e-print https://www.gm.fh-koeln.de/ciopwebpub/Konen22a.d/TR-GBG.pdf

The GBG Class Interface Tutorial V2.3: General Board Game Playing and Learning

Wolfgang Konen

Cologne Institute of Computer Science,
TH Köln,
Germany

wolfgang.konen@th-koeln.de

Last update: Sep 2022

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.

This document is an updated version of the 2020 GBG tutorial[1] (new or enhanced sections on *Parameter Settings for Scheiermann and Konen [2022]*, *Wrapper classes and Wrapper parameters*, *N-Tuples*, *GBG Help* and *Design Principles*, removed section on *ArenaTrain*).

GBG is written in Java and available from GitHub.[2]

[1] https://www.gm.fh-koeln.de/ciopwebpub/Kone20d.d/TR-GBG.pdf

[2] https://github.com/WolfgangKonen/GBG

Contents

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(λ), 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 TicTacToe, 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. 8 discusses some use cases and FAQs for the GBG class framework. 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.

See Konen [2019] for a more research-oriented description of GBG.

See GBG Wiki - Board Games for the actual list of games implemented in GBG.

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] have 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 Artificial Intelligence (AI), 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: „AI-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 AI methods.“ Michulke and Thielscher [2009] and Michulke [2011] present first results on translating GDL rules to neural networks and TD learning: They had some success, but they faced also 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 AI-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, μRTS [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, $N = 1, 2, 3, \ldots$, 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, Blackjack, ...) 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 and in [Konen, 2019]:

- 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 (for the n-tuple agents, see Sec. 3.8 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 AI agents, e.g. TD- and SARSA-agents and – for the first time – a ***generic*** implementation of TD-n-tuple-agents (see Sec. 3.5), 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.
  See Sec. 3.5.1 and Table 1 for a **list of all AI's (agents)**.

- 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.[3] 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.[4] 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.

[3] Examples are the near-perfect-playing `AlphaBetaAgent` for Connect-4 and `BoutonAgent` for Nim.

[4] 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 possible.

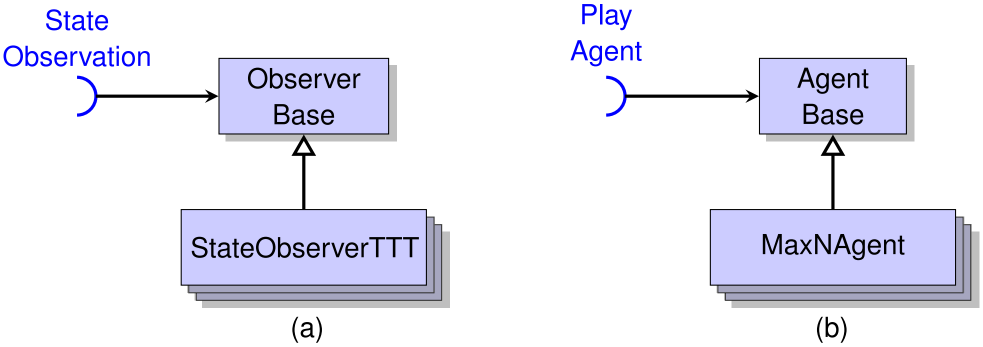

Figure 1: Important interfaces in GBG: (a) StateObservation and (b) PlayAgent.

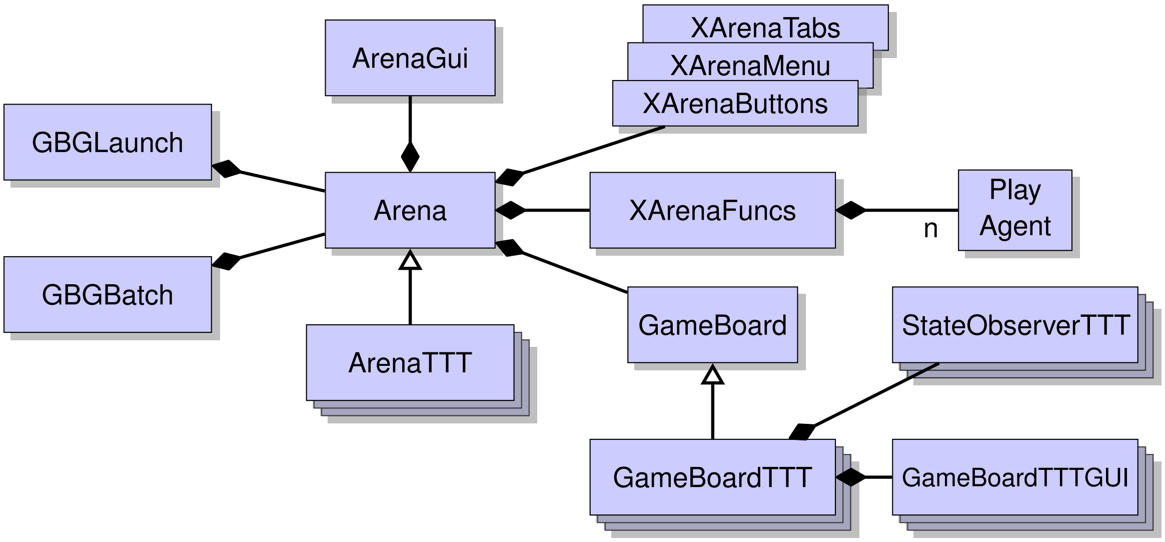

Figure 2: Class diagram GBG: Classes around Arena.

- See GBG Wiki - Board Games for the actual **list of all games** implemented in GBG.

GBG is written in Java and supports parallelization on multiple cores for certain time-consuming tasks. It is available as open source from GitHub[5] and as such – similar to GGP – well-suited for educational and research purposes [Konen, 2019].

2 Overview

2.1 Classes and Interfaces

Figs. 1 and 2 give a first overview of the classes in GBG. In these UML-like diagrams, an interface is depicted as blue half circle, an open-triangle arrow connects a child class with

[5] https://github.com/WolfgangKonen/GBG

a parent class, and a diamond-shaped arrow connects a member with its containing class.

Interface StateObservation is the main interface a game developer has to implement once he/she wants to introduce a new game (Fig. 1a). A class derived from ObserverBase and implementing 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 (Fig. 1b). 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 is the abstract class Arena (Fig. 2). In the Arena all agents meet: They can be loaded from disk, they play a certain game, there can be competitions. If the Arena has train rights, then it has additional options to parametrize, train, inspect, evaluate and save agents. The classes **XArenaMenu**, **XArenaTabs**, **XArenaButtons** and **XArenaFuncs** are helper classes to implement the functionality of Arena.

The Java application classes GBGLaunch and GBGBatch are starter programs and have an Arena object. They allow to start all game arenas, either with GUI or as batch w/o GUI.

The helper classes Feature, XNTupleFuncs, Evaluator and ACTIONS (+ ACTIONS_VT, ACTIONS_ST) support the abstraction in GBG.

See GBG Wiki - Board Games for the actual list of games implemented in GBG.

2.2 Help for GBG

The GBG framework offers various forms of help resources:

- For developers: `https://github.com/WolfgangKonen/GBG/wiki`. The wiki of GBG's GitHub page, which gives a quick overview and tips for **installation and configuration**.

- For developers: *this* technical report, offering an in-depth description of classes and interfaces in GBG. It is available under `https://www.gm.fh-koeln.de/ciopwebpub/Konen22a.d/TR-GBG.pdf`.

- For users: **GBG Help**: Start GBG and select *Help – Help File as PDF* to see a description of the GBG user interface and an in-depth description of all parameters for all agents. Alternative location: *GBG on GitHub – resources – HelpGUI-Arena-GBG.pdf*.

- For users: Start GBG and select *Help – Game Rules as PDF* to show the playing rules of all games implemented in GBG. Alternative location: *GBG on GitHub – resources – GameRules-GBG.pdf*.

3 Classes in Detail

3.1 Interface StateObservation and abstract class ObserverBase

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(ACTIONS)`),
- copying the current state,
- getting the player who has to move in the current state (`getPlayer()`),
- getting the game score of the current state (`getGameScore(int i)`) from the perspective of player `i`,
- signaling end and winner of the game

Example classes implementing StateObservation are: **StateObserverTTT**, **StateObserver2048**, ..., **ObserverBase**.

Many methods of StateObservation e.g. setters, getters and other common methods, have their defaults implemented in abstract class **ObserverBase**. It is strongly recommended to derive a new StateObservation class from class **ObserverBase**, i. e.

```
class StateObsXYZ extends ObserverBase implements StateObservation.
```

This allows that default methods have not to be re-implemented in every class derived from StateObservation. It is however always possible to override them.

3.2 Interface PartialState and class PartialPerfect

Interface **PartialState** is a base of StateObservation and encapsulates the partial-state part of StateObservation. Partial states are needed for imperfect-information games. Sec. Games with Partial States (Sec. 5) has further information on this.

Class **PartialPerfect** provides a default implementation of all methods in **PartialState** for perfect-information games.

