Implementation of a simulation-based General Game Playing agent

Vincent Menger
15 ECTS

January 16, 2012

Contents

Introduction

Computers are capable of playing many games these days, most well-known games can be played with or against a machine. These machines are often specifically designed to play one game very well, one of the first and most renowned examples of such a machine is IBM's Deep Blue that managed to defeat Chess Grandmaster Kasparov in 1997. However a noteworthy achievement that was at the time, one could remark that Deep Blue did not play Checkers or Go at all, it was in fact not able to serve as an opponent for any other game than Chess. Where a human could learn to play many games when given the game description and rules, a computer is usually unable to do so.

The aim of General Game Playing is to construct a system that is able to play a previously unknown game when its presented with a formal definition of the rules. Such agent could play Checkers, Chess, Go, and many more games. Human intervention between receiving the game description and playing the game is not possible, therefore such a system must be able to reason in a very abstract way. It cannot utilize any game-specific algorithms or exploit game-specific properties.

Games have been a topic of research in Artificial Intelligence nearly since the field existed, and General Game Playing is a great way to apply Artificial Intelligence. A system that accepts a game description and can play the game is a clear example of what one would consider to be an intelligent system and is therefore very relevant for Artificial Intelligence. From the moment I first learned about General Game Playing I especially liked the meta aspect of the topic, and implementing an agent seemed like an interesting thesis project. I have attempted to research what elements are needed for such an agent, how to successfully implement these elements, and what the theory behind General Game Playing is. This thesis is a report on my research.

In the first part of this thesis we will review what exactly the possibilities and limitations of General Game Playing are. The second part will be about how a game can be described in a formal way, and in the third part we will discuss different approaches to playing the game. The fourth and part will be about my own attempt at an implementation, which is loosely based on CadiaPlayer, the 2007 and 2008 General Game Playing world champion. Finally we will briefly discuss the results of my implementation in the fifth section.

1 General Game Playing

For many games computer software exists that can play the game against or with a human player. At first glance those machines may seem excellent examples of intelligent software, but in a way their contribution to AI is limited. Most of the intelligent work has been done by the programmer, by selecting a suitable algorithm to play the game or finding adequate heuristics to evaluate game states. The general case of 'playing games' is far more interesting to the field of AI and more challenging to computer programmers than the specific case of playing some game. This observation inspired the concept of General Game Playing (commonly abbreviated as GGP). How can one design a system that is able to play more than one game successfully, without human intervention?

An early attempt at creating such a system was called Metagame [1] and was developed by Barney Pell in 1994. The system focused on playing certain classes of games, mainly chess-like games that were randomly generated. It analyzed the rules of the game and attempted to derive an evaluation function on its own [2]. A general search engine was used to find the best moves to play, using the minimax algorithm combined with $\alpha\beta$-pruning and iterative deepening. We will discuss the exact workings of these algorithms later. Although it did not learn from experience, it did fairly well against human players.

A more recent example dates from 1998 and is called Zillions of Games [3], a commercial platform that allowed users to create their own new games and play them against their computer. It uses brute force search on the game tree, which is guaranteed to find an optimal solution for simple games, but uses way too much time when playing very complex games. Because it is primarily designed for playing board games it is still somewhat limited in the context of GGP.

An important addition to the field of research is the annual GGP Competition that is held at the AAAI conference ever since 2005. It is open for participation by anyone and to stimulate research it awards a $10,000 price to the winning entrant. Its standards are specified in [4] and [5]. As this is the most widely used standard for GGP these days, I have limited my research to those standards as well.

1.1 First constraints

Designing a system that is capable of playing literally every game seems a bit too ambitious. In order to demarcate what games our agent should be able to play, we introduce some constraints. A GGP system should be able to play every game that is:

- Finite – the number of states in a game must be finite
- Discrete – the use of continuous variables is not supported
- Deterministic – random elements such as dice or shuffled decks of cards are not allowed
- Of complete information – all game information must be available to every player at any time

This quite drastically reduces the set of games our system should accept, as video games, most card games, and even some board games will no longer be considered.

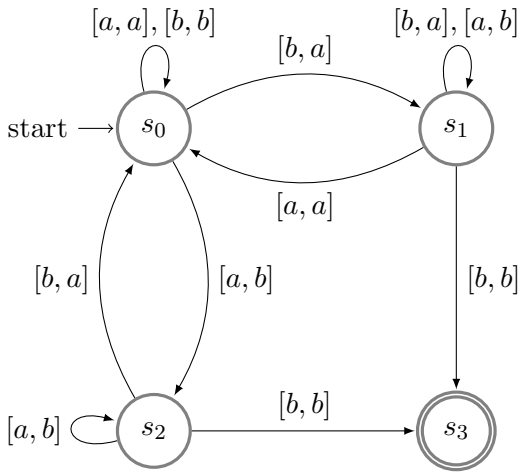
1.2 Formal description of a game

Genesereth et al proposed a formal definition of a game in [4], after slight modification of their definition we define a game G as an 8-tuple $G = (P, S, s_0, S_f, I, L, u, g)$, where:

P	A set $\{p_1, ..., p_n\}$ of n players
S	A set of states
s_0	The initial state, $s_0 \in S$
S_f	A set of final states, $S_f \subseteq S$
I	A set of actions for each player $\{I_1, ..., I_n\}$
L	A set of legal actions for each player $\{L_1, ..., L_n\}$ and each $L_i \in I_i \times S$
$u : (S \times L_1 \times ... \times L_n) \to S$	An update function, mapping moves and a state to another state
$g : (S_f \times P) \to [0..100]$	A goal function, mapping a terminal state and a player to a goal value

A game starts in state s_0, every player then picks a move from their set of legal moves for that particular state. The update function provides us with the next state. This loop will be repeated until a terminal state is encountered, in which every player receives its goal value. The player with the highest goal value is the winner of the game. It is possible for a game to end in a tie, when multiple players receive the same highest goal value. GGP is not necessarily limited to multiplayer games, it also allows singleplayer games like for instance mazes or puzzles.

The game can be represented by a game graph, which may for some game look like this:



Every node of the graph corresponds to a game state, and we can transition from one state to another through the edge corresponding to the selected moves. For instance, if player 1 plays b, a and b, and player 2 plays a, a, b the consecutive game states will be s_0, s_1, s_1 and s_3.

One limitation of this description is that both players must pick their moves simultaneously, while in many games players take turns. To solve this problem the `noop` ('no operation') move is introduced. In every game state there is one player that has a usual set of moves, whereas the other players can only play their `noop` move. `Noop` is not supposed to alter the game in any way, it is just a tool to model games where players take

turns.

As most people are familiar with the game of Tic Tac Toe, throughout this thesis the game will serve as a helpful way to make things less abstract. A state in Tic Tac Toe is a 3x3 grid with each cell either empty, with an x or marked with an o. The initial state is the empty grid, and a final state is either one that has no blanks left, or one that has a line of x's or o's. The moves are mark cell (1,1) through mark cell (3,3) and noop, and the legal moves are mark a blank cell when it's the players turn, and noop when it's the other players turn. The update function provides us with a new state, in which the cell the player has selected to mark will be marked with an x or o. The goal function finally might for instance be 100 for a player that made a line and 0 for the other player in that case, and 50 for both players when the grid is full and no player succeeded in making a line.

1.3 Additional constraints

To make the research somewhat more challenging, the GGP agent has to be able to come up with their choice for a move within a certain amount of time. Together with the description of the rules the agent receives both a start clock and a play clock, respectively the number of seconds before the game begins and the number of seconds the agent may use to select a move. The start clock can be used to parse the rules and possibly do some precomputations.

One may come up with many different games that satisfy the formal description of a game, not every game however is equally interesting for our research. Therefore we will only consider games that are well-formed:

(Def) A game is said to *terminate* if every infinite sequence of states from the initial states reaches a terminal state after a finite number of steps. A game is said to be *playable* if every player has at least one available move in every non-terminal state. A game is said to be *weakly winnable* if every player can reach a winning state through a sequence of joint moves, and *strongly winnable* if every player can reach a winning state through a sequence of moves independent of the other player's action. A game is said to be *well-formed* if it terminates, it is playable and it is either weakly or strongly winnable [5].

Although the formal definition is a sound way of describing a game, it is usually not a very efficient way. The game of Tic Tac Toe for example has $9! = 362880$ states, and the update function is bound to have more elements still. A far more common way to describe a game is using the rules and structure that describe a game. In the next chapter we will discuss Game Description Language, a way to formalize game rules.

2 Game Description Language

As we have seen in the first chapter, a formal definition of a game is a useful but inefficient way to describe games. When one would describe a game in real life, one would for instance explain how many players participate, what the board or setting looks like, and what the rules are. Game Description Language (commonly abbreviated as GDL) is intended to formalize this type of description. To do so it uses a subset of first order logic. Specifically, it uses constants, relations, and logical connectives. The logical sentences that define a game are usually stored in a single file called a rule sheet.

2.1 Facts

In GDL, a propositional fact is either:

- An object constant `a`, `b`, `c`
- A variable, marked with a question mark `?x`, `?player`, `?y`
- A relation `f(a)`, `g(?x,b)`, `f(g(?x))`, `g`

In the game of Tic Tac Toe we have a 3x3 grid that consists of cells. Each cell has three properties: a coordinate (x, y) in the grid and a sign that marks it either 'b' (blank), 'x' or 'o'. To represent a cell we therefore use a ternary relation `cell`: the facts `cell(1,1,b)` and `cell(3,3,x)` respectively denote that the bottom left cell is empty and that the top right cell is marked with 'x'.

