a number of limitations or challenges in GGP which are hard to overcome within the GGP-framework:

- Simulations of games written in GDL are slow. This is because math expressions, basic arithmetic and loops are not part of the language.
- Games formulated in GDL have suboptimal performance as a price to pay for its universality: This is because „it is almost impossible, in a general case, to detect what the game is about and which are its crucial, underpinning concepts.“ [Świechowski et al., 2015]
- The use of Computational Intelligence (CI), most notably neural networks, deep learning and TD (temporal difference) learning, have not yet had much success in GGP. As Świechowski et al. [2015] writes: „CI-based learning methods are often too slow even with specialized game engines. The lack of game-related features present in GDL also hampers application of many CI methods.“ Michulke and Thielscher [2009], Michulke [2011] presented first results on translating GDL rules to neural networks and TD learning: Besides some successes they faced problems like overfitting, combinatorial explosion of input features and slowness of learning.

GBG aims at offering an alternative with respect to these limitations, as will be further exemplified in Sec. 1.3. It has not the same universality as GGP, but agents from the CI-universum (TD, SARSA, deep learning, ...) can train and act fast on all available games.

Other works with relations to GBG: *General Video Game Playing* (GVGP, Levine et al. [2013]) is a related field which tackles video games instead of board games. Likewise,  $\mu$ RTS [Ontanón and Buro, 2015, Barriga et al., 2017] is an educational framework for AI agent testing and competition in real-time strategy (RTS) games. *OpenAI Gym* [Brockman

et al., 2016] is a toolkit for reinforcement learning research which has also a board game environment supporting a (small) set of games.

### 1.3 Introducing GBG

We define a **board game** as a game being played with a known number of players,  $N = 1, 2, 3, \dots$ , usually on a game board or on a table. The game proceeds through actions (moves) of each player in turn. This differentiates board games from video or RTS games where usually each player can take an action at any point in time. Note that our definition of board games includes (trick-taking) card games (like Poker, Skat, ...) as well. Board games for GBG may be deterministic or nondeterministic.

What differentiates GBG from GGP? – GBG has not the same universality than GGP in the sense that GBG does not allow to present new, previously unknown games at *run time*. However, virtually any board game can be added to GBG at *compile time*. GBG then aims at overcoming the limitations of GGP as described in Sec. 1.2:

- GBG allows fast game simulation due to the compiled game engine (10.000-90.000 moves per second for TD-agents on a single core.
- The game or AI implementer has the freedom to define game-related features or **symmetries** (see Sec. 3.6 and Appendix C.3) at compile time which she believes to be useful for her game. Symmetries can greatly speed up game learning.
- GBG offers various CI agents, e.g. TD- and SARSA-agents and – for the first time – a **generic** implementation of TD-n-tuple-agents (see Sec. 3.2), which can be trained fast and can take advantage of game-related features. With *generic* we mean that the n-tuples are defined for arbitrary game boards (hexagonal, rectangular or other) and that the same agent can be applied to 1-, 2-, ...,  $N$ -player games.
- For evaluating the agent's strength in a certain game it is possible to include game-specific agents which are strong or perfect player for that game. Then the *generic* agents (e. g. MCTS or TD) can be tested against such specific agents in order to see how near or far from strong/perfect play the generic agents are on that game.<sup>1</sup> It is important to emphasize that the generic agents do *not* have access to the specific agents during game reasoning or game learning, so they cannot extract game-specific knowledge from the other strong/perfect agents.
- Each game has a game-specific visualization and an inspect mode which allows to inspect in detail how the agent responds to certain game situations. This allows to get deeper insights where a certain agent performs well or where it has still remarkable deficiencies and what the likely reason is.

---

<sup>1</sup>Note that in GGP agents are compared with other agents from the GGP league. A comparison with strong/perfect game-specific (non-GGP) agents is usually not made.

GBG is written in Java and supports parallelization of multiple cores for time-consuming tasks. It is available as open source from GitHub<sup>2</sup> and as such – similar to GGP – well-suited for educational and research purposes.

## 2 Class and Interface Overview

Interface **StateObservation** is the main interface a game developer has to implement once he/she wants to introduce a new game. A class derived from **StateObservation** observes a game state, it can infer from it the available actions, knows when the game is over, can advance a state into a new legal state given one of the available actions. If a random ingredient from the game environment is necessary for the next action (of the next player), the advance function will add it.

The second interface a game developer has to implement is the interface **GameBoard**, which realizes the board GUI and the interaction with the board. If one or more humans play in the game, they enter their moves via **GameBoard**.

The interface an AI developer has to implement is the interface **PlayAgent**. It represents an „AI“ or agent capable of playing games. If necessary, it can be trained by self-play. Once trained, it has methods for deciding about the best next action to take in a game state **StateObservation** and getting the agent's estimate of the score or value of a certain game state.

The heart of GBG are the abstract classes **Arena** and **ArenaTrain**. In the **Arena** all agents meet: They can be loaded from disk, they play a certain game, there can be competitions. In **ArenaTrain**, which is a class derived from **Arena**, there are additional options to parametrize, train, inspect, evaluate and save agents.

The helper classes **Feature**, **XNTupleFuncs**, **Evaluator**, and **ACTIONS** (plus **ACTIONS\_VT**, **ACTIONS\_ST**) support the abstraction in the classes **Arena** and **ArenaTrain**.

## 3 Classes in Detail

### 3.1 Interfaces **StateObservation** and **StateObsNondeterministic**

Interface **StateObservation** observes the current state of the game, it has utility functions for

- returning the available actions (`getAvailableActions()`),
- advancing the state of the game with a specific action (`advance()`),
- copying the current state
- getting the score of the current state  
(`getScore(StateObservation referringState)`)

---

<sup>2</sup><https://github.com/WolfgangKonen/GBG>

- signaling end and winner of the game

If a game has random elements (like rolling the dices in a dice game or placing a new tile in 2048), `advance()` is additionally responsible for invoking such random actions and reporting the results back in the new state. Examples:

- For a dice-rolling game: the game state is the board & the dice number.
- For 2048: the game state is just the board (with the random tile added).

Implementing classes: `StateObserverTTT`, `StateObserver2048`, ..., `ObserverBase`.

As an example, `StateObserverTTT` is a state observer for the game `TicTacToe`: It has constructors with game-specific parameters (`int [][] table, int player`). It has access functions `getTable()` and `getPlayer()`. The latter returns the player who has to move in the current state.

Some methods, e.g. setters, getters and other common methods, have their defaults implemented in abstract class **ObserverBase**. It is recommended to derive a new `StateObserver` class from class `ObserverBase`.

Interface **StateObsNondeterministic** is derived from **StateObservation** and provides functionality around nondeterministic actions. Examples using or implementing **StateObsNondeterministic** are **ExpectimaxNAgent** and **StateObserver2048**.

### 3.2 Interface PlayAgent and class AgentBase

Interface **PlayAgent** has all the functionality that an AI (= game playing agent) needs. The most important methods are:

- `getNextAction2(sob, ...)`: given the current game state `sob`, return the best next action.
- `double getScore(sob)`: the score (agent's estimate of final reward) for the current game state `sob`.
- `trainAgent(sob, ...)`: train agent for one episode<sup>3</sup> starting from state `sob`.

Some more methods, e.g. setters and getters, have their defaults implemented in class **AgentBase**.<sup>4</sup> It might be useful to design a new agent class with the signature

```
... extends AgentBase implements PlayAgent}.
```

<sup>3</sup>An *episode* is one specific game payout.

<sup>4</sup>**AgentBase** does not implement all methods of the interface `PlayAgent`, so it has no `... implements PlayAgent`.



There is an additional method `double estimateGameValue(sob)` which has the default implementation `getScore(sob)` in **AgentBase**. This method is called when a training game is stopped prematurely because the maximum number of moves in an episode ('Episode length') is reached.<sup>5</sup> See Sec. 3.3, 3.4 and Appendix B for more details on **game score** and **game value**.