3.3 Interface StateObsNondeterministic

For nondeterministic games there is an additional interface **StateObsNondeterministic** derived from StateObservation. More details are in Appendix A.

3.4 Classes StateObsWithBoardVector and BoardVector

Class **StateObsWithBoardVector** is a container class containing a **BoardVector** object and its creating StateObservation object.

Class **BoardVector** contains an `int[]` representation of a StateObservation object. See Appendix C.1 for a specific example and Sec. 8.9 for several **BoardVector**-creating methods.

3.5 Interface PlayAgent and abstract 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[6] starting from state `sob`.

Many methods, e.g. setters, getters and other common methods, have their defaults implemented in abstract class **AgentBase**. It is strongly recommended to derive a new agent class from class **AgentBase**, i. e.

```
class AgentXYZ extends AgentBase implements PlayAgent
```

3.5.1 List of Agents implemented in GBG

Classes implementing interface PlayAgent and derived from AgentBase are shown in Table 1 and listed below:

- 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 ExpectimaxNAgent for deterministic and nondeterministic games, resp.[7]

[6] An *episode* is one specific game playout.

[7] 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.

Table 1: Agents available in GBG.

agent	game	remark
generic agents		
RandomAgent	all	acts completely random
HumanPlayer	all	human play
MaxNAgent	all	generalized 'Minimax' [Korf, 1991]
ExpectimaxNAgent	all	MaxN for nondeterministic games
MCAgent	all	Monte Carlo
MCTSAgentT	all	Monte Carlo Tree Search [Browne et al., 2012]
MCTSExpectimax	all	MCTS extension for nondeterministic games [Kutsch, 2017]
TDAgent	all	TD(λ) agent according to Sutton and Barto [1998] with user-supplied features
TDNTuple4Agt	all	TD(λ) agent with n-tuple features [Lucas, 2008]
Sarsa4Agt	all	SARSA agent (state-action-reward) [Sutton and Barto, 1998] with n-tuple features [Lucas, 2008]
QLearn4Agt	all	Q-learning agent [Sutton and Barto, 1998] with n-tuple features [Lucas, 2008]
wrapper agents		
MaxN2Wrapper	all	MaxN wrapper around inner agent
ExpectimaxNWrapper	all	ExpectimaxN wrapper around inner agent
MCTSWrapper	all	MCTS wrapper around inner agent
MCTSEWrapper	all	MCTS Expectimax wrapper around inner agent
game-specific agents		
AlphaBetaAgent	Connect-4	near-perfect Connect-4 player (alpha-beta search with opening books) [Thill, 2015]
BSBJA	BlackJack	Basic Strategy BlackJack agent, plays perfect
BoutonAgent	Nim	perfect Nim player (theory of Bouton [1901])
BenchPlayer	Othello	baseline Othello player, weighted piece counter
HeurPlayer	Othello	baseline Othello player, weighted piece counter
Edax2	Othello	very strong Othello player (only for Windows) [Delorme, 2017]
PokerAgent	Poker	a medium-strength Poker agent
DAVI3Agent	RubiksCube	Deep Approx. Value Iter. [McAleer et al., 2018]
DAVI4Agent	RubiksCube	Deep Approx. Value Iter. [Agostinelli et al., 2019]

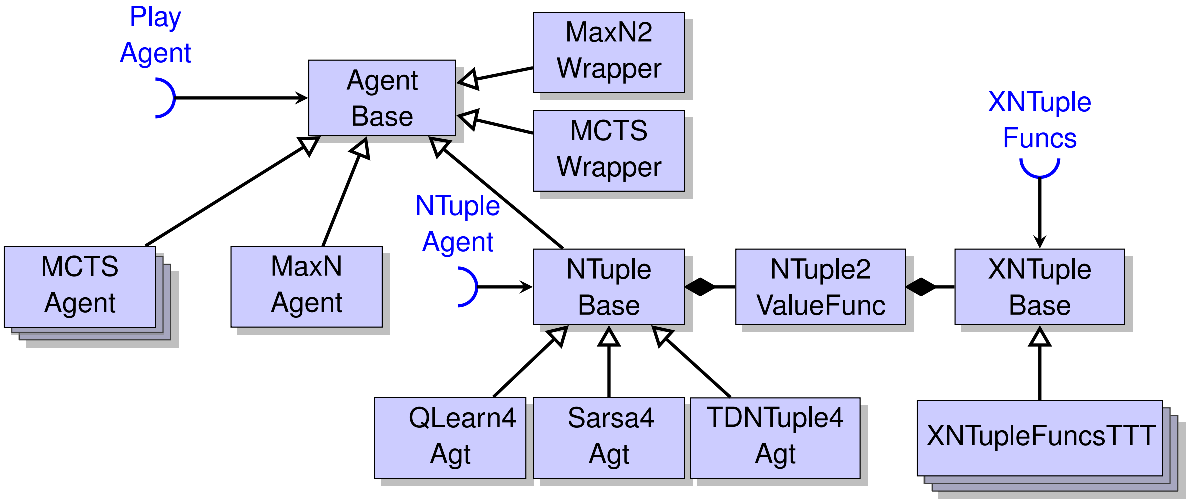

Figure 3: Class diagrams GBG: Classes around AgentBase.

- MaxNAgent: the generalization of Minimax to N-player games with arbitrary N (see Korf [1991]). It maximizes the kth score in a score tuple.

- ExpectimaxNAgent: the generalization of MaxNAgent to nondeterministic games: alternating layers of chance nodes and expectimax nodes. Since 2021 it offers also an extension for imperfect-information games (e.g. Poker, Kuhn Poker, Blackjack), that is, games with partial states.

- MCAgent: Monte-Carlo agent (no tree)

- MCTSAgentT: 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(λ) agent (temporal difference reinforcement learning) with neural network value function (see Sec. 8.8 for more details). This agent requires a Feature object in constructor, see Sec. 3.7.

- TDNTuple3Agt: ***Deprecated***, former version of TDNTuple4Agt where all cells had to have the same number of position values.

- TDNTuple4Agt: TD(λ) agent (temporal difference reinforcement learning) using n-tuple sets as features (see Sec. 8.7, 8.9 and Appendix C for more details). This agent requires an object of class XNTupleFuncs in constructor, see Sec. 3.8. Uses n-tuples where each cell may have individual position values, see Sec. 8.7.

- QLearn4Agt: Q-learning agent 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. 3.8.

- Sarsa4Agt: 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. 3.8.

The last five agents are reinforcement learning agents (temporal difference). The last four agents are based on n-tuple features. TDNTuple4Agt is the generic implementation of a TD-n-tuple-agent. More details on TD(λ) (temporal difference learning, reinforcement learning for games, eligibility traces) can be found in the technical reports Konen [2015] (outdated) and Konen and Bagheri [2020a,b] (up-to-date, including *final adaptation*).

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

Some of the agents (RandomAgent, HumanAgent, 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, TDNTuple3Agt, TDNTuple4Agt, SarsaAgt) require training, they are after construction in state INIT.

Classes implementing PlayAgent should also 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.

3.5.2 The Wrapper Agents

There are four wrapper agents in Table 1. **MaxN2Wrapper** and **ExpectimaxWrapper** can be wrapped around any other agent. This means that an inner agent is wrapped by either MaxNAgent or ExpectimaxNAgent of prescribed depth. Which of the two is taken depends on the deterministic nature of the game. If the wrapper is selected, then – instead of using the inner agent directly – a tree of prescribed depth is constructed and only at the leafs of the tree the inner agent's value function is used. This allows the agent to 'look farther' and thus make better decisions. The price to pay for this is a longer execution time. Especially if the branching factor of the game is high, wrapper depths larger than 2 or 3 should be used with caution.

As shown in Fig. 3, **MaxN2Wrapper** is not derived from MaxNAgent but from AgentBase. This is only because it simplifies software maintenance. Algorithmically, both follow the same principles.

Two other wrappers **MCTSWrapper** and **MCTSEWrapper**, inspired by AlphaZero [Silver et al., 2017], wrap an MCTS (MCTS-Expectimax) around any inner agent. Which of the two is taken depends on the deterministic nature of the game. **Normally, the wrappers will be activated after training** (to improve the planning capabilities during play and evaluation). If they are activated prior to training, MCTS iterations will be done during self-play training, which improves the quality of the selected moves but which also may take very long. See Scheiermann [2020], Scheiermann and Konen [2022] for further information.

