e-print http://www.gm.fh-koeln.de/ciopwebpub/Kone16a.d/TR-GBG.pdf

The GBG Class Interface Tutorial (General Board Game Playing and Learning)

Wolfgang Konen

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

wolfgang.konen@th-koeln.de

November 2016)

Abstract

This technical report ...

Contents

1 Introduction

1.1 Motivation

General board game (GBG) playing and learning is a fascinating area in the intersection 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 Tic-Tac-Toe, Othello, Chess, ...), many-person games (like Settlers of Catan, ...). 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 to 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. GBG is written in Java.

1.2 Related work

TODO

Hoyle Epstein [2001]

2 Class and Interface Overview

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

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

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

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

3 Classes in Detail

3.1 Interface StateObservation

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

- returning the available actions (`getAvailableActions()`),
- advancing the state of the game with a specific action (`advance()`),

- copying the current state
- signaling end, score and winner of the game

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

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

Implementing classes: StateObserverTTT, StateObserver2048, ...

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

3.2 Interface PlayAgent and class AgentBase

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

- `getNextAction(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 starting from state `sob`.

Some more methods, e.g. setters and getters, have their defaults implemented in class **AgentBase**.[1] It might be useful to design a new agent class with the signature `... extends AgentBase implements PlayAgent`.

There is an additional method `double rewardEstimate(sob)` which has the default implementation `getScore(sob)` in **AgentBase**. This method is called when a training game is stopped prematurely because the maximum number of moves in an episode ('Episode length') is reached. The implementing class may override this and implement a more sophisticated method to estimate the final reward.

Classes implementing interface **PlayAgent**:

[1] **AgentBase** does not implement all methods of the interface PlayAgent, so it has no `... implements PlayAgent`.

- 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)[2]
- MCTSAgent: Monte-Carlo Tree Search agent
- TDAgent: general TD(λ) agent (temporal difference, reinforcement learning) with neural network value function (abstract class due to factory method `makeFeature`, see Sec. 3.3)
- TDPlayerTTT: Instantiation of TDAgent for the game TicTacToe. Uses the specific feature class FeatureTTT.
- TDPlayerTT2: (deprecated) TD(λ) agent with neural network value function for the game TicTacToe.[3]

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

3.3 Interface Feature

Some PlayAgent need a game-specific feature vector. This is for example the case for 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. To create such an **Feature** object within the general PlayAgent-code, the *factory method pattern* is used: PlayAgent defines an abstract method

```
abstract public Feature makeFeature(int featmode);
```

[2]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.

[3]This class has TicTacToe elements hard-coded and is thus less general and deprecated. Although TDAgentTTT is specific to TicTacToe, it has the member `TD_func m_net` of class `TD_Lin` or `TD_NNet` which implements the main functionality of TD(λ) for neural networks.

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

Interface Feature has the method

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

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

Implementing classes: FeatureTTT, Feature2048, ...

3.4 Some Remarks on the Game Score

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

- `PlayAgent.getScore(StateObservation so)` returns the agent's estimate of the score for the player **who has to move** in StateObservation `so`. The score 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.

- 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 will be -1 („O loses“). This seems a bit awkward at first sight, but it is the only way to guarantee in a succession of actions that the current score is always the negative of the next state's score (negamax principle). Fig. 1 shows an example.

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

  Example: A 2-player game like TicTacToe is terminal when X makes a winning move. On this terminal state O would have to move next (if it «add a TikZ picture with 3 TTT game states here»

Figure 1: A succession of states in TicTacToe: (a) is a clear loss for O, (b) is a clear win for X, so (c) should be a clear loss for O again to be consistent.

were not terminal). So the game score for this terminal state is a negative reward for player O. It turns out that TicTacToe is always terminated with either a negative reward or a tie.

- `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 by a 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.[4]

3.5 Interface GameBoard

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

- Maintaining its own StateObservation object `m_so`. This object is after construction in a default start state (e. g. empty board). The same state can be reached via `clearBoard()` or `getDefaultStartState()` as well. The associated GUI will show the default start state.
- Showing or updating the current game state (StateObservation) in the GUI and enabling / disabling the GUI elements (`updateBoard(...)`).

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

- Human interaction with the board: see Sec. 3.6.
- Returning its current StateObservation object (`getStateObs()`).
- `chooseStartState01()`: This method returns randomly one out of a set of different start states. This is useful when training an agent so that not always the same game episode is played but some variation (exploration) occurs.

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

Implementing classes: GameBoardTTT, GameBoard2048, ...