### 3.2.1 List of Agents implemented in GBG

Classes implementing interface **PlayAgent** and derived from **AgentBase**:

- **RandomAgent**: an agent acting completely randomly
- **HumanPlayer**: an agent waiting for user interaction
- **MinimaxAgent**: a simple tree search (max-tree for 1-player games, min-max-tree for 2-player games). Deprecated, better use **MaxNAgent** or **ExpectimaxAgent** for deterministic and nondeterministic games, resp.<sup>6</sup>
- **MaxNAgent**: the generalization of Minimax to N-player games with arbitrary N (see [Korf \[1991\]](#)). It maximizes the  $k$ th score in a score tuple.
- **ExpectimaxNAgent**: the generalization of **MaxNAgent** to nondeterministic games: alternating layers of chance nodes and expectimax nodes.
- **MCAgent**: Monte-Carlo agent (no tree)
- **MCTSAgent**: Monte-Carlo Tree Search agent
- **MCTSExpectimaxAgt**: Monte-Carlo Tree Search agent for non-deterministic games: alternating layers of chance nodes and expectimax nodes. See [Kutsch \[2017\]](#) for more details.
- **TDAgent**: general  $TD(\lambda)$  agent (temporal difference reinforcement learning) with neural network value function (see Sec. 4.6 for more details). This agent requires a **Feature** object in constructor, see Sec. 3.5.
- **TDNTuple2Agt**:  $TD(\lambda)$  agent with n-tuple sets. Deprecated, use **TDNTuple3Agt** instead.
- **TDNTuple3Agt**:  $TD(\lambda)$  agent (temporal difference reinforcement learning) using n-tuple sets as features (see Sec. 4.7 and Appendix C for more details). This agent requires an object of class **XNTupleFuncs** in constructor, see Sec. 3.6.

<sup>5</sup>Or, for agents MCTS or MC, when the maximum rollout depth ('Rollout depth') is reached.

<sup>6</sup>Note that Minimax is only for 2-player games, while MaxN is for 1-, 2-,..., N-player games. Note that Minimax in this simple implementation may not be appropriate for games with random elements, because Minimax follows in each tree step only *one* path of the possible successors that `advance()` may produce.

- SarsaAgt: SARSA agent (SARSA is a variant of Q-learning with state-action-pairs) using n-tuple sets as features (see Appendix C for more details). This agent requires an object of class `XNTupleFuncs` in constructor, see Sec. 3.6.

The last four agents are  $TD(\lambda)$  agents which learn by reinforcement (temporal difference). The last two agents (`TDNTuple3Agt` and `SarsaAgt`) are based on n-tuple features. `TDNTuple3Agt` is the generic implementation of a TD-n-tuple-agent. More details on  $TD(\lambda)$  (temporal difference learning, reinforcement learning for games, eligibility traces) can be found in the technical report [Konen \[2015\]](#).

Each agent has an **AgentState** member, which is either RAW, INIT or TRAINED.

Some of the agents (`RandomAgent`, `HumanAgent`, `MinimaxAgent`, `MaxNAgent`, `ExpectimaxNAgent`, `MCAgent`, `MCTSAgent`, `MCTSExpectimaxAgt`) are directly after construction in a TRAINED state, i.e. they are ready-to-use. They make their observations on-the-fly, starting from the given state. Other agents (`TDAgent`, `TDNTuple2Agt`, `TDNTuple3Agt`, `SarsaAgt`, ...) require training, they are after construction in state INIT.

Classes derived from `PlayAgent` should implement the `Serializable` interface. This is needed for loading and saving agents. Agent members which need *not* to be included in the serialization process can be flagged with keyword `transient`. Agent members which are user-defined classes should implement the `Serializable` interface as well.

### 3.3 Some Remarks on the Game Score

Although the game score (the final result of a game, e. g. „X wins“ or „O wins with that many points“) seems to be a pretty simple and obvious concept, it becomes a bit more confusing if one wants to define the game score consistently for a broader class of states, not just for a terminal state. We use the following conventions:

- For `StateObservation so`,

```
so.getGameScore(StateObservation refer)
```

returns the sum of rewards in state `so`, seen from the perspective of the player to move in state `refer`. Most 2-player games will give the reward only in the end (win/tie/loss), so that for those games `so.getGameScore(refer)` is usually 0 as long as `so` is non-terminal. If the game state is terminal, a negative reward will be returned if `refer`'s player loses and a positive reward if this player wins. For other games there might be also rewards during the game.

- If a state is terminal (e. g. „X wins“) then the „player who moves“ has changed a last time (i. e. to player O, although the game is over.). Thus the score for O will be  $-1$  („O loses“). It seems a bit awkward at first sight to assign a terminal state a „player to move“, but this is the only way to guarantee in a succession of actions for 2-player games that the current score is always the negative of the next state's score (negamax principle). Fig. 1 shows an example.

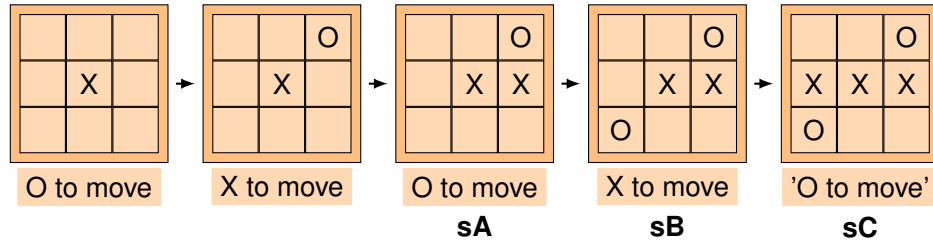


Figure 1: A succession of states in TicTacToe: If O makes in state **sA** the losing move leading to **sB**, then **sB** is a clear win for X, and so is the terminal state **sC**. The **game score** for **sC** from the perspective of O, the player to move in **sC**, is  $-1$ . The **game values** (see Sec. 3.4) for **sB** and **sC** are  $+1$  and  $-1$ , resp.

*Example: State **sC** in Fig. 1 (TicTacToe) is a terminal state: X has made a winning move. On this terminal state O would have to move next (if it were not terminal). So the game score for this terminal state is a negative reward  $sC.getGameScore(sC) = -1$  for player O.*

```
sC.getGameScore(sC) = -1;
sC.getGameScore(sB) = +1;
sC.getGameScore(sA) = -1;
```

- There is the (now deprecated) method `sC.getGameScore()`, which is the game score of **sC** for the player to move in **sC**. This is deprecated, because it can be expressed equivalently as

```
sC.getGameScore() = sC.getGameScore(sC);
```

- `StateObservation.getGameWinner()` may only be called if the game is over for the current state (otherwise an assertion fires). It returns an enum `Types.WINNER` which may be one out of `{PLAYER_WINS, TIE, PLAYER_LOSES}`. The player is always the player who has to move. The method `Types.WINNER.toInt()` converts these enums to integers which correspond to  $\{+1, 0, -1\}$ , resp.

**StateObservation** defines two methods

```
public double getMinGameScore();
public double getMaxGameScore();
```

These methods should return the minimum and maximum game score which can be achieved in a specific game. This is needed since some **PlayAgent** (e.g. `TDAgent`) make predictions of the estimated game score with the help of a neural network. Since a neural network has often a sigmoid output function which can emit only values in a certain range (e.g.  $[0,1]$ ), it is necessary to map the game scores to that range as well. This can only be done if the minimum and maximum game score is given.<sup>7</sup>

<sup>7</sup>If a precise maximum game score for a certain game is not known, a reasonable 'big' estimate is usually also sufficient.

### 3.4 Difference between Game Score and Game Value

There is a subtle distinction between game score and game value. The **game score** is the score of a game according to the game laws. For example, TicTacToe has the score 0 for all intermediate states, while a terminal state has either +1/0/-1 as game score for win/draw/loss of the player to move. In 2048, the game score is the cumulative sum of all tile merges. Each player usually wants to maximize the expectation value of 'his' score at the end of the game.

But the score in an intermediate game state is not a good indicator of the *potential* of that state. The **game value** of a state is an estimate of the final score attainable from that state. Two states in 2048 might have the same score, but the game value of these states can be different. While the first state might be close to the terminal state, the second one might have a higher mobility and thus 'last' longer and receive a higher final score. The precise game value of a state is often not known / not computable, but it is of course desirable to estimate it. An estimate can be based on a simple heuristic like the weighted piece count in chess.