The wrappers are not selectable via the agent select box in the Arena GUI. Instead, specify in the agent select box the *inner agent*. To wrap this agent, select in the combo box *Wrapper Mode* of tab *Wrapper pars* (Params window) a mode different from *None*:

- If *Wrapper Mode = MaxN Wrapper* (stands for both MaxN and ExpectimaxN wrapper), set *Wrapper nPly* to a value *nply* greater than 0. Then a wrapper with depth *nply* will be constructed and wrapped around the inner agent.
- If *Wrapper Mode = MCTS Wrapper* (stands for both MCTS and MCTSE wrapper), set *Wrapper MCTS* to a value *iter* greater than 0 and set optionally other MCTS wrapper parameters (see also Appendix G). Then a wrapper with *iter* MCTS iterations will be constructed and wrapped around the inner agent.[8]

3.6 Game Score and Game Value

3.6.1 Conventions for 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 complicated 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(int i)
  ```

  returns the sum of rewards in state `so`, seen from the perspective of the player `i`. Most 2-player games will give the reward only in the end (win/tie/loss), so that for those games `so.getGameScore(i)` is usually 0 as long as `so` is non-terminal. But other games (e.g. 2048, Blackjack, Poker) give rewards not only at the end of an episode. Then the game score will be the sum of all rewards given so far in this episode (**cumulative** score or reward). For example, the game score in Poker for player i is the amount of chips that she has in state `so`, and this amount changes at the end of each round. In 2048, the game score is the cumulative sum of all tile merges.
- 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 estimate is always the negative of the next state's score (negamax principle). Fig. 4 shows an example.

[8] Older agents stored on disk may have their wrapper params set in ParOther (now deprecated) and have ParWrapper=null. In this case, ParWrapper will be instantiated from ParOther and suitable other defaults.

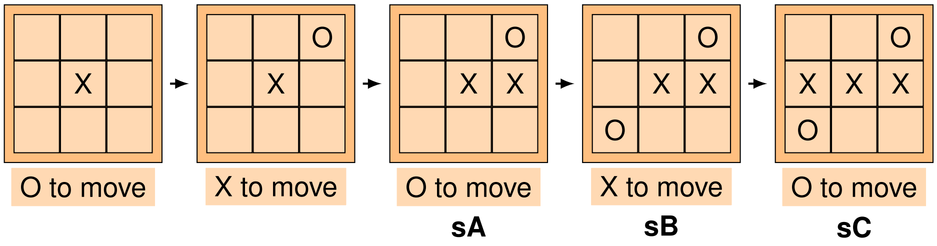

Figure 4: A succession of states in TicTacToe: If O makes in state **sA** the losing move leading to **sB**, then **sB** is a clear win for X, and so is the terminal state **sC**. The game score for **sC** from the perspective of O, the player to move in **sC**, is -1. The game values (see Sec. 3.6.3) for **sB** and **sC** are $+1$ and -1, resp.

> *Example: State **sC** in Fig. 4 (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**.getPlayer())*$= -1$ *for player O.*
>
> ```
> sC.getGameScore(sC.getPlayer()) = -1;
> sC.getGameScore(sB.getPlayer()) = +1;
> sC.getGameScore(sA.getPlayer()) = -1;
> ```

sC.getGameScore() = sC.getGameScore(sC.getPlayer());

- `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.[9]

[9] If a precise maximum game score for a certain game is not known, a reasonable 'big' estimate is usually also sufficient.

3.6.2 Difference between Game Score and Game Reward

In many cases, game score and **game reward** are the same and can be used synonymously. But in some cases it might be beneficial – e.g. when training an agent – to train it on a different reward than the raw score of a game state. Example: In 2048, the number of empty tiles might be an indicator of the player's mobility and indicate an difference between states that otherwise have the same game score. This fact can be used to form a game-specific reward (by combining it with the cumulative game score or by replacing it).

More details on **game reward** are in Appendix B.

3.6.3 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 state 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 intermediate (score-0) states in TicTacToe might differ in their game value: While a first state might be an inevitable loss for the player-to-move, a second state might be just one step away from a sure win. Both states have the same game score (0), but obviously quite different game values. 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. Or it can be based on sophisticated (deep or n-tuple) neural networks as approximators.

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(int i,boolean)`
- `PlayAgent.estimateGameValue(StateObservation so)`

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

3.6.4 Ranges for Game Score and Game Value

In principle the range of game values is arbitrary, only the order of values matters for agent decisions. So a shift or stretch of the game value range should be normally irrelevant.

But there are two other reasons why the range of the game value or game score is important:

- When displaying game values on the gameboard or printing game values, it is better to have either $[-1, 1]$ or $[0, 1]$ as game value range. Classes derived from GameBoard will often multiply all game values by 100 and then they will display numbers in the convenient range $[-100, 100]$.
- When a neural network is used for predicting game values, it may contain a sigmoid function in the output neuron. Depending on the type of sigmoid, the range of neural network outputs is either $[-1, 1]$ or $[0, 1]$. The game values should fall into the neural network output range, otherwise the net cannot learn them.

3.7 Interface Feature

Some classes implementing PlayAgent need a game-specific feature vector. As an example, consider **TDAgent**, the general TD(λ) 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 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, ...

More details on how to set up a new Feature class are in Sec. 8.8.

3.8 Interface XNTupleFuncs and abstract class XNTupleBase

There are the special agents **TDNTuple3Agt** and **SarsaAgt** to realize TD- or SARSA-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. 8.9 for more details on the member functions of XNTupleFuncs and Appendix C for a more detailed 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 needs a XNTupleFuncs object, it will call this method `makeXNTupleFuncs()`. This will take place whenever an n-tuple agent object is constructed, because these n-tuple agent constructors need an XNTupleFuncs object as parameter.

It is strongly recommended to derive a new XNTuple class from class XNTupleBase i. e.

```
class XNTupleXYZ extends XNTupleBase implements XNTupleFuncs.
```

XNTupleBase is an abstract class which has however default implementations for members `instantiateAfterLoading()` and `makeBoardVectorEachCellDifferent()` and others.

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

More details on how to set up a new XNTupleFuncs class are in Sec. 8.9.

3.9 Interface GameBoard

Interface **GameBoard** prescribes things specific to the game board. All implementing classes GameBoardXyz have an optional member **GameBoardXyzGui** that realizes 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.
- `updateBoard(so, ...)`: Showing or updating the current game state (StateObservation `so`) in the GUI and enabling / disabling the GUI elements.
- Human interaction with the board: see Sec. 3.12.

- Returning its current StateObservation object (`getStateObs()`).
- `chooseStartState()`: This method returns randomly one out of a set of different start states. This is useful when training an agent in such a way 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).*

There is a peculiar thing to note about GameBoard's object StateObservation `m_so`: Methods that interact with GameBoard, like Arena's methods `PlayGame` or `InspectGame`, call in their main while-loop

```
so = m_gb.getStateObs();
```

and establish `so` to be a **reference** to `m_so`. This means that each action applied to `so` is applied to `m_so`. It is important to work with this reference and **not** to make – e.g. in `gb.updateBoard(so,...)` – a copy of object `so`. This is because a human interaction with the board (see Sec. 3.12) acts on `m_so`.[10] and the resulting state is only correctly passed over to the `so`-context if `so` is a reference to `m_so`.

Classes implementing GameBoard should *not* inherit from JFrame in order to make GBG runnable also on systems without GUI capabilities. Instead, each class implementing GameBoard has an element `GameBoardXyzGui m_gameGUI` which holds all GUI-related stuff. But `m_gameGUI` can be also `null` in the non-GUI case. Example classes implementing GameBoard are: **GameBoardTTT**, **GameBoard2048**, ...

3.10 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 *Compete* variant in menu *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

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

the argument `mode` allows derived classes to create different types of evaluators. These may test different abilities of PlayAgent.[11]

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

[10] since in the context where `HGameMove` is called, the reference `so` is not available

[11] 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 know what the perfect action for a certain state is and that (b) a perfect Evaluator should evaluate the actions of PlayAgent for *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.

```
abstract protected boolean evalAgent();
```

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. This test for a premature training stop is however only done if `stopTest>0` and `stopEval>0`.[12]

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 MaxN player).

Concrete objects of class Evaluator are usually constructed by Arena's factory method

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

Example classes derived from Evaluator: **EvaluatorTTT**, **Evaluator2048**, ...

More details on how to set up a new evaluator are in Sec. 8.10.

3.11 Abstract Class Arena

Class **Arena** is an abstract class for loading agents and playing games. Its constructors are

```
public Arena(String title, boolean withUI);
public Arena(String title, boolean withUI, boolean withTrainRights);
```

If argument `withTrainRights` is `true`, then the constructed Arena object will have train rights (see below).

Why is **Arena** an abstract class? – It 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

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

[12] `stopTest` and `stopEval` are members of ParOther.

If it is not overridden by derived classes, it will throw a `RuntimeException` (no game-tailored object implementing the Feature interface is 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 agents **TDNTuple3Agt** or **SarsaAgt** are 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 for 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. 9 Open Issues for planned extensions to the Arena functionality.

Class Arena has in method `run()` a long switch statement on member `Task taskState` which takes one of the possible `Task` values (`PARAM`, `TRAIN`, `COMPETE`, ...) and controls what Arena is actually doing.

If class **Arena** has train rights then it has this *additional* functionality:

- specifying all parameters for agents (including parameters for training),
- training agents (one or multiple times),
- saving agents.

Examples of non-abstract classes derived from Arena: **ArenaTTT**, **Arena2048**, ...
They usually have a method