There are three types of facts in GDL that have a special meaning.

2.1.1 Roles

The roles (i.e. the players) are specified with the role relation, for example: `role(x)`, `role(o)`, `role(alice)`, `role(bob)`. A game can have one role (singleplayer) or multiple roles (multiplayer).

2.1.2 Init

The `init` relation specifies what the initial state s_0 looks like. The facts `init(a)`, `init(control(x))`, `init(cell(1,1,b))` respectively indicate that `a`, `control(x)` and `cell(1,1,b)` hold in the initial state. A game state is exactly defined by its facts, that is to say two game states are equal if and only if they comprise the same facts.

2.1.3 Does

The `does` relation cannot be used in the description of a game, but is only added to the known facts when a player has picked a move. For instance, when player `x` has selected the action `mark(1,1)`, the fact `does(x, mark(1,1))` will be asserted in order to derive how this will alter the game state.

2.2 Rules

From the facts that hold in a state we can derive certain information, for instance whether the state is terminal, what the legal moves are for the players, and what the goal values are. How this information can be derived is defined in the game rules. A rule has the form $h \Leftarrow b_1 \wedge ... \wedge b_n$, where h is a fact that will hold when all b_i hold.

2.3 Special relations in the rule body

The following relations have special functions and can only occur as a body literal:

`true`	`true(x)` holds if `x` is a fact in the current state
`not`	`not(x)` holds if `x` does not hold in the current state
`distinct`	`distinct(x,y)` is true if `x` and `y` are not equal
`or`	`or(x,y)` holds if either `x` or `y` holds

2.4 Special relations in the rule head

These relations can only occur in a rule head, meaning they can only be derived from other facts in a state.

`legal`	`legal(p,m)` indicates it is legal for player `p` to play move `m`
`goal`	`goal(p,v)` indicates player `p` receives goal value `v`
`terminal`	nullary relation that indicates whether a state is terminal
`next`	`next(f)` indicates fact `f` holds in the next state

The next relation tells us what facts are true in the next state. This needs to be asserted explicitly, as facts that hold in a state are not automatically retained. Additionally GDL allows introduction of any auxiliary relation that is needed to describe the game, as long as it does not have the name of one of the special relations mentioned above. In order to generalize rules need to use variables. The following examples are all valid instances of rules:

```
f(?x) <= g(?x)
goal(x, 25) <= true(cell(?x,?y,b))
next(?x) <= true(cell(?x,?y,b)) ∧ not(g(?x, ?y)) ∧ distinct(?x,?y)
open <= cell(?x, ?y, b)
```

Whether these implications are valid or not naturally depends on the facts that hold in the current game state. Suppose we have the following rule:

```
row(?x, ?player) <=
      true(cell(?x, 1, ?player))
      true(cell(?x, 2, ?player))
      true(cell(?x, 3, ?player))
```

and the following facts:

```
cell(2,1,x)
cell(2,2,x)
cell(2,3,x)
```

We can derive `?x`=`2` and `?player`=`x`, and can then add `row(2, x)` to our database of facts.

2.5 Tic Tac Toe example

Once again, let's look at the Tic Tac Toe example to make things a little less abstract. The example has been taken from [5].

2.5.1 Roles

Tic Tac Toe is played by two players, one of them is x and the other one is o.

```
role(x)
role(o)
```

2.5.2 Init

In the initial state, all cells are marked with a 'b' for 'blank'. Player `x` is in control, meaning it will mark a cell first.

```
init(cell(1,1,b))
init(cell(1,2,b))
init(cell(1,3,b))
init(cell(2,1,b))
init(cell(2,2,b))
init(cell(2,3,b))
init(cell(3,1,b))
init(cell(3,2,b))
init(cell(3,3,b))
init(control(x))
```

2.5.3 Legal

If a player is in control, it may mark a cell that is currently marked with a 'b'. If a player is not in control, its only legal move is the `noop` move that we use to model games where players do not pick a move simultaneously.

```
legal(?player mark(?x ?y)) <=
      true(cell(?x, ?y, b))
      true(control(?player))

legal(x, noop) <= true(control(o))
legal(o, noop) <= true(control(x))
```

2.5.4 Goal

A player scores 100 if the player managed to make a line, in that case the other player automatically scores 0. When there are no blank cells left but no player managed to make a line both players score 50 and the game ends in a tie.

```
goal(?player, 100) <=
      line(?player)

goal(?player, 50) <=
      not(line(x))
      not(line(o))
      not(open)

goal(?player, 0) <=
```

```
      line(?player2)
      distinct(?player1, ?player2)
```