The main possibility to deliver a game value is:

- `PlayAgent.getScore(StateObservation so)` returns the agent's estimate of the final score for the player **who has to move** in `StateObservation so` – assuming perfect play of that player. That is what we call the **game value** of `so`. The game value for 2-player games is usually +1 if it is expected that the player wins finally, 0 if it is a tie and -1 if he loses. Values in between characterize expectation values in cases where different outcomes are possible or likely (or where the agent has not yet gathered enough information or experience).

There are other methods to deliver a game value or a reward

- `StateObservation.getReward(refer,boolean)`
- `PlayAgent.estimateGameValue(StateObservation so)`

but they are only for advanced users and their description is deferred to Appendix B.

### 3.5 Interface Feature

Some classes implementing `PlayAgent` need a game-specific feature vector. As an example, consider **TDAgent**, the general  $TD(\lambda)$  agent (temporal difference reinforcement learning) with neural network value function. To make the neural network predict the value of a certain game state, the network needs some feature input (e.g. specific board patterns which form threats or opportunities, number of them, number of pieces and so on). These features are usually game-specific. We assume here that every feature can be expressed as double value (neural networks can only digest real numbers as input), so that the whole feature vector can be expressed as `double[]`.

To create an **Feature** object within the general `Arena`-code, the *factory method pattern* is used: `Arena` defines an abstract method

```
public Feature makeFeatureClass(int featmode);
```

The argument `featmode` allows to construct different flavors of **Feature** objects and to test and evaluate them.

In all cases where **Arena** or **ArenaTrain** needs a **Feature** object, it will call this method `makeFeatureClass(int)`. This will take place whenever a **TDAgent** object is constructed, because the **TDAgent** constructor needs a **Feature** object as parameter.

Interface **Feature** has the method

```
public double[] prepareFeatVector(StateObservation so);
```

which gets a game state and returns a double vector of features. This vector may serve as an input for a neural network or other purposes.

Implementing classes: **FeatureTTT**, **Feature2048**, ...

### 3.6 Interface **XNTupleFuncs**

There is one special agent **TDNTuple3Agt**, which realizes TD-learning with n-tuple features. N-tuple features or n-tuple sets (Lucas [2008], Thill et al. [2014], Bagheri et al. [2015], Thill [2015]) are another way of generating a large number of features. An n-tuple is a set of board cells. For every game state **StateObservation** it can translate the position values present in these cells into a double score or value. In order to construct such n-tuples, the user has to implement the interface **XNTupleFuncs**. See Sec. 4.7 for more details on the member functions of **XNTupleFuncs** and Appendix C for a more in-detail description of n-tuples.

To create an **XNTupleFuncs** object within the general **Arena**-code, the *factory method pattern* is used: **Arena** defines an abstract method

```
public XNTupleFuncs makeXNTupleFuncs();
```

Whenever **Arena** or **ArenaTrain** needs a **XNTupleFuncs** object, it will call this method `makeXNTupleFuncs()`. This will take place whenever a **TDNTuple3Agt** object is constructed, because the **TDNTuple3Agt** constructor needs an **XNTupleFuncs** object as parameter.

*Note:* If you do not plan to use **TDNTuple3Agt** in your game, you do not need to implement a specific version of class **XNTupleFuncs** in your game. Since you do not construct an object of class **TDNTuple3Agt**, `makeXNTupleFuncs()` should never be called. If it is called nevertheless, the default implementation of `makeXNTupleFuncs()` in **Arena** will throw a **RuntimeException**.

### 3.7 Interface **GameBoard**

Interface **GameBoard** has the game board GUI (usually in a separate **JFrame**). It provides functionality for:

- Maintaining its own **StateObservation** object `m_so`. This object is after construction in a default start state (e. g. empty board). The same state can be reached via `clearBoard()` or `getDefaultStartState()` as well. The associated GUI will show the default start state.
- Showing or updating the current game state (**StateObservation**) in the GUI and enabling / disabling the GUI elements (`updateBoard(...)`).
- Human interaction with the board: see Sec. 3.8.
- Returning its current **StateObservation** object (`getStateObs()`).
- `chooseStartState01()`: This method returns randomly one out of a set of different start states. This is useful when training an agent so that not always the same game episode is played but some variation (exploration) occurs.

*Example TicTacToe: The implementation in `GameBoardTTT` returns with probability 0.5 the default start state (empty board) and with probability 0.5 one of the possible next actions (an 'X' in any of the nine board positions).*

Implementing classes: `GameBoardTTT`, `GameBoard2048`, ...

### 3.8 Human interaction with the board and with **Arena**

During game play: How is the integration between user actions (human moves) and AI agent actions implemented?

If **GameBoard** request an action from **Arena**, then its method `isActionReq()` returns `true`. This causes the selected AI to perform a move. If on the other hand a human interaction is requested, **Arena** issues a `setActionReq(false)` and this causes `isActionReq()` to return `false` as well. **GameBoard** then waits for GUI events until a user (human) action is recorded. **GameBoard** is responsible for checking whether the human action is legal (`isLegalAction()`).<sup>8</sup> If so, then **GameBoard** issues an `advance()`. Method `advance()` opens the possibility for invoking random elements from the game environment (e. g. adding a new tile in 2048), if necessary.

When all this has happened, **GameBoard** sets its internal state such that `isActionReq()` returns `true` again. Thus it asks **Arena** for the next action and the cycle continues. Finally, **Arena** detects an `isGameOver()`-condition and finishes the game play.

### 3.9 Abstract Class Evaluator

Class **Evaluator** evaluates the performance of a **PlayAgent**. Evaluators are called in menu item 'Quick Evaluation', during training and at the end of each competition in menu item 'Multi-Competition'. It is important to note that Evaluator calls have no influence on the training process, they just measure the (intermediate or final) strength of a **PlayAgent**.

In the constructor

---

<sup>8</sup> see method `HGameMove(x,y)` in `GameBoardTTT` for an example.

```
public Evaluator(PlayAgent e_PlayAgent, int stopEval,
                int mode, int verbose);
```

the argument `mode` allows derived classes to create different types of evaluators. These may test different abilities of **PlayAgent**.<sup>9</sup>

A normal evaluation is started by calling **Evaluator**'s method `eval` which calls in turn the abstract method

```
abstract protected boolean eval_Agent();
```

and counts the consecutive successful returns from that method. The argument `stopEval` sets the number of consecutive evaluations that the abstract method `eval_Agent()` has to return with `true` until the evaluator is said to reach its *goal* (method `goalReached()` returns `true`). This is used in **XArenaFunc**'s method `train()` as a possible condition to stop training prematurely.

Method `eval_Agent()` needs to be overridden by classes derived from **Evaluator**. It returns `true` or `false` depending on a user-defined success criterion. In addition, it lets method `double getLastResult()` return a double characterizing the evaluation result (e.g. the average success rate of games played against Minimax player).

Concrete objects of class **Evaluator** are usually constructed by the factory method

```
abstract public Evaluator makeEvaluator(PlayAgent e_PlayAgent,
                                       int stopEval, int mode, int verbose);
```

in **Arena** or **ArenaTrain**.

Implementing classes: **EvaluatorTTT**, **Evaluator2048**, ...

### 3.10 Abstract Class Arena

Class **Arena** is an abstract class for loading agents and playing games. Why is it an abstract class? – **Arena** has to create an object implementing interface **GameBoard**, and this object will be game-specific, e.g. a **GameBoardTTT** object. To create such an object within the general **Arena**-code, the *factory method pattern* is used: **Arena** defines the abstract methods

```
abstract public GameBoard makeGameBoard();
abstract public Evaluator makeEvaluator(...);
```

The first method is a factory method for **GameBoard** objects. The second method is a factory method for **Evaluator** objects. Both will be implemented by classes derived from **Arena**. That is, a derived class **ArenaTTT** can be very thin, it just implements the methods `makeGameBoard()` and `makeEvaluator()` and lets them return (in the example of TicTacToe) **GameBoardTTT** and **EvaluatorTTT** objects, resp.

Class **Arena** has in addition the factory method

---

<sup>9</sup>For complex games it is often very difficult or impossible to have a perfect evaluator. Remember that (a) that the game tree can be too complex to retrieve the perfect action for a certain state and that (b) a perfect Evaluator should evaluate the actions of **PlayAgent** to *every* possible state, which would take too long (or is impossible) for games with larger state space complexity. A partial way out is to have different Evaluator modes which evaluate the agent from different perspectives.

```
public Feature makeFeatureClass(int);
```

If it is not overridden by derived classes, it will throw a `RuntimeException` (no game-tailored **Feature** object available). If a class derived from **Arena** wants to use a trainable agent requiring **Feature** (e. g. `TDAgent`) then it has to override `makeFeatureClass`.

