Extended General Gaming Model

Michel Quenault and Tristan Cazenave

LIASD
Dept. Informatique
Université Paris 8, 93526, Saint-Denis, France
miq75@free.fr, cazenave@ai.univ-paris8.fr

Abstract. General Gaming is a field of research on systems that can manage various game descriptions and play them effectively. These systems, unlike specialized ones, must deal with various kinds of games and cannot rely on any kind of game algorithms designed upstream, such as DEEPER BLUE ones. This field is currently mainly restricted to complete-information board games. Given the lack of more various games in the field of General Gaming, we propose in this article a very open functional model and its tested implementation. This model is able to bring together on the same engine both card games and board games, both complete- and incomplete-information games, both deterministic and chance games.

1 Introduction

Many people are playing strategy games like Chess all around the world. Beyond the amusement they provide and their abstract forms, they surely lead to create some kind of reflection process that would be useful in everyday life. They build this way some kind of 'intelligence'. This is one of the main reason why they always are a prolific research field in computer sciences, which needs highly efficient frameworks and especially in Artificial Intelligence which try to define new ones.

Many Artificial Intelligence projects such as DEEPER BLUE are focused on specific games. This implies they use highly specific players, which use specialized data corresponding to some particular attributes of the game they are playing. Such players would certainly be efficient when confronted to their game, even to very similar ones, but they could not afford playing any other simply different games. However, since quite some time[1] a branch of Artificial Intelligence focuses on this problem and tries to develop players that would be able to play any kind of strategy games: General Gaming.

In General Gaming, as in the Metagame project[2], the objective is to compute players that would be efficient in playing various kinds of games. As we have seen, these players are not be implemented focusing on a specified game. This implies to define games engines which are able to deal with such players and to link them to the desired games. General Gaming's objective is to afford playing a maximum of different games, but card games, for example, are so much different from strategy games that nowadays, no general gaming project integrates them with strategy games. General Gaming engines focus only on strategy games, which is already a large and interesting field.

General game engines already exist. The General Game Playing Project from the Stanford University Computer Science Department[3] is a good example. Using games rules defined as logic programs, it makes play worldwide computer players on various strategy games, allowing players to know the game rules and to adapt themselves to be more efficient on a specific rule. Some commercial engines exist too, such as ZILLIONS OF GAMES[4], which has a specific functional language to describe a game's rule. Using a proprietary engine, he creates the game's engine and allows human players to play against their own proprietary computer player. A huge collection of games is already downloadable and is extended day by day by the users. However, these two projects have some limitations and use structures that restrict their usage to strategy games. Some of these restrictions are: no incomplete-information, no or very few chance and, for the second one, no way to create efficient connection games like Hex.

Based on this observation we have designed and implemented an engine based on three objectives. The principal one is to allow players to confront each other on many game kinds, including strategy games but also card games and board games such as Monopoly. Our second point is to give to players of various kinds the ability to simulate sequences of moves. The last but not the least point is to make such games and players easy to define. Such an engine is useful to compare players' efficiency on various kinds of games.

However, allowing players to confront each other on many games has some limits. We planned to integrate as many board games as possible, but computers are limited. We intend to manage finite strategy games such as Chess, card games, board games such as Monopoly and domino games. For different reasons, we limit our games to non-continuous boards, no query/answers games, no speech or dispute games, and no ultra-rich game universes as for role-playing games. Some games like wargames or deck collection games (such as Magic the Gathering) are left aside in the current version, but our structure intends to be easily adaptable to them.

This article presents the general gaming model we have defined. First the way we define games, then the way we connect players to games and eventually the main engine process. We conclude with a short discussion.

2 Games Descriptions

Any game is composed by exactly two things: A set of **Equipments**, with specific properties (boards, pieces, cards, dices) and the **Rule** of the game, which describes how players interact with components. Equipments which may seem to be very different can often be unified together. Here we present an organization of all the components we need to use to play a game, then we define what are game rules and eventually we present a simple rule as it is written in our engine.

2.1 Equipments

A strategy game often uses a board and some pieces. The board is an **Area** composed of a list of independent positions where the pieces can take place. These **Positions** are themselves relatives, they form a graph where nodes are positions and arcs are **Directions**. The pieces are **Elements** with different specifications. They are organized into **Assortments** of pieces, sometimes with many identical pieces, and sometimes no two pieces are the same. The Equipments we use here are Areas, Positions, Directions, Elements, and Assortments. Areas are graphs of Positions and Directions, Assortments are composed of Elements.