3.6 Human interaction with the board and with Arena

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

If GameBoard request an action from Arena, then its method `isActionReq()` returns `true`. This causes the selected AI to perform a move. If on the other hand a human interaction is requested, Arena issues a `setActionReq(false)` and this causes `isActionReq()` to return `false` as well. GameBoard then waits for GUI events until a user (human) action is recorded. GameBoard is responsible for checking whether the human action is legal (`isLegalAction()`).[5] If so, then GameBoard issues an `advance()`. Method `advance()` opens the possibility for invoking random elements from the game environment (e. g. adding a new tile in 2048), if necessary.

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

3.7 Abstract Class Evaluator

Class **Evaluator** evaluates the performance of a PlayAgent.

In the constructor

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

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

the argument `mode` allows derived classes to create different types of evaluators. These may test different abilities of PlayAgent. The argument `stopEval` sets the number of consecutive evaluations that the abstract method

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

has to return with `true` until the evaluator is said to reach its goal (method `goalReached()`). This is used in XArenaFunc's method `train()` as a possible condition to stop training prematurely.

Method `evalAgent()` needs to be overridden by classes derived from Evaluator.

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

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

in Arena or ArenaTrain.

3.8 Abstract Class Arena

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

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

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

Class Arena has the following functionality:

- choice of agents for each player (load)
- playing games (AI agents & humans)
- inspecting the move choices of an agent

- (TODO) logging of played games (option for later replay or analysis)
- (TODO) undo/redo possibilities
- (TODO) game balancing
- (TODO) game leagues, round-robin tournaments, ...

Derived abstract class: ArenaTrain. Derived non-abstract classes: ArenaTTT, Arena2048, ...

3.9 Abstract Class ArenaTrain

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

- specifying all parameters for an agent
- training an agent (one or multiple times)
- evaluating agents, competitions (one or multiple times)
- saving agents
- (TODO) replay memory for better training

The helper classes XArenaFuncs, XArenaButtons, XArenaMenu, XArenaTabs contain functionality needed for Arena and ArenaTrain.

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

4 Use Cases

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

As a game developer you have to implement the following three interfaces or abstract classes for your game:

- `StateObserverXYZ implements StateObservation`
- `GameBoardXYZ implements GameBoard`
- `EvaluatorXYZ extends Evaluator`

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

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

Finally you need a class with `main()` to launch ArenaTrain. You may copy and adapt the example in `LaunchTrainTTT`.

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

4.2 How to train an agent and save it

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

5 Open Issues

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

- The GUI for Arena and ArenaTrain is just a quick hack adapted from earlier programs and may need further refinement.
- The above-mentioned elements for Arena and ArenaTrain that are planned but not yet implemented:
  - (TODO) logging of played games (option for later human replay or analysis), logging of competition games
  - (TODO) undo/redo possibilities
  - (TODO) a slider during agent-agent game play to control the playing velocity. Optional game visualization during competitions as well.
  - (TODO) game balancing
  - (TODO) game leagues, round-robin tournaments, ...
  - (TODO) option to enable/disable value function display during game play
  - Refine the menu: one column for each agent, disable the not-working menu items
  - (TODO) replay memory for better training: This is the idea used by DeepMind in learning Atari video games. Played episodes are stored in a replay memory pool and used repeatedly for training.
- The extension to n-player games $(n > 2)$ is not fully functional yet. The cases $n = 1$ and $n = 2$, which are fully functional, need to be tested on a variety of 1- and 2-player games.
- The n-tuple agents developed for C4 (Connect Four) and TTT need to be ported to GBG.
- There should be a new use case which discusses what a developer has to do to design a TD(λ) agent for a new game (class Feature).
- There seems to be a slight discrepancy in quality between TDPlayerTTT (new version, class Feature) and TDPlayerTT2 (old version, features hard-coded). Clarify, if and why. And: Make factory method `makeFeature` safe

against a `featmode` which is not constructable by a certain Feature implementation.

- Allow only trained agents to be saved.
- Clarify: Is the params data flow safe, if we issue a 'play' or 'compete' for 2 agents of same type but with different parameters?

A Appendix

A.1 Interface StateObservation

```
/**
 * Class StateObservation observes the current state of the game,
 * it has utility functions for
 * <ul>
 * <li> returning the available actions (getAvailableActions()),
 * <li> advancing the state of the game with a specific action (advance()),
 * <li> copying the current state
 * <li> signaling end, score and winner of the game
 * </ul>
 *
 * @author Wolfgang Konen, TH Köln, Nov'16
 */
public interface StateObservation {

public StateObservation copy();
public String toString();
public double getGameScore();
public double getGameScore(StateObservation referingState);
public double getMinGameScore();
public double getMaxGameScore();
public void advance(ACTIONS action);
public boolean isLegalState();
public boolean isGameOver();
public Types.WINNER getGameWinner();
public ArrayList<ACTIONS> getAvailableActions();
public void setAvailableActions();
public int getNumAvailableActions();
```

```
public Types.ACTIONS getAction(int i);
public void storeBestActionInfo(ACTIONS actBest, double[] vtable);
public int getNumPlayers(); // n
public int getPlayer();     // (0,1,...,n-1)
public int getPlayerPM();   // (+1,-1) for a 2-player game
}
```