Class **Arena** has similarly the factory method

```
public XNTupleFuncs makeXNTupleFuncs();
```

which can be used to generate a game-tailored **XNTupleFuncs** object, if needed (if agent **TDNTuple3Agt** is used). If not overridden, it will throw a `RuntimeException`.

Class **Arena** has the following functionality:

- choice of agents for each player (load or set),
- specifying parameters agents (except parameters for training),
- playing games (AI agents & humans),
- evaluating agents, competitions (one or multiple times),
- inspecting the move choices of an agent,
- logging of played games (option for later replay or analysis),
- a slider during agent-agent game play to control the playing velocity,
- tournaments (round-robin, ..., Elo, Glicko, ...).
- See Sec. 5 **Open Issues** for planned extensions.

Derived abstract class: **ArenaTrain**.

Derived non-abstract classes: `ArenaTTT`, `Arena2048`, ...

### 3.11 Abstract Class **ArenaTrain**

Class **ArenaTrain** is an abstract class derived from **Arena** which has this *additional* functionality:

- specifying all parameters for agents (including parameters for training),
- training agents (one or multiple times),
- saving agents.
- See Sec. 5 **Open Issues** for planned extensions.

Derived non-abstract classes: `ArenaTrainTTT`, `ArenaTrain2048`, ...



### 3.12 The Param Classes

Each agent or group of agents has associated classes for setting its parameters. These classes come in two flavours:

**...Params** e.g. **TDParams** holds the parameters **and** the GUI for setting them for all TD-related parameters. These classes are usually derived from `Frame` and as such their objects tend to be rather big.

**Par...** e.g. **ParTD** holds solely the parameters of TD-agents. Thus the objects of class `Par...` are much smaller and can be easily copied, attached to other objects and passed to other methods.

## 4 Use Cases and FAQs

### 4.1 I have implemented game XYZ and want to use AI agents from GBG – what do I have to do?

As a game developer you have to implement the following four interfaces for your game:

- `StateObserverXYZ` implements `StateObservation`
- `GameBoardXYZ` implements `GameBoard`
- `EvaluatorXYZ` extends `Evaluator`
- `FeatureXYZ` extends `Feature` (only needed, if you want to use the trainable agent **TDAgent**, see Sec. 4.6).
- `XNTupleFuncsXYZ` extends `XNTupleFuncs` (only needed, if you want to use the trainable n-tuple agent **TDNTuple3Agt**, see Sec. 4.7).

Once this is done, you only need to write a very 'thin' class `ArenaTrainXYZ` with suitable constructors, which overwrites the abstract methods of class `ArenaTrain` with the factory pattern methods

```
public GameBoard makeGameBoard() {
    gb = new GameBoardXYZ(this);
    return gb;
}
public Evaluator makeEvaluator(PlayAgent pa, GameBoard gb,
                              int stopEval, int mode, int verbose) {
    return new EvaluatorXYZ(pa,gb,stopEval,mode,verbose);
}
```

If needed, you should overwrite the methods (see Sec. 4.6 and Sec. 4.7)

```

public Feature makeFeaturClass(int featmode) {
    return new FeatureXYZ(featmode);
}
public XNTupleFuncs makeXNTupleFuncs() {
    return new XNTupleFuncsXYZ();
}

```

as well.

If you do not want to use the agents **TDAgent** and **TDNTuple3Agt** needing these factory methods, you may just implement stubs throwing suitable exceptions:

```

public Feature makeFeaturClass(int featmode) {
    throw new RuntimeException("Feature not implemented for XYZ");
}
public XNTupleFuncs makeXNTupleFuncs() {
    throw new RuntimeException("XNTupleFuncs not implemented for XYZ");
}

```

Finally you need a `main()` to launch **ArenaTrain**. You may copy and adapt the `main()` example in `ArenaTrainTTT`. (And a similar `main()` and similar factory pattern methods in `ArenaTTT`.)

Then you can use for your game all the functionality laid down in **ArenaTrain** and all the wisdom of the AI agents implementing **PlayAgent**. Cool, isn't it?

## 4.2 How to train an agent and save it

1. Create an **ArenaTrain** object
2. Select an agent and set its parameters
3. Set training-specific parameters:
  - `maxTrainNum`: 'Training games' = number of training episodes,
  - `numEval`: after how many episodes an intermediate evaluation is done,
  - `epiLength`: 'Episode length' = maximum allowed number of moves in a training episode. If it is reached, the game is stopped and `PlayAgent.estimateGameValue()` is returned (either up-to-now-reward or estimate of current + future rewards). If the game terminates earlier, the final game score is returned.
4. Train the agent & visualize intermediate evaluations.
5. Optional: Inspect the agent (how it responds to certain board situations).
6. Save the agent

## 4.3 Which AI's are currently implemented for GBG?

See Sec. 3.2.1 for a list of all AI's (agents), i. e. classes that implement interface **PlayAgent**.

## 4.4 How to write a new agent (for all games)?

Of course your new agent `NewAgent` has to implement the interface `PlayAgent`. You may want to derive your new agent from `AgentBase` to have a few basic functions already with their default implementations. These functions can be overridden if necessary.

The new agent should as well implement the interface `Serializable` (`java.io`) to be loadable and savable.

There are a few places in the code where the new agent has to be registered:

- `Types.GUI_AGENT_LIST`: Add a suitable agent nickname "nick". This is how the agent will appear in the agent choice boxes.
- `XArenaFuncs.constructAgent()`: Add a suitable clause  
`if (sAgent.equals("nick")) ...`
- `XArenaFuncs.fetchtAgent()`: Add a suitable clause  
`if (sAgent.equals("nick")) ...`
- `XArenaTabs.showParamTabs()`: Add a suitable clause  
`if (selectedAgent.equals("nick")) ...`
- `XArenaMenu.loadAgent()`: Add a suitable clause  
`if (td instanceof NewAgent) ...`
- `LoadSaveGBG.loadGBGAgent()`: Add a suitable clause  
`if (obj instanceof NewAgent) ...`

If the agent has new sensible default parameters, they may be added to function `setParamDefaults` in classes `TDParams`, `NTPParams` or other.

If the agent requires a whole set of new parameters which do not fit into the existing `Param` classes, then new `Param` classes `...Param` and `Par...` have to be constructed and added to the `Param` tab.

## 4.5 What is the difference between `TDAgent` and `TDNTuple3Agt`?

Both agents are trained by TD (temporal difference learning). They differ only in their feature vectors: While for `TDAgent` the user has to specify each feature (see Sec. 3.5 through method

```
public double[] prepareFeatVector(StateObservation so),
```

the class `TDNTuple3Agt` constructs the features automatically from the given n-tuple sets and **position values** (see Appendix C.1).

## 4.6 How to specialize the **TDAgent** to a new game?

Suppose you have implemented a new game XYZ and want to write a TD agent (temporal difference agent) which learns this game. What do you have to do? – Luckily, you can re-use most of the functionality laid down in class **TDAgent** (see Sec. 3.2).

1. Write a new **Feature** class

```
public class FeatureXYZ implements Feature, Serializable
```

This is the only point where some code needs to be written: Think about what features are useful for your game. In the simplest case this might be the raw board positions, but these features may characterize the win- or loose-probability for a state only rather indirectly. Other patterns may characterize the value (or the danger) of a state more directly. For example, in the game TicTacToe any two-in-a-line opponent pieces accompanied by a third empty position pose an immanent threat. A typical feature may be the count of those threats. N-tuple sets (Lucas [2008], Thill et al. [2014], Bagheri et al. [2015], Thill [2015]) are another way of generating a large number of features in a generic way (but they are not part of **TDAgent**, see Sec. 4.7, Appendix C and **TDNTuple3Agt** instead).

2. Add to ArenaXYZ and ArenaTrainXYZ the overriding method

```
public Feature makeFeatureClass(int featmode) {  
    return new FeatureXYZ(featmode);  
}
```

**TDAgent** will generate by reinforcement learning a mapping from feature vectors to game values (estimates of the final score, see Sec. 3.4) for all relevant game states.