2.5.5 Terminal

The game ends when a line is formed or when the game is no longer open (i.e. no blank cells exist).

```
terminal <= line(?player)
terminal <= not(open)
```

2.5.6 Next

The first rule specifies that a cell that is being marked by a player will in the next state be labeled with a 'x' or an 'o'. The second rule specifies that cells that are not marked will remain blank.

```
next(cell(?x, ?y, ?player)) <=
      does(?player, mark(?x, ?y))

next(cell(?x, ?y, ?mark)) <=
      true(cell(?x, ?y, ?mark)
      does(?player, mark(?m, ?n))
      distinctCell(?x, ?y, ?m, ?n)
```

These rules specify that the control alternates.

```
next(control(x)) <= true(control(o))
next(control(o)) <= true(control(x))
```

2.5.7 Auxiliary

Additional relations are needed to complete the rule sheet. For instance, we would like to define what cells must be marked to obtain a line, when the game is open, and when two cells are distinct.

```
row(?x, ?player) <=
      true(cell(?x, 1, ?player))
      true(cell(?x, 2, ?player))
      true(cell(?x, 3, ?player))

column(?x, ?player) <=
      true(cell(1, ?x, ?player))
      true(cell(2, ?x, ?player))
      true(cell(3, ?x, ?player))

diagonal(?player) <=
      true(cell(1, 1, ?player))
      true(cell(2, 2, ?player))
      true(cell(3, 3, ?player))
```

```
diagonal(?player) <=
      true(cell(1, 3, ?player))
      true(cell(2, 2, ?player))
      true(cell(3, 1, ?player))

line(?player) <= row(?x, ?player)
line(?player) <= column(?x, ?player)
line(?player) <= diagonal(?player)

open <= true(cell(?x, ?y, b))

distinctCell(?x, ?y, ?m, ?n) <= distinct(?x, ?m)
distinctCell(?x, ?y, ?m, ?n) <= distinct(?y, ?n)
```

3 Game Strategies

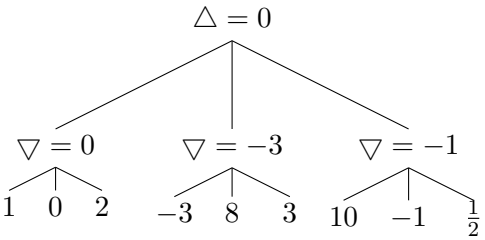
Once we have interpreted the game rules we can move on to playing the game. This problem reduces to finding the sequence of moves that will result in the highest goal value. In this section we will discuss some approaches to playing the game.

3.1 Tree search

As we have seen in the first section a formal description of a game is an inefficient way to describe the game. It is however inevitable to consider game states when playing the game. A classical an useful way to represent a game is using a game tree. The initial game state is s_0 at the root of the tree, and each next layer contains the game states that can be reached from the previous layer. Each time the players pick their moves the game state changes to a state one layer deeper, corresponding to the moves selected.

3.1.1 Minimax

One way to decide what move to select is the minimax algorithm. Suppose we have the following game tree.



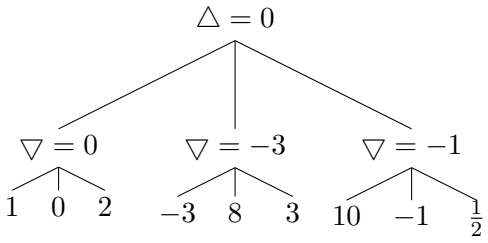
The $\triangle$-nodes are called 'max' nodes, and the $\triangledown$-nodes are the 'min' nodes. First the game tree is computed entirely, and then every node receives a value according to:

$$u(t) = \begin{cases} goalvalue(t) & \text{if } t \text{ is terminal} \\ max_{s \in succ(t)}\, u(s) & \text{if } t \text{ is } \triangle\text{-node} \\ min_{s \in succ(t)}\, u(s) & \text{if } t \text{ is } \triangledown\text{-node} \end{cases}$$

When done computing, the player picks the move that will take the player to the game state with the highest utility. The algorithm can easily be extended to multiplayer games by assuming the layers in which the other players take turns all have min-nodes. Although this brute force approach is simple and guarantees to find the optimal path, it is very inefficient. Suppose a tree has an average branching factor of b, and on average the game ends after d turns, the game tree has b^d states. Exploring a complete game tree is within computational bounds for for a simple game like Tic Tac Toe, but not for a complicated game like Chess.

3.1.2 Reducing the branching factor

One way to reduce the number of states we need to expand would be to reduce the branching factor b. A well-known algorithm to do so is $\alpha\beta$-pruning. This algorithm trims away the branches that don't influence the outcome of the minimax algorithm, and will therefore usually reduce search time. Recall our tree:



Assuming we search the tree depth-first, we will first mark the leftmost $\bigtriangledown$-node with 0. When we then reach the terminal node that is marked with -3, we are certain that its parent's value will be smaller then or equal to -3, and therefore is always less than 0. The outcome of this subtree won't influence the value of the max node, so the nodes with 8 and 3 don't need to be explored. For the same reason the node marked with $\frac{1}{2}$ does not need to be explored. In this tree search time will barely be influenced, but benefits can be significant when the nodes represent large subtrees that can be pruned away.