A.2 Interface GameBoard

```
/**
 * Each class implementing interface GameBoard has the board game GUI.
 *
 * It has an internal object derived from StateObservateion which
 * represents the current game state. It can be retrieved (getStateObs()),
 * reset and retrieved (getDefaultStartState()) or a random start
 * state can be chosen (@link #chooseStartState01()).
 * @author Wolfgang Konen, TH Köln, Nov'16
 */
public interface GameBoard {

public void clearBoard(boolean boardClear, boolean vClear);
public void updateBoard(StateObservation so, boolean showStoredV,
            boolean enableOccupiedCells);
public void showGameBoard(ArenaTrain ticGame);
public boolean isActionReq(); // action requested from Arena?
public void setActionReq(boolean actionReq);
public void enableInteraction(boolean enable);
public StateObservation getStateObs();
public StateObservation getDefaultStartState(); // empty-board state
public StateObservation chooseStartState01();
}
```

A.3 Interface PlayAgent

```
/**
 * The abstract interface for the game playing agents.
 *
```

```
 * @author Wolfgang Konen, TH Köln, Nov'16
 */
public interface PlayAgent {
public enum AgentState {RAW, INIT, TRAINED};

public Types.ACTIONS getNextAction(StateObservation sob, boolean random,
                     double[] vtable, boolean silent);
public double getScore(StateObservation sob);
public double rewardEstimate(StateObservation sob);
public boolean wasRandomAction(); // was last getNextAction random?
public boolean trainAgent(StateObservation so); // for one episode
public boolean trainAgent(StateObservation so,
               int epiLength);        // with episode length limit
public String printTrainStatus();
public String stringDescr();
public String getName();
public void setName(String name);
public AgentState getAgentState();
public void setAgentState(AgentState aState);
public int getMaxGameNum();
public void setMaxGameNum(int num);
public int getGameNum();
public void setGameNum(int num);
}
```

A.4 Interface Feature

```
/**
 * Interface Feature translates game states into feature vectors
 */
public interface Feature {
public double[] prepareInputVector(StateObservation so);
public String stringRepr(double[] featVec);
public int getFeatmode();
public int[] getAvailFeatmode();
}
```

A.5 Abstract class Evaluator

```
/**
 * Evaluates the performance of a PlayAgent in a game.
 */
abstract public class Evaluator {
protected PlayAgent m_PlayAgent;
protected int verbose=1;
public Evaluator(PlayAgent e_PlayAgent, int stopEval);
public Evaluator(PlayAgent e_PlayAgent, int stopEval, int verbose);
public boolean eval();
public boolean goalReached(int gameNum);
public boolean setState(boolean stateE);
public boolean getState();
public String getMsg();
public String getMsg(int gameNum);
abstract protected boolean eval_Agent();
abstract public double getLastResult();
}
```

A.6 Abstract class Arena

```
/**
 * This class contains the GUI and the task dispatcher for the game.
 * The GUI for buttons and choice boxes is in {@link XArenaButtons}.
 *
 * Run this class from the {@code main} in {@link LaunchArenaTTT})
 * for the TicTacToe game.
 *
 * @author Wolfgang Konen, TH Köln, Nov'16
 */
abstract public class Arena extends JPanel implements Runnable
{
public enum Task {PARAM, TRAIN, MULTTRN, PLAY, INSPECTV
                  , COMPETE, SWAPCMP, MULTCMP, IDLE  };
protected GameBoard gb;
public Task taskState;
public Arena();
```

```
public Arena(JFrame);
public void init();
public void run();
public void PlayGame();
public void enableButtons(boolean state);
public void setStatusMessage(String msg);
public StatusBar getStatusBar();
abstract public GameBoard makeGameBoard();
abstract public Evaluator makeEvaluator(PlayAgent pa, GameBoard gb,
                int stopEval, int mode, int verbose);
abstract public void performArenaDerivedTasks();
}
```

A.7 Abstract class ArenaTrain

```
/**
 * This class contains the GUI for the arena with train capabilities.
 * It extends the task dispatcher of Arena with method
 * performArenaDerivedTasks() which contains tasks to trigger functions
 * for agent learning, parameterization, inspection and so on.
 *
 * @author Wolfgang Konen, TH Köln, Nov'16
 */
abstract public class ArenaTrain extends Arena
{
public ArenaTrain();
public ArenaTrain(JFrame frame);
public void performArenaDerivedTasks();  // extend task dispatcher
protected void InspectGame(); // inspect agent X on game positions
}
```

References

Susan L Epstein. Learning to play expertly: A tutorial on Hoyle. *Machines that learn to play games*, pages 153–178, 2001. 4