The class ArenaTrainTTT (together with FeatureTTT) may be inspected to view a specific example for the game TicTacToe.

## 4.7 How to specialize the **TDNTuple3Agt** agent to a new game?

Suppose you have implemented a new game XYZ and want to write a TD (temporal difference) agent using n-tuples which learns this game. What do you have to do? – Luckily, you can re-use most of the functionality laid down in class **TDNTuple3Agt** (see Sec. 3.2). As a game implementer you have to do the following:

1. Write a new **XNTupleFuncs** class (Sec. 3.6)

```
public class XNTupleFuncsXYZ  
    implements XNTupleFuncs, Serializable
```

Here you have to code some rather simple things like `getNumCells()`, the number of board cells in your game, and `getNumPositionValues()`, the number of **position values** that can appear in each cell. This is for example 9 and 3 (O/empty/X) in the game TicTacToe.

Next you implement

```
int[] getBoardVector(so)
```

which transforms a game state `so` into an `int[]` board vector (length: `getNumCells()`). See Appendix **C.1** for board cell numbering and a specific example.

If your game has **symmetries** (the game TicTacToe has for example eight symmetries, 4 rotations  $\times$  2 mirror reflections), the function

```
int[][] symmetryVectors(int[] boardVector)
```

should return for a given board vector all symmetric board vectors (including itself). If the game has no symmetries, it returns just the board vector itself.

The method

```
HashSet adjacencySet(int iCell)
```

returns the set of cells adjacent to the cell with number `iCell`. Whether adjacency is a 4-point- or an 8-point-neighborhood or something else is defined by the user. This function is needed by **TDNTuple3Agt** if the shape of the n-tuples is to be created by random walk.

Finally you implement

```
int[][] fixedNTuples()
```

a function returning a fixed set of n-tuples suitable for your game. If you do not need fixed n-tuple sets, you may leave `fixedNTuples()` unimplemented (i. e. let it throw an exception) and chose in the `NTPar` (n-tuple params) tab 'Random n-tuple generation'.

2. Add to `ArenaXYZ` and `ArenaTrainXYZ` the overriding method

```
public XNTupleFuncs makeXNTupleFuncs() {  
    return new XNTupleFuncsXYZ();  
}
```

The class `ArenaTrainTTT` (together with `XNTupleFuncsTTT`) may be used as a template, showing the implementation for the game TicTacToe.

**TDNTuple3Agt** offers several possibilities to construct n-tuples:

- (a) using a predefined, game-specific set of n-tuples (see `fixedNTuples()` above),
- (b) random n-tuples generated by random-cell-picking (the cells in an n-tuple are in general not adjacent), and
- (c) random n-tuples generated by random walk (every cell in each n-tuple is adjacent to at least one other cell of this n-tuple; needs method `adjacencySet`, see above).

A cell may (and often should) be part of several n-tuples.

The same remarks apply if you want to specialize **SarsaAgt** or **TDNTuple2Agt** (now deprecated) to a new game.

See Appendix C for further information on n-tuples.

## 4.8 What is a ScoreTuple?

A **ScoreTuple** has a vector

```
public double[] scTup
```

of size  $N$  containing the **game score** or **game value** – depending on context – for each player  $0, 1, \dots, N - 1$ .

It has methods to combine the current **ScoreTuple** `this` with a second **ScoreTuple** `tuple2nd` according to one of the following operators:

**AVG** weighted average or expectation value: add `tuple2nd`, weighted with a certain probability weight. The probability weights of all combined tuples should sum up to 1.

**MIN** combine by retaining this **ScoreTuple**, which has in `scTup[playNum]` the lower value.

**MAX** combine by retaining this **ScoreTuple**, which has in `scTup[playNum]` the higher value.

**DIFF** subtract from `this` all values in the other `tuple2nd`.

## 5 Open Issues

The current GBG class framework is still in its test phase. The design of the classes and interfaces may need further reshaping when more games or agents are added to the framework. There are a number of items not fully tested or not yet addressed:

- Further elements for **Arena** and **ArenaTrain** are planned but not yet implemented:
  - Add **Arena** and **ArenaTrain** launchers which allow to select between the different implemented games and then launch the appropriate derived **Arena** and **ArenaTrain** class.
  - undo/redo possibilities
  - game balancing

- time measurements for agents (play & train)
  - client-server architecture for game play via applet on a game page. Option for a 'hall of fame'. An example for the game Hexi is available from TU Wien <sup>10</sup>.
  - Implement the game Sim (= Hexi) in the **Arena** and **ArenaTrain** framework. A Java code example of the Sim board GUI is available from TU Wien <sup>11</sup>. Generalize the number of nodes (not only 6). Later, one may create a 3-player variant of Sim and test the framework on this.
- Replay memory for better training: This is the idea used by DeepMind in learning Atari video games. Played episodes are stored in a replay memory pool and used repeatedly for training.
  - The extension to  $N$ -player games ( $N > 2$ ) is fully functional but not yet tested. An example to fully test  $N$ -player games may be the 3-player variant of the game Sim. The cases  $N = 1$  and  $N = 2$  need also to be tested on a variety of 1- and 2-player games.

## A Appendix: Interface Summary

### A.1 Interfaces StateObservation and StateObsNondeterministic

```
/**
 * Class StateObservation observes the current state of the game,
 * it has utility functions for
 * <ul>
 * <li> returning the available actions (getAvailableActions()),
 * <li> advancing the state of the game with a specific action (advance()),
 * <li> copying the current state
 * <li> signaling end, score and winner of the game
 * </ul>
 *
 * @author Wolfgang Konen, TH Köln
 */
public interface StateObservation {

    public StateObservation copy();
    public String getName();
    public String stringDescr();
    public double getGameScore();
    public double getGameScore(int player);
```

<sup>10</sup><http://www.dbai.tuwien.ac.at/proj/ramsey>

<sup>11</sup><http://www.dbai.tuwien.ac.at/proj/ramsey>

```

public double getGameScore(StateObservation referringState);
public ScoreTuple getGameScoreTuple();
public double getReward(boolean rgs);
public double getReward(StateObservation referringState, boolean rgs);
public ScoreTuple getRewardTuple(boolean rgs);
public double getMinGameScore();
public double getMaxGameScore();
public int getMinEpisodeLength();
public void advance(ACTIONS action);
public void advanceDeterministic(ACTIONS action);
public void advanceNondeterministic();
public StateObservation getPrecedingAfterstate();
public boolean isDeterministicGame();
public boolean isLegalState();
public boolean isGameOver();
public Types.WINNER getGameWinner();
public ArrayList<ACTIONS> getAvailableActions();
public void setAvailableActions();
public int getNumAvailableActions();
public ACTIONS getAction(int i);
public void storeBestActionInfo(ACTIONS actBest, double[] vtable);
public int getNumPlayers(); // n
public int getPlayer();      // (0,1,...,n-1)
}

public interface StateObsNondeterministic extends StateObservation {
    public void advanceDeterministic(ACTIONS action);
    public void advanceNondeterministic();
    public void advanceNondeterministic(ACTIONS action);
    public boolean isNextActionDeterministic();
    public ACTIONS getNextNondeterministicAction();
    public ArrayList<ACTIONS> getAvailableRandoms();
    public int getNumAvailableRandoms();
    public double getProbability(ACTIONS action); // probability of this action
    public StateObsNondeterministic copy();
}

```

## A.2 Interface GameBoard

```

/**
 * Each class implementing interface GameBoard has the board game GUI.
 *
 * It has an internal object derived from StateObservation which

```



```

    * represents the current game state. This game state can be retrieved
    * (getStateObs()), reset-retrieved (getDefaultStartState()) or a random
    * start state can be chosen (@link #chooseStartState01()).
    * @author Wolfgang Konen, TH Köln
    */
public interface GameBoard {

    public void clearBoard(boolean boardClear, boolean vClear);
    public void updateBoard(StateObservation so, boolean enableOccupiedCells
        , boolean showValueOnGameboard);
    public void showGameBoard(Arena arena,boolean alignToMain);
    public void toFront();
    public boolean isActionReq(); // action requested from Arena?
    public void setActionReq(boolean actionReq);
    public void enableInteraction(boolean enable);
    public StateObservation getStateObs();
    public StateObservation getDefaultStartState(); // empty-board state
    public StateObservation chooseStartState();
    public StateObservation chooseStartState(PlayAgent pa);
    public String getSubDir();
}