A card game often uses a table with some defined placements and cards. It also uses some abstract Equipments such as players' hands which are sets of cards. The table is a kind of Area, generally but not systematically without Directions defined. The cards are Elements, organized into Assortments too. The abstract sets of cards are represented on tables Areas as Positions because it makes easier the representation of the game and the human interface. This implies that Positions can be occupied with multiple Elements, but this is already the case in some strategy games, so it implies no modification in our Equipments list. Moreover, we use exactly the same Equipments to define both strategy and card games.

However, some other Equipments are necessary to completely cover the field of board games: **Dices** as chance generators and **Score** markers to register score. At last, we need a **Player** which represents one of the active player in our system and a **Table** which is the container of all other Equipments. With all these Equipments, we could define almost all computable board games, if we specify some restrictions. Here is a recapitulated view of these Equipments with few spotted remarks:

1. Area: A picture associated with a Position graph.
   – Area, Direction and Positions define the board or the card table.
   – Almost all games have at least one Area, but this number is totally free.
2. Position: Node of Area's graph.
3. Direction: Oriented and labeled arc of Area's graph.
   – Positions could be empty, occupied with one Element, or occupied with an ordered list of Elements.
4. Assortment: A list of Elements used in the game. This could represent a card deck, or a stock of go stones.
   – Almost all games have at least one Assortment, but this number is totally free.
5. Element: These are cards, pieces, stones, etc.
   – Elements must be able to receive any kind of attribute, with any possible value. This allows to define cards (with colors, values, ranks, etc.) or Chess pieces (with type, cost, etc.). Actually, this restriction is extended to all the Equipments in the game, for easiness in the rule definition process.
6. Dice: A specific equipment to generate chance.
7. Score: A specific equipment to keep score data as in most card games.
8. Player: One more specific equipment representing a player in the game.
9. Table: This last equipment is a container of all other equipments in play.

All of these Equipments possess methods that return other related Equipments. This way the engine and the Rule can navigate through them ad libitum.

2.2 Rule

The second thing defining a game is **Rule**. First it defines the number of players. Then it defines a graph of successions of player's turns[1]. Nodes of this graph are play stages

[1] Many strategy two players games simply alternate the two players roles, but some complex games like traditional Mah-Jong needs all the potential of graphs.

where one or more players may choose to execute Actions. Arcs are brace of players and possible Actions.

Then the rule defines the initial state of the Equipments[2] and final conditions with associated winners. These parts of the rules need the use of a method call. The last thing defined by the rule is the explanation of legal Actions. These legal Actions are the link between initial state and final states described with final conditions. Here again, the use of a method call to create the list of legal Actions is coerced.

Method calls are needed to define initial states, Actions and final conditions. These methods must be defined in the rules objects and will always have as single argument a Table. This argument refers to all Equipments defined in the Rule and used in the play. The method must return new built objects corresponding to atomic **Actions** that alter the Table and correspond to players moves. These Actions could be any ordered combination of any number of Actions to ensure complex Actions ability to be defined. The initial state method must return exactly one Action. The final condition method return nothing or one special end game Action.The moves methods must return the list of actual legal Actions for the current stage of the play.

The possible Actions and their effects are:

1. Pass: Nothing to do.
2. Move: Move any number of Elements from a Position or a Assortment to another.
3. Score: Mark points in a Score Equipment.
4. FinishPlay: Declare some winners, some losers or a draw game.
5. Set: Add an attribute to any Equipment.
6. Del: Removes an attribute to any Equipment. Access to these attributes is ensured by methods in Equipments. Here are just defined Actions that alter the Equipments.
7. Distribute: Distributes all Elements from a Position or a Assortment to a list of Positions or Assortments and affects them a new owner relative to the Positions.
8. Sort: Sort the list of Elements in a Position or a Assortment.
9. Shuffle: Shuffle the list of Elements in a Position or a Assortment.
10. Roll: Randomly changes the value of a Dice list.

2.3 Example

Algorithm 1 is an example of the full definition file for a rule. The game is basic Tic-tac-toe. The language used is python. The `board` and `turns` values respectively describe the board graph and the turn order arc. Inline tools are provided to easily generate these lists but the use of such lists ensures that any board or turn order graph can be defined even when automatic method fails.

The `defineEquipments` method selects the Equipments used in the game. The Assortment last argument is the Elements layout. The two last methods define the final conditions and the legal Actions.