3.1.3 Reducing the tree depth

Another way to reduce the search time is to reduce the tree depth, a method to to so is called iterative deepening. First we calculate the game tree up to a certain depth, say 10 layers. For the nodes at the maximum depth the utility is unknown, so we estimate it. After that the minimax algorithm is applied in the usual way. When a player selects an action we need to compute a new layer at the bottom, re-estimate the utility and update the minimax values for each node. When estimating the utility the algorithm is no longer guaranteed to provide the best move, but will provide a reasonably good move depending on how well the utility is estimated.

3.1.4 Estimating utility

The estimation of utility is done with what is usually called a heuristic function. Most types of heuristics for General Game Playing share the same principle. Consider the following rule:

```
goal(xplayer, 100) <=
      true(cell(1,?a,x))
      true(cell(2,?a,x))
      true(cell(3,?a,x))
```

This rule is the 'column' rule in Tic Tac Toe, `xplayer` is awarded a score of 100 when it managed to form a line. Suppose now that both cell (1,1) and cell (3,1) have been marked with an x. The first and last literal of the rule have been satisfied, which is a fairly good indication that we're close to receiving 100 as a goal value. This approach can be improved by introducing fuzzy logic, where truth values are no longer exactly 0 and 1 but can take on all values between 0 and 1. The 2006 GGP world champion FluxPlayer [6] made use of this technique. It associated a degree of truth with every fact according to the following formulas [6] where a is an atom, f and g are formulas and z is a game state:

$$
\begin{aligned}
eval(a, z) &= \begin{cases} p & \text{if } a \text{ holds in the current state} \\ 1-p & \text{otherwise} \end{cases} \\
eval(f \wedge g, z) &= 1 - S(1 - eval(f, z), 1 - eval(g, z)) \\
eval(f \vee g, z) &= S(eval(f, z), eval(g, z)) \\
eval(\neg f, z) &= 1 - eval(f, z) \\
S(a, b) &= (a^q + b^q)^{\frac{1}{q}}
\end{aligned}
$$

The S formula is called the truth function for disjunctions, the truth function for conjunction is defined using this function as well. The outcome of $S(a, b)$ is greater when a and b are greater, i.e. a is more true if a or b is more true. Negation is modeled as usual with the complement operation. These formulas offer a clever way to do calculations with these degrees of truth. This method however requires some experimentation with different values for p and q.

3.2 Simulation-based approach

Finding adequate heuristics turns out to be a very difficult and precise process, since a badly derived heuristic function barely contributes to selecting the right action and may even point players in the wrong direction. In 2007 a new approach emerged that proved to be superior to the heuristic-based approach in many games.

One advantage that computers have over humans is computational power and the simulation-based approach uses this to its advantage. The concept is rather simple: let the computer simulate lots of random matches and see what moves result in a large payoff. The game of Go had previously turned out to be quite difficult to play for computers, and the idea of simulating large numbers of games had successfully been applied to Go.

The 2007 winning entrant CadiaPlayer [7] first employed this Monte Carlo method. It selects actions to explore according to the formula:

$$a^* = argmax_{a \in A(s)} \left\{ Q(s, a) + C \sqrt{\frac{\ln N(s)}{N(s, a)}} \right\}$$

a^*	The action to explore
$A(s)$	The set of actions in state s
$Q(s, a)$	The average payoff for action a in state s
C	A parameter to tune balance between exploration and exploitation
$N(s)$	The number of visits to state s
$N(s, a)$	The number of times action a has been sampled in state s

The search begins in the initial state as usual, all players select an action to explore and the search proceeds to the next state. This is repeated until a terminal state is encountered. All states on the path from the initial to the final node have their $Q(s, a)$, $N(s)$ and $N(s, a)$ values updated. Each player keeps track of their own average payoff values and number of visits in each game state. Short paths to a winning state are usually preferred over long paths, so a discount factor $\gamma = 0.99$ is used. Suppose action a is sampled in state s, the goal value is g, and s is T steps away from the terminal node, the payoff in that node is updated according to:

$$Q(s, a) = g\gamma^T \frac{N(s)}{N(s)+1}$$

The algorithm offers a way to balance between exploration and exploitation. When an

action has not been sampled before ($N(s,a) = 0$) the algorithms defaults to exploring this action as the second term – also called the UCT bonus – is infinite. When an action has been sampled the UCT bonus for that action slightly decreases because $N(s,a)$ is increased, and the UCT bonus for every other action slightly increases because $N(s)$ is increased. When the time has come to select an action in the actual game the player picks the action with the highest value for $Q(s,a)$, since it has turned out to result in the highest average payoff. In order to reduce required memory, only one new game state is stored in the search tree for every simulated game.