```

### A.3 Interface PlayAgent

```

/**
 * The abstract interface for the game playing agents.
 *
 * @author Wolfgang Konen, TH Köln
 */
public interface PlayAgent {
    public enum AgentState {RAW, INIT, TRAINED};
    public ACTIONS_VT getNextAction2(StateObservation sob,
        boolean random, boolean silent);
    public double getScore(StateObservation sob);
    public ScoreTuple getScoreTuple(StateObservation sob);
    public double estimateGameValue(StateObservation sob);
    public Types.ScoreTuple estimateGameValueTuple(StateObservation sob);
    public boolean trainAgent(StateObservation so); // for one episode
    public String printTrainStatus();
    public String stringDescr();
    public String stringDescr2();
    public String getName();
}

```

```

public void setName(String name);
public AgentState getAgentState();
public void setAgentState(AgentState aState);
public int getMaxGameNum();
public void setMaxGameNum(int num);
public int getGameNum();
public void setGameNum(int num);
public int getNumEval();
public void setNumEval(int num);
public long getNumLrnActions();
public long getNumTrnMoves();
public ParOther getParOther();
}

```

#### A.4 Classes ACTIONS, ACTIONS\_VT and ACTIONS\_ST

```

public class ACTIONS implements Serializable, Comparable<ACTIONS> {
    public ACTIONS(int numVal, boolean random);
    public int toInt();
    public static ACTIONS fromInt(int iAct);
    public int hashCode();
    public boolean equals(Object object);
    public int compareTo(ACTIONS action);
    public boolean isRandomAction();
    public void setRandomSelect(boolean randomSelect);
}

public class ACTIONS_VT extends ACTIONS { // ACTIONS + VTable
    public ACTIONS_VT(int numVal, boolean random, double [] vtable, double vbest);
    public double[] getVTable();
    public double getVBest();
}

public class ACTIONS_ST extends ACTIONS { // ACTIONS + ScoreTuple
    public ScoreTuple m_st;
    public ACTIONS_ST(Types.ACTIONS oa, ScoreTuple st);
}

```

#### A.5 Interface Feature

```

/**
 * Interface Feature translates game states into feature vectors.

```

```

*
* Method prepareFeatVector(so) returns the feature vector
* for state so. Child classes have usually constructors accepting a
* single argument 'featmode'. The argument 'featmode' allows to
* construct different flavors of Feature objects.
* The acceptable values for 'featmode' in a certain child class
* are retrieved with getAvailFeatmode().
*/
public interface Feature {
public double[] prepareFeatVector(StateObservation so);
public String stringRepr(double[] featVec);
public int getFeatmode();
public int[] getAvailFeatmode();
public int getInputSize(int featmode);
}

```

## A.6 Interface XNTupleFuncs

```

/**
* Interface XNTupleFuncs contains game-specific functions
* for using n-tuple sets.
*/
public interface XNTupleFuncs {
public int getNumCells();
public int getNumPositionValues();
public int getNumPlayers();
public int[] getBoardVector(StateObservation so);
public int[][] symmetryVectors(int[] boardVector);
public int[][] fixedNTuples(int mode);
public int[] getAvailFixedNTupleModes();
public HashSet adjacencySet(int iCell);
}

```

## A.7 Abstract class Evaluator

```

/**
* Evaluates the performance of a PlayAgent in a game.
*/
abstract public class Evaluator {
protected PlayAgent m_PlayAgent;
}

```

```

protected int verbose=1;
public Evaluator(PlayAgent e_PlayAgent, int stopEval);
public Evaluator(PlayAgent e_PlayAgent, int stopEval, int verbose);
public boolean eval();
abstract protected boolean eval_Agent();
abstract public double getLastResult();
public boolean goalReached(int gameNum);
public boolean setState(boolean stateE);
public boolean getState();
public String getMsg();
public String getGoalMsg(int gameNum);
abstract public boolean isAvailableMode(int mode);
abstract public int[] getAvailableModes();
public static int getDefaultEvalMode()
abstract public int getQuickEvalMode();
abstract public int getTrainEvalMode();
abstract public int getMultiTrainEvalMode();
abstract public String getPrintString();
abstract public String getPlotTitle();
}

```

## A.8 Abstract class Arena

```

/**
 * This class contains the GUI and the task dispatcher for the game. The GUI for
 * buttons and choice boxes is in {@link XArenaButtons}.
 * <p>
 * Run this class for example from {@code main} in {@link ArenaTTT} or
 * {@link ArenaTrainTTT} for the TicTacToe game.
 *
 * @author Wolfgang Konen, TH Köln
 */
abstract public class Arena extends JPanel implements Runnable
{
    public enum Task {PARAM, TRAIN, MULTRN, PLAY, INSPECTV
                     , COMPETE, SWAPCMP, MULTCMP, IDLE };
    protected GameBoard gb;
    public Task taskState;
    public Arena();
    public Arena(JFrame);
    public void init();
    public void run();
    public void PlayGame();
}

```

```

protected void InspectGame(); // inspect agent X on game positions
public void enableButtons(boolean state);
public void setStatusMessage(String msg);
public StatusBar getStatusBar();
public boolean hasTrainRights(); // returns false
public Feature makeFeatureClass(int featmode);
public XNTupleFuncs makeXNTupleFuncs();
abstract public String getGameName();
abstract public GameBoard makeGameBoard();
abstract public Evaluator makeEvaluator(PlayAgent pa, GameBoard gb,
                                         int stopEval, int mode, int verbose);
abstract public void performArenaDerivedTasks();
}

```

## A.9 Abstract class ArenaTrain

```

/**
 * This class contains the GUI for the arena with train capabilities.
 * It extends the task dispatcher of Arena with method
 * performArenaDerivedTasks() which contains tasks to trigger functions
 * for agent learning, parameterization, inspection and so on.
 *
 * @author Wolfgang Konen, TH Köln
 */
abstract public class ArenaTrain extends Arena
{
    public ArenaTrain();
    public ArenaTrain(JFrame frame);
    public void performArenaDerivedTasks(); // extend task dispatcher
    public boolean hasTrainRights(); // returns true
}

```

## B Appendix: Other Game Value Functions

Sec. 3.3 and 3.4 have introduced with

```

StateObservation.getGameScore(refer)
PlayAgent.getScore(StateObservation sob)

```

the main functions to retrieve a **game score** or **game value**, resp. There are two other functions delivering a game value; they are only required for more advanced needs:

- Interface **StateObservation** delivers with `getReward(StateObservation refer, boolean rgs)` a function returning the **game reward**. This reward can be simply the game score in case `rgs==true` ('reward is game score'), but it can be also another (game-specific) function in case `rgs==false`. This opens the possibility that the reward might be something different from game score. Example: In the game 2048, a possible reward for a state can be the number of empty tiles in that state.
- Interface **PlayAgent** delivers with `estimateGameValue(so)` a function (perhaps trainable / adjustable from previous experiences) that estimates the future game value at end of play. The difference to `PlayAgent.getScore(StateObservation so): estimateGameValue(so)` may NOT call `getScore(so)` or `getNextAction(so)`, since these functions may call `estimateGameValue(so)` inside (e.g. if a certain episode length is reached) and this would result in infinite recursion. A simple implementation can be to return just `so.getReward(rgs)`, but other implementations are possible as well.

A potential use of `pa.estimateGameValue(sob)` is to compute in MC or MCTS the final value of a random rollout in cases where the rollout did not reach a terminal game state (since the episode lasts longer than the 'Rollout depth' as it is for example in 2048 often the case).

A second use of `pa.estimateGameValue(sob)` is in trainable agents, when the maximum training episode length (if any) is reached.

A third use of `pa.estimateGameValue(sob)` is in `MaxNAgent` when the tree depth is reached but the game is not yet over. Then we call `pa.estimateGameValueTuple(sob)`.

A fourth use of `pa.estimateGameValue(sob)` is in wrapper `MaxNWrapper` and `ExpectimaxWrapper`. These wrappers implement `estimateGameValue(sob)` and let it return the wrapped agent's game value via

```
wrappedAgt.getScore(sob).
```

It is dependent on the class implementing **PlayAgent** what

```
estimateGameValue(sob)
```

actually returns. If it is too complicated to train a value function (or if it is simply not needed, because for a game like TicTacToe we come always to an end during rollout), then `estimateGameValue(sob)` may simply return `sob.getReward(rgs)`.