Notice the way the play data are accessed: `table.getPositions()`. Such as in `table.getElement(player=table.getCurrentPlayer())`, Equipments methods may have arguments to restrict returned Equipments on any attribute values.

[2] Equipments are indeed first defined in the rule too, so the rule is enough to fully define a game.

This short page is enough to create a complete game with our engine. The complex parts of code of Algorithm 1 are detailed in Appendix A.

```
from rule import *

board=[ ('A1', (60, 60), [('H', 'B1'), ('V', 'A2'), ('B', 'B2')]),
        ('B1', (150, 60), [('H', 'C1'), ('V', 'B2'), ('B', 'C2')]),
        ('C1', (240, 60), [('V', 'C2')]),
        ...]
turns=[ ('wait Cross', True, True, [('Cross', TicTacToe.move, 'wait Circle')]),
        ('wait Circle', True, True, [('Circle', TicTacToe.move, 'wait Cross')]) ]
pawns=[ ('X', 'Cross', 'images/cross.gif'),
         ('O', 'Circle', 'images/circle.gif')]

class TicTacToe (Rule):
    def __init__(self):
        self.name = 'Tic Tac Toe'
        self.players = ['Cross', 'Circle']
        self.turns = turns

    def defineEquipments(self, table):
        table.addEquipment(Area('images/ttt_board.gif', board))
        table.addEquipment(Assortment(pawns, ['name', 'player', 'image']))

    def playResult(self, table):
        action=table.getLastPlayerAction()
        if action!=None and table.hasNewLine([move.getPositionTo()], 3,
                    elementRestraints={'player': table.getCurrentPlayer()}):
            return FinishPlay(table, table.getCurrentPlayer())

    def move(self, table):
        res=[ ]
        for pos in table.getPositions():
            if pos.isEmpty():
                res.append(Move(table.getAssortment(), pos,
                        table.getElement(player=table.getCurrentPlayer())))
        return res
```

ALG. 1: Tic-Tac-Toe Class.

3 Players Descriptions

Here is a quick list of possible players we have started to develop and integrate in our general gaming model. All these methods can be easily combined:

1. Alpha-Beta, Min-Max, tree exploration based,

2. Monte-Carlo methods,
3. Neural Networks,
4. Genetic Algorithms and Genetic Programming.

Our model is based on functional rule description and step by step play unfolding. At any time in the game when some player can make an Action, this player is called with a list of each player possible Actions computed following the rule definition on the players variant of the Table. The player has to send back the Action he prefers. He may also launch a simulation of an Action and his consequences on the play. He could pursue this process anytime he wants and explore the play tree. For incomplete-information games the player sees all unknown information trough a randomly generated possible Table state. This unknown part of the play can be shuffled anytime to simulate another possible situation.

As for games, there are some limits to our application and the player that we could connect to it. One is that our model is based on functional rule description and step by step play deployment. This implies that we do not provide tools for analyzing game rules before the play begins. Actually no access to these data is provided yet and it can be easily improved, but it is complex enough to implement this model without querying about pre rule analyzer players. The other point is that our players are highly integrated in our game engine and that the engine is in charge of generating possible Actions for the player, even in simulations. Detaching players to try to solve this problem is one of the planned evolution of our engine.

Our players are connected to our engine with some few methods:

1. `doAction`: Play the selected player's Action.
2. `doSimulateAction`: Play any legal Action and compute the next possible Actions for all players, modify only the players specific Table.
3. `undoSimulateAction`: Undo the last simulated Action and restore specific Table and next possible Actions.
4. `getChoices`: Return the list of all possible players Actions corresponding to the current simulation or play.
5. `getEval`: Read the game engine's result on current simulation or play.

4 Engine Description

In this section we will focus on our game engine. First we will explain its global behavior, then we will present how we have implemented it and how we want to use and improve it later.

4.1 Main Loop of the Engine

The Algorithm 2 presents the main tasks of the Engine. After having initialized the rule object, the engine uses its attributes to define the different parts of the play: the turn order graph and the Tables related Equipments. The turn order graph leads the main course of the events by defining the possible players and the possible Actions during each play step. In order to do that, the engine needs to apply the rules Action creation method on each player's Tables (lines 7 and 8).

Then the engine calls the players to let them define the Action they want to realize (lines 9 and 10). During this phase, each player can use its own Table to manage Action simulations. Then the engine deals with different priority kind of rules to select the next legal Action in the players answers (Line 11). There are two possible ways to select the Action, one is to choose the faster player (this allows to compare quick-thinking players to deep thought ones on fast based games). The other way is to describe priority rules in the rule file as for in traditional Mah-Jong.