The great advantage over the heuristic-based solution is that the algorithm does not require any domain-specific knowledge. It may however be disadvantageous to use the algorithm when dealing with games that take a long time to play randomly. Because games such as puzzles or mazes will take fairly long to play randomly, playing a large number of these matches will probably not be possible when dealing with strict time constraints.

4 Implementation

This section concerns my own implementation of a GGP agent, based on CadiaPlayer [7] that was discussed in section 3.

4.1 Parsing

A successful GGP agent needs to be able to read and interpret a rule sheet. This is done in the parser class. First it reads the sheet and omits every line that starts with a semicolon as they indicate a comment. Then it classifies each line as either a:

- Dynamic fact – indicated with `init`
- Role – indicated with `role`
- Rules and static facts – everything else

4.1.1 Fact class

We have seen that a fact is either an object constant, a variable or a relation applied to zero or more constants and variables. Object constants and variables are represented as nullary relations. A `Fact` is therefore a `String` (the object, variable or relation) and an array of `Fact`s which is only filled for relations. The `Fact` class supports building the fact from a `String` in either infix of prefix mode.

`arity()` – The arity of the fact, 0 if the fact is a variable or an object constant
`buildInfix(String s)` – Builds a fact in infix mode, e.g. `f(x)`
`buildPrefix(String s)` – Builds a fact in prefix mode, e.g. `(f x)`
`containsVar()` – True if the fact contains a variable
`isConstant()` – True if the fact is an object constant
`isDistinct()` – True if the fact is `distinct`
`isVariable()` – True if the fact is a variable
`isNegative()` – True if the fact is `not`

4.1.2 Rule class

As discussed, a rule has the form $h \Leftarrow b_1 \wedge ... \wedge b_n$. A rule therefore has a fact as a head and an array of facts as its body. All static facts (such as `role(x)` or `succ(1,2)`) are represented as a rule that has zero literals and hence is always true.

4.1.3 KnowledgeBase and RuleBase classes

The `RuleBase` is a collection of all rules of a game. When playing a game typically only one `RuleBase` is needed as the rules and static facts do not change throughout the game.

`add(Rule r)` – Add a `Rule`
`toRuleArray()` – Returns an array of all `Rule`s in the `RuleBase`

A `KnowledgeBase` is a collection of facts, and it corresponds to the dynamic part of the game. Every game state has its own `KnowledgeBase` containing the facts corresponding to that game state.

`add(Fact f)` – Add a `Fact`
`addDoes(Fact f)` – Add a does-`Fact`, kept separate to be able to remove them easily
`getFactArray()` – Returns an array of all `Facts` in the `KnowledgeBase`
`removeDoes()` – Removes all does-`Facts`

4.2 GameState

The game practically revolves around this class, as the game state is the essential part of the search. It comprises a `KnowledgeBase` and a `RuleBase`, as well as an array of `Roles`.

`getLegalMoves(Role r)` – Computes the legal moves for `r`
`getNext(Move[] m)` – Computes the next `GameState` for moves `m`
`getGoalValue(Role r)` – Computes the goal value for `r`
`isTerminal()` – True if the state is terminal

4.2.1 Game

The `Game` class contains the `main` method, a game is played using the following command:

```
new Game(game, players, startclock, playclock).play(verbose)
```

The first argument `game` is the name of the game to be played, `players` is an array of `Player` objects, `startclock` is the number of milliseconds between receiving the game rules and commencing the game, `playclock` is the number of milliseconds each player has for selecting a move. Finally, `verbose` is a boolean that controls program output, `play(true)` will cause the program to show the course of the game. The method returns an array of `Integers` with the respective goal value for each player.

4.3 Unification

The methods in `GameState` provide information that needs to be derived from the game rules and facts. We first make one addition to the `Rule` class by classifying each literal either:

- Positive (b_p) – literals that `true`, `does`, `or` or `rule`
- Negative (b_n) – literals that contain `not`
- Distinct (b_d) – literals that contain `distinct`

The rule now has the following syntax:

$$h \Leftarrow (b_{p1} \wedge ... \wedge b_{pn}) \wedge (b_{n1} \wedge ... \wedge b_{nn}) \wedge (b_{d1} \wedge ... \wedge b_{dn})$$

The `RuleBase` has two methods for unification

`unificationExists(String s)` – True if a unification for `s` (e.g. terminal) exists
`findAllUnifications(String s)` – Returns a list of unifications for `s` (e.g. next)

The `GameState` can call these methods to derive information from the rules. For instance, when a player or game master calls `getNext()` on the `GameState`, the `GameState` will then call `findAllUnifications("next")` on the `RuleBase`. When the `GameState` is asked if its a terminal state or not, it will call `unificationExists("terminal")`.