```
public static void main(String[] args)
```

for starting a specific game.

3.12 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. This should trigger GameBoard's `HGameMove`. GameBoard's `HGameMove` is responsible for checking whether the human action is legal (`isLegalAction()`).[13] If so, then GameBoard's `HGameMove` issues an `advance(ACTIONS)`. Method `advance(ACTIONS)` opens the possibility for invoking random elements from the game environment (e. g. adding a new tile in 2048), if necessary.

It is also important for a second reason that GameBoard's `HGameMove` calls `advance()`: Some games (e.g. ConnectFour, Sim) can only detect a win- or a lose-situation of a state correctly, if the state was reached via `advance()`. This is because it is much simpler to detect whether the last action created a win- or a lose-pattern than it is to scan the whole board for such patterns.

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.

3.13 The Param Classes

Each agent or group of agents has associated classes for setting its parameters. Table 2 gives an overview of these param classes. These classes come in two flavours:

...Params For example, class **TDParams** holds the parameters for all parameters related to TD (Temporal Difference learning). It holds additionally the GUI (param tab) to set them. Similar for all other ...Params classes. These classes are usually derived from `Frame` or `JFrame` and as such their objects tend to be rather big.

Par... For example, class **ParTD** holds solely the parameters related to TD. Thus, objects of class Par... are much smaller and can be easily copied, attached to other objects, passed to other methods, saved and loaded.

It is advisable to use the classes ...Params only once for the multi-pane Param Tabs window. For all other use cases (inside agents, loading and saving to disk, ...) you should use the Par... variant.

Classes **ParOther** and **OtherParams** hold parameters relevant for all agents in one way or the other. Among these parameters are:

- the evaluator modes to use during quick evaluation (*Quick Eval Mode*) or during training (*Train Eval Mode*),

[13] see method `HGameMove(x,y)` in GameBoardTTT for an example.

Table 2: The Param classes in GBG.

Par...	...Params	related agents
ParTD	TDParams	TDAgent, TDNTuple3Agt, TDNTuple4Agt, SarsaAgt (TD settings)
ParNT	NTParams	TDNTuple3Agt, TDNTuple4Agt, SarsaAgt (n-tuple [Lucas, 2008] & temporal coherence [Beal and Smith, 1999] settings)
ParMaxN	MaxNParams	MaxNAgent [Korf, 1991], ExpectimaxNAgent and wrappers (Sec. 3.5.2)
ParMC	MCParams	MCAgent
ParMCTS	MCTSParams	MCTSAgentT
ParMCTSE	MCTSEParams	MCTSExpectimaxAgent
ParEdax	EdaxParams	Edax parameter (only Othello)
ParRB	RBParams	Replay buffer parameters (all trainable agents)
ParWrapper	WrapperParams	Wrapper parameters (all agents) (Sec. 3.5.2)
ParOther	OtherParams	Other parameters (all agents)

- parameters relevant for *all* trainable agents (`numEval`, *Episode Length*, *Choose Start 01*, *Learn from RM*),

See *GBG Help* for more detailed information.

3.14 The ACTION Classes

There are three classes (public subclasses of class Types) for specifying actions:

ACTIONS is an action specified by an `int` and a Boolean predicate `randomSelect` whether it was selected by a random move or not.

ACTIONS_ST is derived from ACTIONS and has additionally the ScoreTuple of this action.

ACTIONS_VT is derived from ACTIONS and has additionally a value table for all available actions, the value of this action and the ScoreTuple of this action.

3.15 GBG Starter Classes

3.15.1 GBGLaunch

GBGLaunch is the class that allows to launch all the games implemented in GBG. This can be done either via GUI or via command line arguments. See *GBG Help*, Sec. *GBG Launcher*, for details.

Note that the third parameter (either `T` or `P`) of **GBGLaunch**'s `main` controls whether Arena is started with train rights or without. Default is `T`.

If a game is scalable, i.e. it comes in different variants, the scalable parameters may be specified in the launcher GUI prior to calling the game.

3.15.2 GBGBatch

GBGBatch is the class that allows to run (time-consuming) tasks in batch mode, usually without any GUI. **GBGBatch** is completely configurable via command line arguments. See See *GBG Help*, Sec. *GBG Batch*, for more details.

3.15.3 SetupGBG

SetupGBG is the base class for both GBGLaunch and GBGBatch and bundles common functionality.

4 Games with Rounds

Poker, Kuhn Poker and Blackjack are games where an episode consists of several rounds. All other games in GBG do not have rounds (i.e. each episode is exactly one round).

For round-based games: If a round is completed, a new round with completely new settings starts, only the chips of the players remain what they were at the end of last round. Two things are necessary to change in the general interface:

1. Signaling the end of round (because a MC rollout may want to stop there and a TD learning cycle may NOT transport the value from the successor state in the new round back to the end-of-round-state).
2. A mechanism to stop the game play in the GameBoard at the end-of-round-state (after showdown, to let the user watch the round result) and – only after the user has hit the „Continue“ button in the GameBoard – continue to the next state, which is the start of a new round.

To achieve this, we add the following to the general GBG part:

- New methods for interface StateObservation
  - `boolean isRoundOver()`
  - `void setRoundOver(boolean)`
  - `void initRound()`
- For games with rounds we change the methods `advance` and its helper methods `advanceDeterministic` & `advanceNondeterministic`: If it comes to an end-of-round-condition, only **call setRoundOver**, but **DO NOT call initRound** → the state remains in an artificial round-over position after showdown: the cards are all open on the table and the resulting distribution of the chips in the pot is calculated. But the transition to the first state of the new round is not yet made.
- This transition is now delegated to Arena's `PlayGame()` where the relevant part in its while-loop reads:

```
...
actBest = pa.getNextAction2(...)
so.advance(actBest)
...
gb.updateBoard(so, ...) // which may stop if so.isRoundOver()
...
if (so.isRoundOver() && !so.isGameOver()) so.initRound();
if(so.isGameOver()) {
 ...
 break;  // out of while
}
```

- Now GameBoard's `updateBoard(so,...)` may stop if it detects `so.isRoundOver()` and may continue only after the user has hit the „Continue“ button.

5 Games with Partial States

Poker, Kuhn Poker and Blackjack are **imperfect-information games**, that is, games where each player has only partial information about the current state. We call this a *partial state*. As an example consider a state in Kuhn Poker where the player only knows his own card (but not the opponent's card). All other games in GBG are perfect-information games.

When an agent gets a state of an imperfect-information game, it gets only the *partial state*, that is, a state where the elements of other players are hidden (i. e. by replacing their cards with empty cards or null values). If the episode ends (or if the agent reasons about the different possibilities), the missing elements are completed (either by random completion or by walking through all possibilities).

The base interface PartialState of StateObservation has the methods to implement this behavior:

- `StateObservation partialState()`
- `boolean isPartialState(...)`
- `ArrayList<ACTIONS> getAvailableCompletions(...)`
- `Tuple<StateObservation,Double> completePartialState(...)`

See the JavaDoc of PartialState for further information.

PartialState's default implementation for perfect-information games is in class **PartialPerfect**: partialState() returns just `this`, isPartialState(...) returns always `false` and completePartialState(...) returns `<this,1.0>`. **PartialPerfect** is a base class of ObserverBase.

6 JUnit Tests

Package `test` of GBG offers various software tests which are runnable within the JUnit framework.

Packages `test/games`, `test/ludiiInterfaceOthello`, `test/tools` offer quick-running tests for various aspects of the GBG software. These tests are usually strict, i.e. if any of them fails, there is usually something wrong with the tested software part.

Package `test/controllers` offers various tests for the agents in GBG. Some of them, especially in `TDNTuple3AgtTest`, have a longer runtime (up to 25 min) since they may train a complete **TDNTuple3Agt** and test whether the resulting agent matches some predefined evaluation thresholds. Due to the stochastic nature of agent training, these tests are not strict, i.e. there is a low probability that a certain agent may not pass the evaluation threshold. If it misses the threshold only by a small margin, the user might try to run this test again. If it misses the threshold by a large margin, there is usually something wrong with the tested software part.

7 Design Principles

The development of GBG is based on design principles that should help to write well-maintainable, versatile and error-free code:

7.1 Separate GUI from core GBG

Have a clean separation of GUI elements and core GBG functionality. It should be possible to perform time-consuming GBG tasks on different machines, but not all machines are capable of executing GUI code (X11, Swing etc.). Therefore, the core GBG functionality should be runnable without any GUI elements. GBG classes should **not inherit** from GUI classes (like JFrame, JPanel). Instead, GUI elements, where necessary, should be optional members of GBG classes. 'Optional' means that these members may be `null` without obstructing any GBG core functionality. Examples:

- Class Arena has the optional member **ArenaGui**. **ArenaGui** extends `JFrame`.
- Class **ParMC** has the optional member `MCParams`. `MCParams` extends `Frame`.

When a batch program like GBGBatch is run, it should not need any GUI elements. Then all these optional parameters are left at their initial value `null` and the program runs even on machines without X11.

7.2 Number the Players with $0, 1, 2, \ldots$

Players should be numbered with $0, 1, 2, \ldots$. Do not use the numbering $1, 2, 3, \ldots$, because this needs transformations in all cases where we have arrays over players.[14] The code is

[14] For 2-player games there is also the string constant `String[] GUI_2PLAYER_NAME = {"X", "O"}`, but this is only for display purposes in the GUI.

less well-maintainable, since we might be unclear in which player-numbering scheme we are.
As a convention, always player 0 will start a game episode.

7.3 Copy Constructor for StateObservation

All classes `StateObserverXYZ` implementing StateObservation should have a copy constructor and a method `copy()`. This is to ensure that really all elements of the derived class are copied.

Each class derived from StateObservation should usually extend **ObserverBase**.

Then the first line of the copy constructor should be `super(other);` in order to guarantee that all elements of **ObserverBase** are copied.