In incomplete-information games the player has in his Table one possible distribution of the Elements he doesn't knows. During the creation of Actions (relatives to player's Tables but equivalents to the engine selected one)(line 12) there is a coherency engine which modifies any player's Table[3] so that these Tables correspond to the desired Action.

Then, the program loops until the engine detects a FinishPlay Action returned by `Rule.playResult()` (line 6).

1 Create $rule$ using $rule.\_\_init\_\_()$
2 Create $turn\_graph$ using $rule.turns$ and select start node
3 Create $Table[engine]$ using $rule.defineEquipments$
4 For each $player$ in $rule.players$:
5   Create $Table[player]$ using $rule.defineEquipments$
6 While $rule.playResult(Table[engine]) == None$:
7   For each arc in $turn\_graph.current\_node$:
8   Create $possible\_actions[player]$ using $(Table[arc.player], arc.method)$
9   For each $player$ having $possible\_actions$:
10   Select $favorite\_action[player]$ using $Table[player]$
11   Select one player's $favorite\_action$
12   Recreate $selected\_favorite\_action$ on all $Tables$
13   Apply $selected\_favorite\_action$ on all $Tables$
14   Update $turn\_graph.current\_node$

ALG. 2: Engine main loop.

4.2 Implementation of the Engine

At the moment, the engine implementation is in progress in the Python language. Three games are defined: A Tic-Tac-Toe which is described by Algorithm 1, a Go-Moku which use a very similar file and a Chinese-Poker which is a four players card game using poker combinations. One player is implemented too: A Min-Max player with op-

[3] e.g. Before George plays a $3\spadesuit$, Jane was thinking he had in his hands only a $2\diamondsuit$ and a $7\heartsuit$. When George play his $3\spadesuit$, Jane's knowledge on Georges hands would change this way: $3\spadesuit$ anywhere guessed position would be swapped with either the $2\diamondsuit$ or the $7\heartsuit$ ones in Jane's idea of George's hands. Then George could play this card.

tional Alpha-Beta cut. A Monte-Carlo player is on the point of being added, as soon as multiple Table control is fully realized[4].

All Equipments are already defined and created, except the Dice. All Equipments are linked to some others ones. The state of these relations represents the state of the table during the play. Rule can use many Equipments methods to test this state. For instance, these are few of the Position Equipment methods to illustrate the principle:

1. `getArea()`: Returns the Area which the Position depends on.
2. `getElements(restraints)`: Returns the list of Elements played on the Position. Restraint is an optional dictionary of attributes which filters the returned Elements.
3. `getDirections(name, orientation)`: Returns the possible Directions that links this Position with others on its Area.
4. `getOppositeDirections(direction)`: Returns the opposite Directions (if any) to the one given as argument.

All Actions are already defined too. They have two main methods allowing the engine to really alter Tables. One is `doAction()` which performs the desired action and the other is `undoAction()` which restores the Table in its previous state. This way the engine manages players simulations. Actions have the ability to add themselves to each others, so `Move()+Set()` is seen by the engine as only one Action. There are many other Actions methods used by the engine (to manage graphic interface for instance) we choose not to describe here.

The engine uses a few more classes to implement the model: Graphic interface, engine which manages the main loop, and graphs are used too. Some other no fundamental tools are provided: It is possible to define options in game rules (such as exactly or at least five pawns on Go-Moku). These options are defined in the `rule.__init__()` method and must be chosen before the engine starts to play. A tool is provided to automatically generate the Area's graph definition lists, based on dimensions. Some complex non required methods are provided (`Table.hasNewLine()` for instance) to make easier rule creation.

Despite of its early development stage, this engine can already manage both complete-information strategy games such as Chess and both incomplete-information card chance based games such as traditional Mah-Jong. Today, as far as we knew, there is no another engine with similar proficiency.

4.3 The Future of the Engine

There are many possible uses or upgrades to this engine. We present now the main ones.

The first use is for Artificial Intelligence benchmarking. With the capability to create very different kinds of games, based on opposite moves process or goals and the capability of develop various General Gaming players, this engine would be useful to compare their efficiency, their robustness and even their creativeness facing various

[4] Each player has its own Table, which corresponds to what he knows of the play. These Tables are highly cross-referenced to each others to manage incomplete-information games coherence engine. This part is in debugging stage.

problems which have often already been classified[5]. The model was first developed in this perspective.