4.3.1 Unifier

A unifier is the result of unifying two facts, and contains a mapping from variables to facts. For example:

`f(?x,1)` and `f(1,1)` result in unifier `[?x=1]`
`f(1,1)` and `f(?x,1)` result in unifier `[?x=1]`
`f(?x,1)` and `g(1,1)` are not unifiable
`f(?x,1)` and `f(1,2)` are not unifiable
`f(?x,?y)` and `f(1,2)` result in unifier `[?x=1, ?y=2]`
`f(1)` and `f(1)` result in unifier `[]`

4.3.2 Positive unification

Two possibilities for positive unification are true relations and does relations. We compute a set of unifiers by iterating over every fact, we use a separate class for sets of unifiers called `UnifierSet`. For example, when attempting to unify `cell(?x, ?y, x)` with the facts `cell(1,1,b)`, `cell(1,2,x)`, `cell(2,2,x)`, `cell(2,3,o)` the resulting `UnifierSet` is {`[?x=1,?y=2]`,`[?x=2,?y=2]`}.

There are two operations on `UnifierSet`s for positive unification: union and intersection. The intersection of two `UnifierSet`s u_1 and u_2 results in a `UnifierSet` u_3 that contains every unifier that is in u_1 and u_2, and the union results in a `UnifierSet` u_3 that contains the every unifier that is in u_1 or in u_2. We then define the relations and and or as follows:

unifications(and(g,f)) = intersection(unifications(f), unifications(g))
unifications(or(f, g)) = union(unifications(f), unifications(g))

4.3.3 Negative unification and distinct

Body literals that contain `not` require a different approach. When trying to unify `not(cell(?x,?y,b))` we first apply positive unification to `cell(?x, ?y, b)` and call the resulting `UnifierSet` u_n. If there were positive literals in the body we call the `UnifierSet` from positive unification u_p, and we proceed by removing every unifier from u_p that occurs in u_n. If there weren't any positive literals we return true if the `UnifierSet` is empty (i.e. no unification exists) and false if the `UnifierSet` contains a unifier (i.e. some unification exists).

Distinct literals finally only occur when the rule has positive literals. Suppose the distinct literal is `distinct(?p,?q)`. First we find all positive unifications for the positive literals and call the resulting `UnifierSet` u_p. We then proceed by removing every unifier from u_p that has `?p` and `?q` mapping to the same fact or that has `?p` and `?q` map to each other.

4.4 Player

To allow easy extension the implementation features a `Player` interface that has the following methods:

`initialize(GameState g, Role r, int startClock, int playClock)` – Initializes the game with `GameState g`, `r` is the role of this player, the clocks specify how long a player may take to select a move

`nextMove()` – Requests the player for a new move
`nextTurn(Move[] m)` – Allows the player to update its model according to the moves `m` that the players have chosen
`getName()` – Each player has its own name

The implementation includes two players, a `RandomPlayer` that simply picks a random move every turn and a `SimulationPlayer` that is based on CadiaPlayer [7] that was also discussed in section 3.2. When the `initialize()` method is called it creates an initial `sState` object which is a node in the search tree. When `nextMove()` is called, it first simulates an entire game as specified in section 3.2 and then updates the $Q(s, a)$ values in the tree. It times the amount of milliseconds needed for one search and then estimates how many games it can play before the playclock runs out. After this number of searches it returns the move that has the highest $Q(s, a)$ value in the current sState. When `nextTurn()` is called the player updates its model by updating the current `sState`.

4.5 Improvement

There is still some room for improvement in the reasoning part. After implementing the algorithm some games turned out to be using recursive rules from the following type:

```
greater(?x, ?y) <= succ(?x, ?y)
greater(?x, ?y) <= succ(?x, ?z) ∧ greater(?x, ?y)
```

When attempting to unify the second rule the algorithm will first find all unifications for `succ(?x, ?z)`, every unification for `greater(?x, ?y)` and then attempts to merge them. Since the rule is recursive however, the algorithm will find itself in an infinite regression and eventually causes the program to error. Unfortunately the algorithm is fundamentally unfit for these kinds of rules so games that use them are not supported. Another possible way to improve the implementation is to make the unification more efficient. It is able to play most simple games real time against human players ($\sim$1s per turn) but more complex games may take quite long.

5 Results

5.1 Single player games

Performance on single player games can be tested by comparing the score of the implementation with a player that plays a random move every turn. Recall that scores between 0 and 100 are appointed to each player. Both the random player and the simulation player have played all 41 single player games 10 times using a play clock of 500 ms and a start clock of 0 ms. The simulation player scored significantly better.