```
newState = new StateObserverXYZ(this);
```
and
```
newState = this.copy();
```
do the same. The latter is just a shorthand for the former.

If elements of `StateObserverXYZ` are complex objects (lists or arrays) do not forget to use the appropriate `clone()`-operations.

It is usually a bit faster (much faster in the case of Othello!) to use in the copy constructor `other.availableActions.clone()` instead of `setAvailableActions()`. Note that the clone operation makes a copy of the ArrayList

```
ArrayList<ACTIONS> availableActions,
```
but not of the actions contained in it. This is usually OK, since a state will not change the actions contained in `availableActions`.

7.4 Avoid Excess Copying

Think whether copying of elements is really required.

It may slow down operation. And it may be a cause of errors, if it is somehow forgotten in a part of the code to copy elements, but another part of the code silently assumes that they have been copied.

Of course there are circumstances where copying is a must, but use it only if necessary.

7.5 Loading and Saving Agents

All agents that store parameters should override the interface method

```
fillParamTabsAfterLoading().
```

This ensures that the stored parameters appear immediately in the Param Tabs.

All agents that have transient members should override the interface method

```
instantiateAfterLoading().
```

This ensures that transient members are properly initialized.

See also Sec. 8.6 and 8.13.

8 Use Cases and FAQs

8.1 My first GBG project

Follow the install and configuration tips from the GitHub Wiki:
https://github.com/WolfgangKonen/GBG/wiki – Install and Configure
in order to install the GBG framework.

Run GBGLaunch as Java Application. Once started, you can use all functionality of Arena (e. g. play, inspect, compete, train, see Sec. 3.11) for all games registered in GBGLaunch. For more detailed information on how to train an agent see Sec. 8.3.

8.2 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 five interfaces for your game:

- `StateObserverXYZ implements StateObservation`
- `GameBoardXYZ implements GameBoard`
- `EvaluatorXYZ extends Evaluator`
- `FeatureXYZ implements Feature, Serializable` (only needed, if you want to use the trainable agent **TDAgent**, see Sec. 8.8).
- `XNTupleXYZ extends XNTupleBase implements XNTupleFuncs, Serializable` (only needed, if you want to use the trainable n-tuple agent **TDNTuple3Agt** or **SarsaAgt**, see Sec. 8.9).

Once this is done, you only need to write a very 'thin' class ArenaXYZ with suitable constructor

```
public ArenaXYZ(String title, boolean withUI, boolean withTrainRights)
```

Class ArenaXYZ overwrites the abstract methods of class Arena 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. 8.8 and Sec. 8.9)

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

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");
}
```

You can inspect class **ArenaTTT** in order to see for all this an example specific to the game TicTacToe.

Finally you need to register your new game for GBGLaunch and GBGBatch To do so, add a few lines of code in the switch statements of SetupGBG's methods `startGBG`, similar to the statements for the other games.[15]

Then your game will show up the next time you start GBGLaunch and you can use for your game all the functionality laid down in Arena and all the wisdom of the AI agents implementing PlayAgent. Cool, isn't it?

8.3 How to train an agent and save it

1. Create and launch an Arena object with train rights
2. Select a trainable agent and set its parameters
3. Set general training-specific parameters:
   - `maxTrainNum`: *Train games* = number of training episodes (main GBG window, Arena),
   - `numEval`: after how many episodes an intermediate evaluation is done (*Other pars* tab),
   - `epiLength`: *Episode length* = maximum allowed number of moves in a training episode.[16] (*Other pars* tab).

[15] If your game has scalable parameters, you may want to add also appropriate code in the switch statements of SetupGBG's method `setDefaultScaPars` and GBGLaunch's method `adjustScaParGuiPart`.

[16] If `epiLength` 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. Set to −1, if no premature stop is allowed.

4. Train the agent & visualize intermediate evaluations.

5. Optional: *Inspect V* = inspect the value or Q-function of an agent (how it responds to certain board situations).

6. Save the agent via menu item *Agent N – Save Agent N*.

Note that also agents like **MCAgentN**, **MCTSAgentT** and so on are 'trainable'. That is, if *Train* or *MultiTrain* is executed on them, they will perform the desired number of self-play episodes. Of course these agents do not really 'train' i. e. modify their behavior. It is just for the side effects: (i) the number of moves per second – train time, excluding evaluation time – is measured and reported and (ii) the agents are repeatedly evaluated.

8.4 Which games are currently implemented in GBG?

See GBG Wiki - Board Games for the actual list of games implemented in GBG.

8.5 Which AI's are currently implemented for GBG?

See Sec. 3.5.1 and Table 1 for a list of all AI's (agents), i. e. classes that implement interface PlayAgent.

8.6 How to write a new agent (for all games)

Of course your new agent `NewAgent` has to implement the interface PlayAgent. It is strongly recommended that you derive your new agent from AgentBase to inherit basic functions 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. – If the agent needs instantiation after loading or parameter tabs setting after loading, add this functionality in member methods `instantiateAfterLoading` (Sec. 8.13) and `fillParamTabsAfterLoading()` (Sec. 8.14). If nothing is needed here, AgentBase will provide appropriate (empty) dummy stubs for these methods.

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")) ...
  ```

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

If the agent requires a whole set of new parameters which do not fit into the existing Param classes, then construct new Param classes `...Params` and `Par...` and add `...Params` to the Param tab.

8.7 What is the difference between TDAgent, TDNTuple3Agt and TDNTuple4Agt?

All these agents are trained by TD (temporal difference) learning. They differ only in their feature vectors: For **TDAgent** the user has to specify all features through a **Feature** object (see Sec. 3.7), which contains method

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

On the other hand, the classes **TDNTuple*Agt** construct their features automatically from the given n-tuple sets and position values (see Sec. 8.9 and Appendix C.1). The differences between the three n-tuple agents are:

TDNTuple3Agt is now deprecated, but retained in the GBG software distribution to keep some older agents loadable. It has the following advantages over the former **TDNTuple2Agt** (no longer part of the GBG distribution): TD-learning is only based on *own* experiences of the agent, not on those of the other players. This allows the same TD algorithm to be used for $N = 1, 2, 3, \ldots$ players. It includes the so-called **F**inal **A**daptation **RL** (FARL) method which is described in detail in Konen and Bagheri [2020a,b].

TDNTuple4Agt is the actual TD-n-tuple agent. It keeps all the advantages of **TDNTuple3Agt** plus that it uses class **NTuple4** which is a generalization of **NTuple2** in the sense that every n-tuple cell may have a different number of position values. This allows for the game Rubik's Cube representations that have memory consumption lower by a factor of 5-10.

8.8 How to apply 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 reuse most of the functionality laid down in class **TDAgent** (see Sec. 3.5). But a few things remain to be done:

1. Write a new class derived from Feature

   ```
   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 the lose-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. Another way to form features is to count the number of pieces for each player and let a network learn weights for it. Or the number of pieces in certain positions on the board.[17]

2. Add to ArenaXYZ 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. 3.6.3) for all relevant game states.

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

8.9 How to apply TDNTuple4Agt 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 **TDNTuple4Agt** (see Sec. 3.5). But a few things remain to be done:

1. Write a new class that implements XNTupleFuncs (Sec. 3.8) and is derived from XNTupleBase

```
public class XNTupleFuncsXYZ extends XNTupleBase
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 the BoardVector-creating method

```
BoardVector getBoardVector(so)
```

[17] The drawback of all these features is that they are not very generic: The user has to code the features in a game-dependent way for each new game again. – 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. 8.9, Appendix C and **TDNTuple3Agt** instead).

which transforms a game state `so` into a 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 × 2 mirror reflections), you implement the method

```
BoardVector[] symmetryVectors(BoardVector boardVector, int n)
```

which returns in case '`n=0` or `n=s`' for a given board vector all symmetric board vectors (including itself). If the game has no symmetries, it returns just the board vector itself. If `0 < n < s`, this method returns the board vector itself plus n-1 (randomly selected) of the s symmetry vectors. (This n-option is for games where the full set of symmetry vectors is too big (e.g. Sim and Nim). n is set via element *nSym* in the *NT pars* (n-tuple params) tab.)

There is also a sibling method