If we integrate a trainable game value estimation into a class implementing **PlayAgent**, then agents that formerly did not need training (Minimax, MC, MCTS, ...) will require training. They should be after construction in **AgentState** INIT. How the training is actually done depends fully on the implementing agent.

Table 1 gives an overview over all functions in GBG returning a game score or a game value. Summary of main facts:

- `getReward(...,rgs)` returns `getGameScore(...)`, if `rgs==true`. Otherwise it returns a specific reward, depending on the nature of the game (whatever the class derived from **StateObservation** implements).

Table 1: Summary of all game score and game value functions in GBG.

method	remark
class StateObservation	
getGameScore()	<i>(deprecated)</i> game score of this
getGameScore(refer)	game score of this relative to referring state refer
getGameScore(int i)	... relative to player i
getGameScoreTuple()	a ScoreTuple with game scores for all players
getReward(rgs)	<i>(deprecated)</i> the game reward of this
getReward(refer,rgs)	game reward of this relative to referring state refer
getReward(int i,rgs)	... relative to player i
getRewardTuple(rgs)	a ScoreTuple with rewards for all players
class PlayAgent (pa)	
getScore(so)	the game value = agent's estimate of so's final score
getScoreTuple(so)	a ScoreTuple with game values for all players
estimateGameValue(so)	agent's estimate of game value for so
estimateGameValueTuple(so)	a ScoreTuple with agent estimates for all players

- `getReward()` and `getGameScore()` return the *cumulative* reward and game score for the StateObservation object `this`. If the user wants the delta reward, he/she has to subtract the reward of the preceding state.
- `getReward(...)` is used in all places where agents reason about the next action. This is in move calculation (`getNextAction2`), in `estimateGameValue...` and during training.
- If a game has its `StateObservation` class derived from `ObserverBase` and it does not implement `getReward(...)`, then default implementations from `ObserverBase` are taken which implement the reward just as game score and issue a warning if they are called with `rgs==true`.
- `pa.getScore` returns the agent's estimate of the **game value**, i.e. the estimate of the final score attainable for `pa` – assuming perfect play. To calculate this, it may ask a model (e.g. NN or some weighted piece-position formula), or it may perform recursive search up to a tree depth / rollout depth, depending on the nature of the agent. If the maximum depth is reached, it may call `estimateGameValue`.
- `pa.estimateGameValue` returns also an game value estimate. But it returns a coarser estimate, since it may not call `getScore` back (to avoid an infinite loop).
- All **game score** and **game value** methods have associated `...Tuple` versions: They return instead of a single game score or game value a **ScoreTuple** (Sec. 4.8) of  $N$  values for all players  $0, 1, \dots, N - 1$ .

## C Appendix: N-Tuples

### C.1 Board Cell Numbering

Each n-tuple is a list of board cells [Lucas \[2008\]](#). Board cells are specified by numbers. The canonical numbering for a rectangular board is row-by-row, from left to right. For example, a  $4 \times 4$  board would carry the numbers

```
00 01 02 03
04 05 06 07
08 09 10 11
12 13 14 15
```

Other (irregular) boards may carry other (user-specified) cell numbers. Each choice of numbering is o.k., it has only to be used consistently throughout the game.

Given the board cell numbering, the method

```
int[] XNTupleFuncs::getBoardVector(StateObservation so)
```

returns a **board vector** (a vector whose length is the number of board cells

```
int XNTupleFuncs::getNumCells()
```

), carrying the position value for each board cell according to this numbering. The **position value** of a cell is a game-specific coding of all states a board cell can be in. It is a number in  $\{0, 1, 2, \dots, P - 1\}$  with  $P = \text{XNTupleFuncs::getNumPositionValues}()$ .

*Example: The canonical board cell numbering for the game TicTacToe run from 00 to 08. The position values are 0: O, 1: empty, 2: X. Each board vector has length 9. For state **sA** in Fig. 1 the board vector is*

```
bVec = {1, 1, 0, 1, 2, 2, 1, 1, 1};
```

### C.2 N-Tuple Creation

N-tuple sets can be created through explicit specification by the user (n-tuple fixed mode) or through a random initialization process.

Fig. 2 shows two examples of  $4 \times 4$  boards with fixed (user-specified) n-tuple sets. The canonical cell number is obtained from

$$4 \times \text{row\_number} + \text{col\_number}$$

*Example (a) in Fig. 2 is coded in a class derived from **XNTupleFuncs** as*

```
public int[][] fixedNTuples() {
    int nTuple[][] = { {0,1,2,3}, {1,5,6,2}, {0,4,8,9,10,11} };
    return nTuple;
}
```



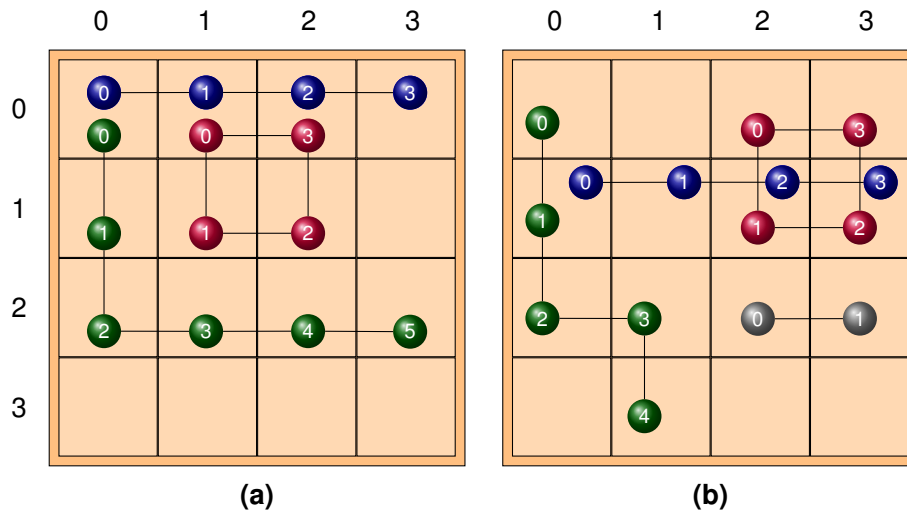


Figure 2: Two examples for n-tuples: (a) 3 n-tuples, (b) 4 n-tuples of varying length and placement.

*Example (b) is coded as*

```
public int[] [] fixedNTuples() {
    int nTuple[] []={ {4,5,6,7}, {2,6,7,3}, {0,4,8,9,13}, {10,11} };
    return nTuple;
}
```

Each n-tuple contains each cell at most once. But a set of n-tuples may (and often should) contain the same cell multiple times.

The following **random** initialization processes to create n-tuple sets are provided by the n-tuple factory:

1. **Random points:** Cells are picked at random, no cell twice<sup>12</sup>, no topographical connection. This is often not advantageous because in many board games the neighborhood of a cell is more important for determining its value than an arbitrary other more distant cell.
2. **Random walks:** Cells are picked at random, no cell twice, with adjacency constraint. That is, each cell of the n-tuple list must be *adjacent* to at least one other cell in the n-tuple. What *adjacent* actually means in a certain game is specified by the user through the **XNTupleFuncs** method

```
public HashSet adjacencySet(int iCell);
```

which returns the set of all neighbors of cell iCell.

<sup>12</sup>within the same n-tuple

### C.3 N-Tuple Training and Prediction

How are the n-tuples used to generate features? – Each n-tuple has an associated look-up table (LUT) of length  $P^n$  where  $n$  is the n-tuple length and  $P$  is the number of **position values** each cell might have.

*Example: TicTacToe has  $P = 3$  cell position values:  $\{0, -, X\}$ . For an n-tuple of length  $n = 2$  this leads to  $3^2 = 9$  possible LUT entries*

$\{00, 0-, 0X, -0, --, -X, X0, X-, XX\}$

These LUT-entries are features. Even for a small number of n-tuples this will generate quite a large number of features. For example in Fig. 2(a), if we assume 3 **position values** for each cell, the number of features is  $2 \cdot 3^4 + 3^6 = 891$ , because there are 2 4-tuples and one 6-tuple. On a larger board, a more realistic setting would be, for example, 40 n-tuples of length 8, resulting in  $40 \cdot 3^8 = 262\,440$  features.