Another evident possible use is for entertainment of many players around the world, connected to many possible games against efficient computer engines. The easiness in the game's rule creation would probably lead to such an amazing collection of games than for ZILLIONS OF GAMES if we bother to distribute this engine as they did.

Furthermore, this engine is quite young and it would be instructive to develop it in some new ways, in extending the range of possible games definition, or in players interface. Some evolutions concerning games could be the integration of collecting games, such as wargames or card collecting games, where the players must first define their army or their deck following specific rules before confronting other players with their owns. Another game interface evolution could be the management of continuous boards with some geometric tools in place of Areas and their Positions lists.

For the engine upgrading, it would be pleasant to tear apart the players and the engine, in order to allow engines players to perform theirs owns play explorations. This would lead to a more open engine players system, with capability of pre rule analysis.

There are many much more ways to improve this system and not enough room here to describe them all.

5 Discussions

Before concluding this article, we suggest some short discussion about our engine in the form of short queries with their answers.

– Is this model better than Stanford ones?
- No, it is not better, it is different. Stanford general game playing model uses rules in logic form, is limited to simple strategy games, and allows players to analyze rules. Our model is driven by the intention of playing easily almost any game and our players are restricted to choose the best Action in a possible tree of Actions. The two model are completing each other.
– Isn't the model too heavy?
- No, the model isn't heavy. It's a relatively light model to cover the large field of possible games. However, as the engine must compute all players simulations trees, it is quite long to play a game. This is one of the reasons why one of the next upgrade would probably be the full players parting from the engine.
– Is it really interesting to test Monte-Carlo methods on complete-information games or Alpha-Beta methods on chance incomplete-information games?
- Yes, good results have been obtained on go with Monte-Carlo players[6].

6 Conclusion

Nowadays, computers are very effective in most board games. The issue of years of research in such fields as Computer Science and Artificial Intelligence is that computers are capable to defeat the best humans players in almost all games (with some exceptions nevertheless). This shows how the advancement of theses sciences are awesome. But all these engines are specifics to their games, and don't reflect even a part of the human mind which is able to be (almost) good in any game, with the same mind.

So, it is the next step to explore the huge field of general solving methods as general gaming tries to address. We have done one more step by creating, implementing and testing a new model which is the first to allow the use of such various games as strategy, card and board games. Furthermore, we have opened the way for collecting games.

This way only, Computer Science and Artificial Intelligence will continue on their march to maybe beat, one day, human mind not because they are faster and more robust systems, but because they are more malleable and adaptive ones.

A Algorithm 1 Code Explication

Some complex calls in Algorithm 1 are detailed here:

- Line 11: `Area` is an Equipment. The arguments are the board picture and a list of positions data. The position layout is (name, coordinate, list of (direction name, direction goal)).
- Line 12: `Assortment` is an Equipment. The arguments are a list of pieces data and the corresponding layout. Some attributes (such as image) must been defined in the layout.
- Line 14: `Table.getLastPlayerAction()` returns the previous move in the game. This test is needed to control that we are not before the first move.
- Line 15: `Table.hasNewLine()` returns a boolean defining if a line is detected into an Area. Arguments are the list of positions that may be in the line[5], the size of the line, and some restraints which must be checked for one Element at each Position on the line. Here the restraint is the name of the player that owns the Element.
- Line 16: `FinishPlay` is an Action which defines the winner, which here is the current player.
- Line 21: `Move` is an Action. Arguments are the source, the target and the Elements moved. Here, we move one Element (returned by `table.getElement()`) from Table's Assortment to the Table's search current Position (line 19).

References

1. Pitrat, J.: Realization of a general game-playing program. In: IFIP Congress (2). (1968) 1570–1574
2. Pell, B.: A strategic metagame player for general chesslike games. In: AAAI. (1994) 1378–1385
3. Genesereth, M.R., Love, N., Pell, B.: General game playing: Overview of the aaai competition. AI Magazine **26**(2) (2005) 62–72
4. Lefler, M., Mallett, J.: Zillions of games. Commercial website http://www.zillions-of-games.com/index.html.
5. Boutin, M.: Le Livre des Jeux de pions. Livres de jeux. Éditions Bornemann (april 1999)
6. Bouzy, B., Helmstetter, B.: Monte-carlo go developments. In van den Herik, H.J., Iida, H., Heinz, E.A., eds.: ACG. Volume 263 of IFIP., Kluwer (2003) 159–174

[5] Typically the last played Positions, to avoid useless searches on all Area's Positions.