```
BoardVector[] symmetryVectors(StateObsWithBoardVector, int n)
```

which is for games where the symmetric states can only be constructed if access to the underlying StateObservation object is provided. This is for example the case in the games Sim and Rubik's Cube. For other games where this is *not* necessary, you just implement *nothing*. Then the default implementation from XNTupleBase is taken:

```
XNTupleBase.symmetryVectors(StateObsWithBoardVector, int n)
```

which just calls `symmetryVectors(BoardVector, int n)`.

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 used by **TDNTuple4Agt** when creating the shape of new n-tuples 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 select in the *NT pars* (n-tuple params) tab the check box *Random n-tuple generation*.

2. Add to ArenaXYZ the overriding method

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

Class ArenaTTT (together with XNTupleFuncsTTT) may be used as a template, showing the implementation for the game TicTacToe.

TDNTuple4Agt 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 **Sarsa4Agt** or **QLearn4Agt** to a new game.

See Appendix C for further information on n-tuples.

8.10 How to set up a new Evaluator

Setting up a good evaluator for a game is not an easy task, because the agent's strength in playing a game depends on its reaction to *all* possible game states, weighted with the relevance of those states. To evaluate this is for most realistic games an intractable task. It can often be only approximated by having different evaluators looking at the problem from different perspectives. Therefore, the Evaluator concept in GBG allows for different evaluator modes.

When testing a deterministic agent against another deterministic opponent, they will always play the same episode, so that the evaluation covers only a tiny part of the complete game tree. And there are only three possible outcomes: win, tie or loss, i.e. {+1,0,-1}. As a consequence it is difficult to decide whether an agent improves or not. A little improvement is achieved when the start state is varied (randomly or by looping through a prescribed set of states). Then the visited fraction of the game tree is slightly bigger. More importantly, the evaluation result is a floating point number (win rate, averaged over a set of different episodes), which signals better than {+1,0,-1} whether an agent improves or not. Therefore, most classes derived from Evaluator should have modes with different start states.

These general aspects should be kept in mind when constructing for a game a new evaluator derived from Evaluator. It is often a good idea to specify different modes where the agent plays against different opponents, either from the default start state or from a set of start states.

When deriving a concrete class from Evaluator, you have to implement the abstract methods of class Evaluator, the most important ones are:

- `getAvailabelModes()`: returns an `int[]` with all available modes,

- `evalAgent()`: run the evaluator with the mode specified in constructor,
- `getTooltipString()`: return a `String` (may be multi-line) describing the different modes (tooltip text of the evaluator mode choice boxes in OtherParams).

A typical constructor `EvaluatorXYZ extends Evaluator` looks like:

```
public EvaluatorXYZ(PlayAgent e_PlayAgent, GameBoard gb, int stopEval,
                    int mode, int verbose) {
  super(e_PlayAgent, mode, stopEval, verbose);
  ...
}
```

8.11 Scalable GUI fonts

When writing a new GUI element, this GUI element may be shown on display screens with largely differing screen sizes. In order to have legible fonts on all such screen sizes, it is advisable NOT to use explicit font sizes like 12, 14, ... Instead it is better to use variable font sizes

```
int Types.GUI_HELPFONTSIZE
int Types.GUI_DIALOGFONTSIZE
```

and similar (see `Types.java`). To define a new font, use for example the form

```
Font font=new Font("Arial",0,(int)(1.2*Types.GUI_HELPFONTSIZE));
```

where the factor 1.2 is optional, if you want to adjust the appearance of the associated text element.

The variable font sizes are automatically scaled to be a certain portion of the screen width. If you want **all** fonts to appear bigger or smaller, you may set

```
double Types.GUI_SCALING_FACTOR
```

in `Types.java` to a value slightly higher or lower than 1.0.

8.12 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 $0, 1, \ldots, N-1$.

The class 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`.

SUM add to `this` all values from the other `tuple2nd`.

8.13 What is the purpose of PlayAgent's instantiateAfterLoading()?

When saving an agent to disk, some agents may not save all their members. Reasons for this are: (a) the member may not be serializable, (b) the member would make the saved agent too big and is not needed for reconstructing it. If the agent is re-loaded from disk, these members will be 0 or `null`. To fully instantiate the agent, we need the method `instantiateAfterLoading()`. This will for example allocate memory for an array, read in static opening books or do other things. Having this method as part of PlayAgent's interface makes the code simpler in all cases where we need this instantiation.[18]

If an agent needs nothing to be done here, the implementer of the agent has nothing to do as well: The base class AgentBase – from which the agent should be derived – has a dummy stub for this method: it just does nothing.

See also Sec. 8.6.

8.14 What is the purpose of PlayAgent's fillParamTabsAfterLoading()?

Similarly, most agents will have parameters which are restored from disk. To make these parameters visible, the appropriate param tabs need to be filled with their values. Method `fillParamTabsAfterLoading()` has the appropriate code for this.

If an agent needs nothing to be done here, the implementer of the agent has nothing to do as well: The base class AgentBase – from which the agent should be derived – has a dummy stub for this method: it just does nothing.

See also Sec. 8.6.

9 Open Issues

The current GBG class framework is still under development. 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:

- Undo/redo possibilities
- Game balancing

[18] Before, a lengthy `switch`-statement with many cases and agent-dependent casts was needed.

- Client-server architecture for game play via applet on a game page. Option for a 'hall of fame'. An example for the game Sim is available from TU Wien [19].
- Replay memory for better training: This idea has been used by DeepMind in learning Atari video games. Played episodes are stored in a replay memory pool and used repeatedly for training.
- The extension of TD-agents to N-player games with $N > 2$ is fully functional but so far tested only on 3-player Sim in a near-trivial configuration (results nearly always in a tie). It is desirable to test it on non-trivial 3-player games. One option may be a 3-player Sim with pre-defined coalition (similar to Skat) or the game Skat itself.

[19] http://www.dbai.tuwien.ac.at/proj/ramsey

A Appendix: Interface StateObsNondeterministic

If a game is nondeterministic it has random elements (like rolling the dices in a dice game or placing a new tile in 2048). Then, `advance(ACTIONS)` 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).

Interface **StateObsNondeterministic** is derived from StateObservation and provides functionality around nondeterministic actions. It splits the usual `advance()` in two parts: `advanceDeterministic()` and `advanceNondeterministic()`. The possible random actions in the non-deterministic part are accessible via `getAvailableRandoms()`. Examples using or implementing **StateObsNondeterministic** are **ExpectimaxNAgent** and **StateObserver2048**.

Why do we need the split of `advance(ACTIONS)` into the two parts

```
advanceDeterministic(ACTIONS)
advanceNondeterministic(),
```

why can we not simply pack the functionality of both into `advance(ACTIONS)`? – This is because some agents, most dominantly **ExpectimaxNAgent** and **MCTSExpectimaxAgent** handle the deterministic and the nondeterministic part of an action differently: Usually an agent selects that deterministic action that **maximizes** the game value. For the nondeterministic part it is different: The agent has no control what nondeterministic action will take place – this is dictated by the game environment. Thus, the agent has to **average** the value over the different possible nondeterministic actions.

If in a certain usage of StateObservation the distinction between the deterministic part and the nondeterministic part is not needed, the user may call `advance(ACTIONS)`, which first calls `advanceDeterministic(ACTIONS)` and then `advanceNondeterministic()`.

Interface **StateObsNondeterministic** has additionally the methods

```
isNextActionDeterministic()
getNextNondeterministicAction()
```

for fine-grained control.

B Appendix: Other Game Value Functions

Sec. 3.6 and 3.6.3 have introduced with

```
StateObservation.getGameScore(int i)
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(int i, boolean rgs)
  ```

  a function returning the **game reward**. This reward is simply the same as game score in case `rgs==true` ('**r**eward is **g**ame **s**core'), 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 MaxN2Wrapper and ExpectimaxWrapper (Sec. 3.5.2) when the prescribed n-ply tree depth is reached. These wrappers implement `estimateGameValue(sob)` and let it return the wrapped agent's game value via

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

where `sob` is the state at the leaf node of the wrapper tree.

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.

Tables 3 and 4 give an overview over all functions in GBG returning a game score or a game value. Summary of main facts:

Table 3: Summary of all game score and game value functions in GBG.

method	remark
class StateObservation (`so`)	
getGameScore(int i)	game score of `this` from perspective of player i
getGameScoreTuple()	a ScoreTuple with game scores for all players
getReward(int i,rgs)	game reward of `this` from perspective of player i
getRewardTuple(rgs)	a ScoreTuple with rewards for all players
getStepRewardTuple()	a ScoreTuple with step 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

Table 4: *Deprecated* game score and game value functions in GBG.

method	remark
class StateObservation (`so`)	
getGameScore()	*(deprecated)* use `so.getGameScore(so.getPlayer())`
getGameScore(refer)	*(deprecated)* use `so.getGameScore(refer.getPlayer())`
getReward(rgs)	*(deprecated)* use `so.getReward(so.getPlayer(),rgs)`
getReward(refer,rgs)	*(deprecated)* use `so.getReward(refer.getPlayer(),rgs)`

- The methods in Table 4 are deprecated because they can be expressed equivalently by other methods as indicated in Table 4. It is better to have a thin interface with `getGameScore(int i)` because in this way it is clearer what a class derived from StateObservation has to implement.

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

- `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 substract 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 when 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, game reward and game value methods have associated `...Tuple` versions: They return instead of a single game score or game value a ScoreTuple (Sec. 8.12) of N values for all players $0, 1, \ldots, N-1$.

- Interface StateObservation delivers with