Player	**Average score**
RandomPlayer	29.4
SimulationPlayer	59.3

5.2 Two player games

Performance on two player games can be measured by letting the implementation compete with a random player. The players played all 54 two player games 10 times, the play clock for the experiment was 100 ms, the start clock was 0 ms.

vs	**RandomPlayer**	**SimulationPlayer**
RandomPlayer	(w) 38.5% (l) 33.1% (d) 28.3%	(w) 20.9% (l) 50.4% (d) 24.6%
SimulationPlayer	(w) 48.4% (l) 22.5% (d) 28.8%	(w) 41.1% (l) 30.0% (d) 28.9%

Every game has a first and second player, that do not necessarily have the same moves. The top left cell shows that when both players were represented by random players, the first player won slightly more often. The bottom right cell shows us that the win/loss ratio for the players are very similar when both players are represented by simulation players. The remaining cells show us that simulation players do well against random players, the first and second players respectively win 48.4% and 50.4% of their games. The draw rate is very constant for every pair of players.

For the second experiment the play clock was increased tenfold to 1000 ms, the start clock remained 0 ms, with following results:

vs	**RandomPlayer**
SimulationPlayer	(w) 68.7% (l) 13.0% (d) 18.3%

For random vs random and simulation vs simulation the experiment has not been rerun, because results would have been very similar to the ones from the first experiment – the random player does not use the play clock at all, the simulation players would likely have become equally stronger. The simulation player performs a lot better against the random player – it wins 68.7% of its games. An equal increase in wins is expected for the random vs simulation case, as experiment 1 did not show any significant differences with the simulation vs random case.

References

[1] Pell, B. (1992). METAGAME: A New Challenge for Games and Learning.

[2] Pell, B. (1994). A Strategic Metagame Player for General Chess-Like Games.

[3] Zillions of Games (n.d.). http://www.zillions-of-games.com/supportedFAQ.html

[4] Genesereth, M., Love, N. (2005). General Game Playing: Overview of the AAAI Competition.

[5] Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M. (2008). General Game Playing: Game Description Language Specification.

[6] Schiffel, S., Thielscher, M. (2007). Fluxplayer: A Successful General Game Player.

[7] Finnsson, H., Bjornsson, Y. (2008). Simulation-Based Approach to General Game Playing

[8] General Games Repository (n.d.). http://ggp-repository.appspot.com/

A List of playable games

Not all of the games are actual existing games, some of them are artificial games solely meant for research. They have been taken from the General Games Repository [8].

2pffa
2pttc
3pConnectFour
3pffa
3pttc
4pffa
aipsrovers01
asteroids
asteroidsParallel
asteroidsSerial
beatMania
blocker
blockerParallel
blockerSerial
blocks
blocksWorld
blocksWorldParallel
blocksWorldSerial
blokbox_simple
bomberman2p
brain_teaser_extended
breakthroughSmallHoles
bunk_t
buttons
catcha_mouse
checkLines
chickentictactoe
chickentoetictac
circlesolitaire
cittaceot
coloredtrails
conn4
connect5
connectFour
connectFourSuicide
cubicup
double_tictactoe_dengji
doubletictactoe
doubletoetictac
eightPuzzle
eotcatcit
eotcitcit
escortLatch
factoringApertureScience
factoringDelusionsOfAdequacy
factoringEasyTurtleBrain
factoringGeorgeForman
factoringIndeterminate
factoringMediumTurtleBrain
factoringPotentPotables
factoringRickRollers
factoringTheTurk
factoringTypedefDestruct
firefighter
ghostMaze2p
gt_attrition
gt_attrition_old
gt_centipede
gt_chicken
gt_prisoner
haystack
hodgepodge
incredible
knightsTour
knightsTourLarge
knightwar
lightsOn
lightsOnParallel
lightsOnSimul4
lightsOnSimultaneous
lightsOut
maze
minichess-evilconjuncts
minichess
nineBoardTicTacToe
numbertictactoe
onestep
pawnWhopping
pearls
peg
pegEuro
pentago
pentagoSuicide
point_grab
ruleDepthLinear
simultaneousWin2
slidingpieces
snakeParallel
snake_2009
snake_2009_big
stateSpaceLarge
stateSpaceMedium
stateSpaceSmall
sudoku
sum15
survival
switches
ticTacToe
ticTacToeClassic
ticTacToeLarge
ticTacToeLargeSuicide
ticTacToeNoVars
ticTacToeParallel
ticTacToeSerial
ticTicToe
ticblock
tictactoe-init1
tictactoe_3d_2player
tictactoe_3d_6player
tictactoe_3d_small_2player
tictactoe_3d_small_6player
tictactoe_3player
tictactoe_orthogonal
tictactoesuicideextra
tictactoex9
toetictac
tpeg
troublemaker01
troublemaker02