Each feature  $i$  in n-tuple  $\nu$  has an associated weight  $w_{\nu,i}$ . Given a certain board state, we look first which of those features are active ( $x_{\nu,i} = 1$ ) or inactive ( $x_{\nu,i} = 0$ ) in that board state. Then the n-tuple network computes its estimate  $V^{(est)}$  of the game value through

$$V^{(est)} = \sigma \left( \sum_{\nu=1}^m \sum_{i=0}^{P^n-1} w_{\nu,i} x_{\nu,i} \right) \quad (1)$$

which is simply a neural net without hidden layer and with a sigmoid function  $\sigma(\cdot)$ .<sup>13</sup> We compare the estimate generated by this net with the target game value  $V$  prescribed by TD-learning. A  $\delta$ -rule learning step with step-size  $\alpha$  (gradient descent) is made for each weight in order to decrease the perceived difference  $\delta = V - V^{(est)}$  between both game values (Thill et al. [2014], Thill [2015]).

For complex games it might be necessary to train such a network for several hundred thousand or even million games in order to reach a good performance. The so-called **eligibility traces** are a general technique from TD-learning to speed up learning. They can be activated in the GBG framework by setting parameter  $\lambda > 0$  in the *TD pars* parameter tab. Further details on eligibility traces are found in Thill et al. [2014].

Once the network is trained, the game value estimate  $V^{(est)}$  is used to decide about the next action.

To further speed up learning, symmetries may be used: **Symmetries** are transformations of the board state which lead to board states with the same game value. If weights for symmetric states are trained simultaneously, this will lead to better generalization of the trained agent. For example, Tic-tac-toe and 2048 have eight symmetries (4 rotations  $\times$  2 mirror reflections). Instead of performing only *one* learning step with the board state itself, one can do *eight* learning steps by looping through all symmetric states. This may greatly speed up learning, since more weights can learn on each move and the network generalizes better.

If the game has symmetries, the user has to code them in **XNTupleFuncs** method

<sup>13</sup>In TDNTuple3Agt the sigmoid function is always  $\sigma = \tanh$  (see helper class NTupleValueFunc), so that  $V^{(est)} \in [-1, 1]$  holds.

Table 2: Summary of various multi-core threads in GBG.

method	remark
class MCAgent, MCAgentN	
getNextAction_PAR	parallelization over available actions
getNextAction_MassivePAR	parallelization over available actions AND over rollouts
class EvaluatorHex	
competeAgainstMCTS_diffStates_PAR	parallelization over different start states for eval mode 10 & agent TDNTuple3Agt
class Evaluator2048	
eval_Agent	two parallelizations over evaluation games for 2 agents MCTS-Expectimax & ExpectimaxWrapper

```
public int[] [] symmetryVectors(int[] boardVector);
```

See Sec. 4.7 for further information on this method.

## D Appendix: Multi-core threads

GBG supports for several time-consuming operations multi-core (parallel) threads to speed up calculation. The operations are given in Table 2.

Note that an operation can only be parallelized, if the relevant routines and agents are thread-safe. This is for example the case for agent TDNTuple3Agt when it is evaluated (where its method getNextAction2 is needed): This method does not change any data of this agent, so different threads can use the same TDNTuple3Agt object and call this method independently. This is what we do in the parallel thread of EvaluatorHex.

On the other hand, an agent like MCTS is not thread-safe, because each call to getNextAction2(sob,...) with a different state sob would construct different MCTS tree data. The only way to parallelize game play with MCTS is that each thread has its own copy of an MCTS with the parameters given. This is exactly what we do in the two parallel threads in Evaluator2048.

## E Appendix: String representations of agents

Table 3 shows and distinguishes different methods for agent string representations.

If a class derived from **AgentBase** does not specify stringDescr and stringDescr2, then the default implementation from **AgentBase** is taken, which is only the simple and the full class name, resp.

Table 3: Summary of various agent string representation methods in GBG.

method	remark
getName	the name given to the constructor of the agent, e.g. TD-Ntuple-3
getClass.getSimpleName	the simple name of the underlying class, e.g. TDNTuple3Agt
getClass.getName	the full class name, e.g. controllers.TD.ntuple2.TDNTuple3Agt
stringDescr	simple class name + parameter settings
stringDescr2	full class name + additional parameter settings

## References

- Samineh Bagheri, Markus Thill, Patrick Koch, and Wolfgang Konen. Online adaptable learning rates for the game Connect-4. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(1):33–42, 2015. 13, 20
- Nicolas A. Barriga, Marius Stanescu, and Michael Buro. Combining strategic learning and tactical search in real-time strategy games. *arXiv preprint arXiv:1709.03480*, 2017. 5
- Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016. 5
- Michael Genesereth and Michael Thielscher. General game playing. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(2):1–229, 2014. 5
- Michael Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the AAAI competition. *AI magazine*, 26(2):62, 2005. URL <https://www.aaai.org/ojs/index.php/aimagazine/article/view/1813>. 5
- Wolfgang Konen. Reinforcement learning for board games: The temporal difference algorithm. Technical report, Research Center CIOP (Computational Intelligence, Optimization and Data Mining), TH Köln – Cologne University of Applied Sciences, 2015. URL [http://www.gm.fh-koeln.de/ciopwebpub/Kone15c.d/TR-TDgame\\_EN.pdf](http://www.gm.fh-koeln.de/ciopwebpub/Kone15c.d/TR-TDgame_EN.pdf). 10
- Richard E. Korf. Multi-player alpha-beta pruning. *Artificial Intelligence*, 48(1):99–111, 1991. 9
- Johannes Kutsch. KI-Agenten für das Spiel 2048: Untersuchung von Lernalgorithmen für nichtdeterministische Spiele, 2017. URL <http://www.gm.fh-koeln.de/ciopwebpub/Kutsch17.d/Kutsch17.pdf>. Bachelor thesis, TH Köln – University of Applied Sciences. 9
- John Levine, Clare Bates Congdon, Marc Ebner, Graham Kendall, Simon M Lucas, Risto Miikkulainen, Tom Schaul, and Tommy Thompson. General video game playing. Technical report, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2013. 5

- Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification. Technical report, Stanford Logic Group Computer Science Department, Stanford University, 2008. 5
- Simon M. Lucas. Learning to play Othello with n-tuple systems. *Australian Journal of Intelligent Information Processing*, 4:1–20, 2008. 13, 20, 32
- Jacek Mańdziuk and Maciej Świechowski. Generic heuristic approach to general game playing. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 649–660. Springer, 2012. 5
- Daniel Michulke. Neural networks for high-resolution state evaluation in general game playing. In *Proceedings of the IJCAI-11 Workshop on General Game Playing (GIGA’11)*, pages 31–37. Citeseer, 2011. 5
- Daniel Michulke and Michael Thielscher. Neural networks for state evaluation in general game playing. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 95–110. Springer, 2009. 5
- Santiago Ontanón and Michael Buro. Adversarial hierarchical-task network planning for complex real-time games. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, pages 1652–1658. AAAI Press, 2015. 5
- Barney Pell. A strategic METAGAME player for general chess-like games. *Computational Intelligence*, 12(1):177–198, 1996. 5
- Maciej Świechowski, HyunSoo Park, Jacek Mańdziuk, and Kyung-Joong Kim. Recent advances in general game playing. *The Scientific World Journal*, 2015, 2015. 5
- Michael Thielscher. A general game description language for incomplete information games. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010. 5
- Markus Thill. Temporal difference learning methods with automatic step-size adaption for strategic board games: Connect-4 and Dots-and-Boxes. Master thesis, TH Köln – Cologne University of Applied Sciences, June 2015. URL <http://www.gm.fh-koeln.de/~konen/research/PaperPDF/MT-Thill2015-final.pdf>. 13, 20, 34
- Markus Thill, Samineh Bagheri, Patrick Koch, and Wolfgang Konen. Temporal difference learning with eligibility traces for the game Connect-4. In Mike Preuss and Günther Rudolph, editors, *CIG’2014, International Conference on Computational Intelligence in Games, Dortmund*, 2014. 13, 20, 34