  ```
  getStepRewardTuple()
  ```

  a method that adds a reward for each step (transition). This is a 0-ScoreTuple for all games except for Rubik's Cube, where it returns a ScoreTuple with the negative value `CubeConfig.stepReward`. This ensures that from two transition sequences leading to a solved cube the shorter one gets a higher total reward. If the step reward were not present, endless cycles would occur.

C Appendix: N-Tuples

C.1 Board Cell Numbering and Board Vectors

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 4×4 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

```
BoardVector XNTupleFuncs::getBoardVector(StateObservation so)
```

returns a **board vector**: Each BoardVector object contains an `int[]` vector whose length is the number of board cells

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

and the `int[]` vector carries the position value for each board cell according to the board cell numbering.[20]

A cell's **position value** is a game-specific coding of all allowed board cell states. It is a number in $\{0, 1, 2, \ldots, P-1\}$ with $P =$ `XNTupleFuncs::getNumPositionValues()`.

> *Example: The canonical board cell numbering for the game TicTacToe runs from 00 to 08. The position values are 0: O, 1: empty, 2: X. Each BoardVector object has length 9. For state **sA** in Fig. 4 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. 5 shows two examples of 4×4 boards with fixed (user-specified) n-tuple sets. The canonical cell number is obtained from

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

[20] In the game Sim, the 'board cells' are the links in the Sim graph. In the game Rubik's Cube, the 'board cells' are the cubie faces (stickers) needed to uniquely define a cube state.

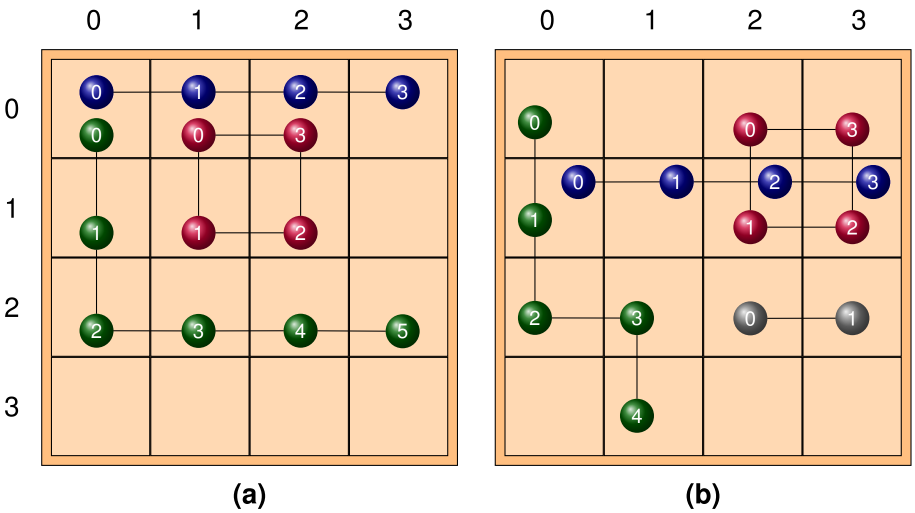

Figure 5: Two examples for n-tuples: (a) 3 n-tuples, (b) 4 n-tuples of varying length and placement.

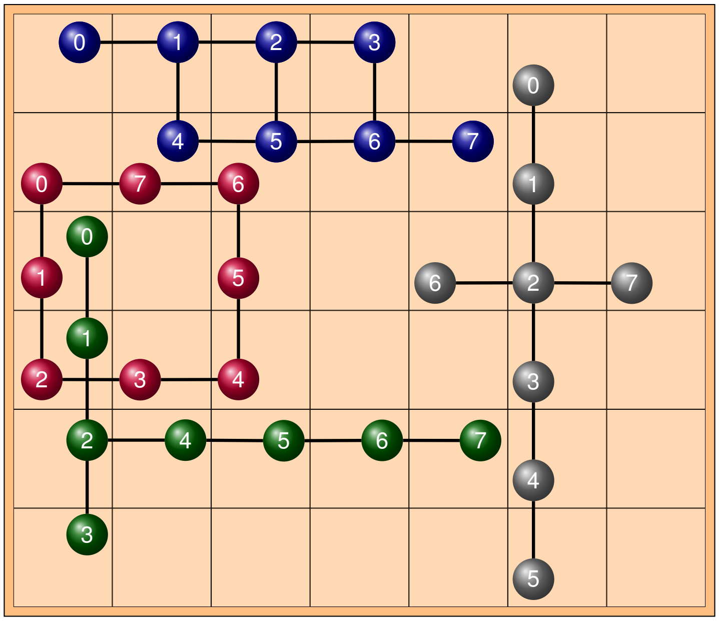

Figure 6: Another example: four 8-tuples on a 6x7 field.

Example (a) in Fig. 5 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;
}
```

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[21], 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`.

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: `{O,-,X}`. For an n-tuple of length $n = 2$ this leads to $3^2 = 9$ possible LUT entries

```
{OO, O-, OX, -O, --, -X, XO, X-, XX}
```

[21] within the same n-tuple

These LUT-entries are features. Even for a small number of n-tuples this will generate quite large number of features. For example in Fig. 5(a), if we assume 3 position values for each cell, the number of features is $3^4 + 3^4 + 3^6 = 891$, 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 ν 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

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

which is a simple a neural net without hidden layer and with a sigmoid function $\sigma(\cdot)$.[22] We compare the estimate generated by this net with the target game value V prescribed by TD-learning. A δ-rule learning step with step-size α (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, TicTacToe and 2048 have eight symmetries (4 rotations × 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

```
public BoardVector[] symmetryVectors(BoardVector boardVector, int n);
```

See Sec. 8.9 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 5.

[22] In `TDNTuple4Agt` the sigmoid function is always $\sigma = \tanh$ (see helper class `NTuple4ValueFunc`), so that $V^{(est)} \in [-1, 1]$ holds.

Table 5: 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

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 6 shows and distinguishes different methods for agent string representations.

Table 6: 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 (see Types.GUI_AGENT_LIST)
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

Table 7: Summary of all files written by GBG.

filename	directory	remark
text files		
playStats.csv	agents/<game>[/<subdir>]/csv	statistics of a 'Play' episode
multiTrain.csv	agents/<game>[/<subdir>]/csv	multi-training results
theNtuple.txt	agents	n-tuple configuration (last loading)
binary files		
*.agt.zip	agents/<game>[/<subdir>]	saved agents
*.tsr.zip	agents/<game>[/<subdir>]/TSR	tournament system results
*.gamelog	logs/<game>[/<subdir>]	log files

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.

F Appendix: Files Written by GBG

Table 7 shows and distinguishes the files written by GBG. Some remarks:

- The file `playStats.csv` is only written, if `Types.PLAYSTATS_WRITING==true`. Usually, this source code variable is set to `false` to avoid cluttering the file system with files `playStats.csv` each time a game episode is played.
- The file `theNtuple.txt` is not meant for permanent storage. It is only an intermediate print-out of a certain n-tuple configuration (perhaps a good-working one generated by random walk) and enables to copy it into the source code of a game as a fixed n-tuple mode.

G Parameter Settings for Scheiermann and Konen [2022]

In this appendix we list all parameter settings for the GBG agents used in Scheiermann and Konen [2022]. Parameters were manually tuned with two goals in mind: (a) to reach high-quality results and (b) to reach stable (robust) performance when conducting multiple training runs with different random seeds. The agents listed further down are the best-so-far agents found for the respective games (best among all agents learning from scratch by self-play).

The detailed meaning of RL parameters is explained in Konen and Bagheri [2021]:

- Algorithms 2, 5 and 7 in Konen and Bagheri [2021] explain parameters α (learning rate), γ (discount factor), ϵ (exploration rate) and output sigmoid σ (either identity or $\tanh$).

- Appendix A.3 explains our eligibility method, parameters are: eligibility trace factor λ, horizon cut c_h, eligibility trace type ET (normal) or RESET (reset on random move). If not otherwise stated, we use $\lambda = 0$ (no eligibility trace), and in this case horizon cut c_h and eligibity trace type are irrelevant. If $\lambda > 0$, their default is $c_h = 0.1$ and eligity trace type ET.
- Appendix A.5 explains our TCL method, parameters are: TC-Init (initialization constant for counters), TC transfer function (TC-id or TC-EXP), β (exponential factor in case of TC-EXP), TC accumulation type (delta or recommended weight-change)

Another branch of our algorithm is the MCTS wrapper, which can be used to wrap TD-N-tuple agents during evaluation and testing. The precise algorithm for MCTS wrapping is explained in detail in [Scheiermann and Konen, 2022, Sec. II-B].[23] The relevant parameters are:

- c_{PUCT}: relative weight for the prior probabilities of the wrapped agent in relation to the value that the wrapper estimates
- d_{max}: maximum depth of the MCTS tree, if -1: no maximum depth
- UseSoftMax: boolean, whether to use SoftMax normalization for the priors or not
- UseLastMCTS: boolean, whether to re-use the MCTS from the previous move within an episode or not

Further parameter explanations:

- Appendix C in this document explains n-tuples, parameters are: number of n-tuples, length of n-tuples, and n-tuple creation mode (fixed, random walk, random points). Appendix C explains as well symmetries.
- USESYMMETRY: whether to use symmetries or not
- LearnFromRM: whether to learn from random moves or not
- ChooseStart-01: whether to start episodes from different 1-ply start states or always from the default start state.
- E_{train}: maximum episode length during training, if -1: no maximum length.
- E_{eval}: maximum episode length during evaluation and play, if -1: no maximum length.

In the following we list the parameter settings for the four agents used in Scheiermann and Konen [2022]. All agents were trained with no MCTS inside the training loop.

If not stated otherwise, these common settings apply to all agents: LearnFromRM = false, ChooseStart-01 = false, $E_{train} = -1$, $E_{eval} = -1$. Wrapper settings during test and evaluation: MCTS wrapper with $c_{PUCT} = 1.0$, $d_{max} = -1$, UseSoftMax = true, UseLastMCTS = true.

[23] As [Scheiermann and Konen, 2022, Sec. IV-E] shows, the MCTS wrapper may be used as well during training, but it may require large computation times.

G.1 Othello 8x8

$\alpha = 0.2$, $\gamma = 1.0$, $\epsilon = 0.2$, $\epsilon_{final} = 0.1$, $\lambda = 0.5$, horizon cut $c_h = 0.1$, eligibility trace type ET, output sigmoid $\sigma = \tanh$. N-tuples: 100 7-tuples created by random walk. TCL activated with transfer function TC-EXP, TC-Init= 10^{-4}, $\beta = 2.7$ and rec-weight-change accumulation. USESYMMETRY, Choose-Start-01 and LearnFromRM active. 250,000 training episodes.

Agent filename in GBG: TCL4-100_7_250k-lam05_P4_nPly2-FAm_A.agt.zip

G.2 ConnectFour

$\alpha = 3.7$, $\gamma = 1.0$, $\epsilon = 0.1$, $\epsilon_{final} = 0.0$, $\lambda = 0.0$, output sigmoid $\sigma = \tanh$. N-tuples: FIXEDNTUPLEMODE=1 (70 specific 8-tuples). TCL activated with transfer function TC-EXP, TC-Init= 10^{-4}, $\beta = 2.7$ and rec-weight-change accumulation. USESYMMETRY and LearnFromRM active. 6,000,000 training episodes.

Agent filename in GBG: TCL-EXP-NT3-al37-lam000-6000k-epsfin0-noFA.agt.zip

G.3 Rubik's Cube

The agents shown here are for the STICKER2 representation and half-turn metric (HTM) [Konen, 2022]. USESYMMETRY, Choose-Start-01 and LearnFromRM are all inactive in the Rubik's cube case.
MCTS wrapper during test and evaluation with $c_{PUCT} = 1.0$, $d_{max} = 50.$[24]
Other parameters are:

G.3.1 2x2x2 Cube

$\alpha = 0.25$, $\gamma = 1.0$, $\epsilon = 0.0$, $\lambda = 0.0$, output sigmoid $\sigma =$ id. N-tuples: 60 7-tuples created by random walk. TCL activated with transfer function TC-id, TC-Init= 10^{-4} and rec-weight-change accumulation. 3,000,000 training episodes. Scrambling by 1,...,13 twists, $E_{train} = 16$, $E_{eval} = 50$.

Agent filename in GBG: 2x2x2_STICKER2_AT/TCL4-p13-ET16-3000k-60-7t-stub.agt.zip

G.3.2 3x3x3 Cube

$\alpha = 0.25$, $\gamma = 1.0$, $\epsilon = 0.0$, $\lambda = 0.0$, output sigmoid $\sigma =$ id. N-tuples: 120 7-tuples created by random walk. TCL activated with transfer function TC-id, TC-Init= 10^{-4} and rec-weight-change accumulation. 3,000,000 training episodes. Scrambling by 1,...,9 twists, $E_{train} = 13$, $E_{eval} = 50$.

Agent filename in GBG: 3x3x3_STICKER2_AT/TCL4-p9-ET13-3000k-120-7t-stub.agt.zip

[24] When evaluating the trained cube agents with different MCTS wrappers, we test in each case whether $c_{PUCT} = 1.0$ or 10 is better. In most cases, $c_{PUCT} = 1.0$ is better, but for (2x2x2, QTM, 800 iterations) and for (3x3x3, HTM, 100 iterations) $c_{PUCT} = 10.0$ is the better choice.

The agent filenames given above which cary '-stub' in their name are just stubs, i.e. agents that are initialized with the correct parameters but not yet trained. This is because a trained agent can require up to 80 MB disk space, which is too much for GitHub. Instead, a user of GBG may load such a stub agent, train it (takes between 10-40 min in the Rubik's cube case) and save it to local disk.

See Konen and Bagheri [2021] for more parameter settings (best-so-far agents) in other games.

References

Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8):356–363, 2019. 12

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. 19, 33

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. 6

Donald F. Beal and Martin C. Smith. Temporal coherence and prediction decay in TD learning. In Thomas Dean, editor, *Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 564–569. Morgan Kaufmann, 1999. ISBN 1-55860-613-0. 24

Charles L Bouton. Nim, a game with a complete mathematical theory. *Annals of Mathematics*, 3(1/4):35–39, 1901. 12

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016. 6

Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012. 12

R. Delorme. Edax, version 4.4, 2017. URL `https://github.com/abulmo/edax-reversi`. 12

Michael Genesereth and Michael Thielscher. General game playing. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(2):1–229, 2014. 6

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`. 6

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. 14

Wolfgang Konen. General board game playing for education and research in generic AI game learning. In Diego Perez, Sanaz Mostaghim, and Simon Lucas, editors, *Conference on Games (London)*, pages 1–8, 2019. URL https://arxiv.org/pdf/1907.06508. 5, 7, 8

Wolfgang Konen. Towards learning Rubik's cube with n-tuple-based reinforcement learning. Technical report, TH Köln (in preparation), 2022. 50

Wolfgang Konen and Samineh Bagheri. Reinforcement learning for n-player games: The importance of final adaptation. In Bogdan Filipic Massimiliano Vasile, editor, *9th International Conference on Bioinspired Optimisation Methods and Their Applications (BIOMA)*, Bruxelles, November 2020a. URL http://www.gm.fh-koeln.de/ciopwebpub/Konen20b.d/bioma20-TDNTuple.pdf. 14, 32

Wolfgang Konen and Samineh Bagheri. Final adaptation reinforcement learning for N-player games. Technical report, Research Center CIOP (Computational Intelligence, Optimization and Data Mining), 2020b. URL http://www.gm.fh-koeln.de/ciopwebpub/Konen20b_TR.d/Konen20b_TR.pdf. 14, 32

Wolfgang Konen and Samineh Bagheri. Final adaptation reinforcement learning for n-player games. *arXiv preprint arXiv:2111.14375*, 2021. 48, 51

Richard E. Korf. Multi-player alpha-beta pruning. *Artificial Intelligence*, 48(1):99–111, 1991. 12, 13, 24

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. 12, 13

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. 6

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. 6

Simon M. Lucas. Learning to play Othello with n-tuple systems. *Australian Journal of Intelligent Information Processing*, 4:1–20, 2008. 12, 19, 24, 33, 43

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. 6

Stephen McAleer, Forest Agostinelli, Alexander Shmakov, and Pierre Baldi. Solving the Rubik's cube with approximate policy iteration. In *International Conference on Learning Representations*, 2018. 12

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. 6

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. 6

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. 6

Barney Pell. A strategic METAGAME player for general chess-like games. *Computational Intelligence*, 12(1):177–198, 1996. 6

Johannes Scheiermann. AlphaZero-inspirierte KI-Agenten im General Board Game Playing, 2020. URL http://www.gm.fh-koeln.de/~konen/research/PaperPDF/BA_Scheiermann_final.pdf. Bachelor thesis, TH Köln – University of Applied Sciences. 14

Johannes Scheiermann and Wolfgang Konen. AlphaZero-inspired game learning: Faster training by using MCTS only at test time. *arXiv preprint arXiv:2204.13307*, 2022. 1, 3, 14, 48, 49

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017. 14

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998. 12

Maciej Świechowski, HyunSoo Park, Jacek Mańdziuk, and Kyung-Joong Kim. Recent advances in general game playing. *The Scientific World Journal*, 2015, 2015. 6

Michael Thielscher. A general game description language for incomplete information games. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010. 6

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. 12, 19, 33, 46

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. 19, 33, 46