Künstl Intell (2011) 25: 17–24
DOI 10.1007/s13218-010-0077-4

FACHBEITRAG

# Centurio, a General Game Player: Parallel, Java- and ASP-based

**Maximilian Möller · Marius Schneider · Martin Wegner · Torsten Schaub**

Received: 30 July 2010 / Accepted: 11 September 2010 / Published online: 21 December 2010


**Abstract** We present the General Game Playing system Centurio. Centurio is a Java-based player featuring different strategies based on Monte Carlo Tree Search extended by techniques borrowed from Upper Confidence bounds applied to Trees as well as Answer Set Programming (for single-player games). Centurio's Monte Carlo Tree Search is accomplished in a massively parallel way by means of multi-threading as well as cluster-computing. Another major feature of Centurio is its compilation of game descriptions, states, and state manipulations into Java, yielding an edge over existing Prolog-based approaches. Centurio is open source software freely available via the web.

**Keywords** Answer set programming · General game playing · Monte Carlo tree search · Parallelization

## 1 Introduction

General Game Playing (GGP; [9]) has attracted much attention in the last years because of its integral nature, necessitating the combination of various techniques from Artificial Intelligence within a realtime environment. In this way, it can be regarded a modern "fruit fly"[1] of Artificial Intelligence. Besides advanced search, GGP requires techniques from agent programming, automated planning, decision making, game theory, knowledge representation and reasoning, and many more. Notably, current GGP systems can be roughly grouped into knowledge-free and knowledge-based approaches, joining an old debate in Artificial Intelligence [3].

Centurio features a mix of such techniques, using a refined Monte Carlo Tree Search approach on symbolic state representations for multiplayer games and the knowledge-based problem solving approach of Answer Set Programming for single-player games. The former is supported by a massively parallel architecture taking advantage of multi-threading and cluster-computing for maximizing the exploration of the game tree. Another major feature of Centurio is its compilation of game descriptions, states, and state manipulations into Java, yielding an edge over existing Prolog-based approaches. The idea is to translate each game description into Java source code while ensuring the semantics of the game description language GDL. The resulting Java code comprises the respective state information and calculates successor states according to the chosen move and the underlying game description. Centurio is open source software, freely available via the web at [13].

This paper is organized as follows. Section 2 gives a short overview about the basics of the Monte Carlo Tree Search method. Section 3 describes the approach of Centurio by detailing its major features. We summarize the approach and discuss future work in Sect. 4.

## 2 Background: Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS; [5]) is a best-first search method that is based on randomized explorations of the

[1] Drosophila melanogaster, often called the common fruit fly, is an important model organism in biology.

M. Möller · M. Schneider (✉) · M. Wegner · T. Schaub
Institute for Informatics, University of Potsdam,
August-Bebel-Str. 89, 14482 Potsdam, Germany
e-mail: manju@cs.uni-potsdam.de

T. Schaub
e-mail: torsten@cs-uni-potsdam.de

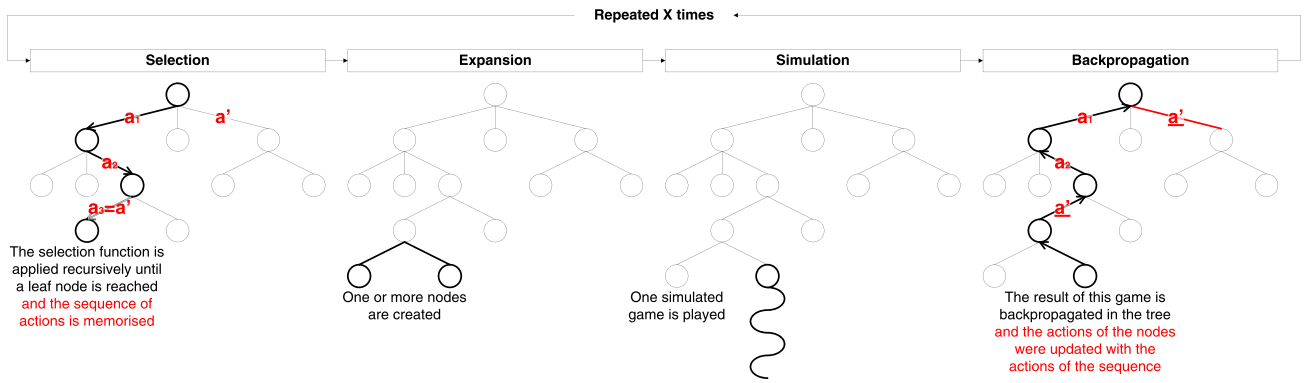



**Fig. 1** (Color online) Monte Carlo Tree Search (inspired by [5])

search space and a popular technique in General Game Playing having its roots in the old Asian game Go. Figure 1 shows the main four phases of MCTS: selection, expansion, simulation (random game) and backpropagation. It is an anytime algorithm, where at any time the collected information can be used to choose the currently best action.

The main extension to MCTS is Upper Confidence bound applied to Trees (UCT; [12]). It uses the Upper Confidence Bound Equation (1) as the selection strategy for MCTS. It provides a balance between exploring new states and exploiting existing knowledge about states.

$$Q^{\oplus}_{UCT}(s,a) = Q_{UCT}(s,a) + c * \sqrt{\frac{\log(n(s))}{n(s,a)}} \tag{1}$$

$Q_{UCT}(s,a)$ is the winning probability or average score in state $s$ with action $a$ and represents the exploitation part of (1). The second summand is weighted by a factor $c$ and represents the exploration part. $n(s,a)$ counts how often $a$ was chosen in state $s$, whereas $n(s) = \sum_a n(s,a)$. The exploration part decreases the more a move $a$ is chosen in state $s$.

In large state spaces UCT converges slowly to the most promising move. Therefore Rapid Action Value Estimation (RAVE; [8]) is an important extension to UCT. RAVE rates every action, which is selected at any time following state $s$. It is shown red in Fig. 1. The combination of UCT and RAVE is calculated through a weighting:

$$Q_{MCTS} = \alpha \cdot Q^{\oplus}_{UCT} + (1-\alpha) \cdot Q^{\oplus}_{RAVE}$$

where $\alpha$ is the weight, which starts with a value near 0 and increases over time.

Another extension for large branching factors is progressive widening [5]. It prunes all actions, which are under a certain threshold, to let UCT concentrate on important actions. After a certain number of simulations one action after another is unpruned.

## 3 The Centurio Approach

This section introduces the four major features of Centurio, whose interaction is visualized in Fig. 2. We begin with adoptions of MCTS necessary for a successful use in the domain of General Game Playing in Sect. 3.1. Because the strength of a MCTS-based General Game Player heavily depends on the number of random games, we focus on techniques that are able to increase it. One technique is the parallelization of MCTS to increase the number of random games run within a given time, described in Sect. 3.2. The other technique is to speed up the state manipulations and so each random game by translating the game description received from the gamemaster into Java sourcecode (see Sect. 3.3). To compensate the weakness of MCTS in single-player games with many game states but only one solution Centurio makes use of Answer Set Programming as described in Sect. 3.4.

### 3.1 MCTS—Adoptions to Domain of GGP

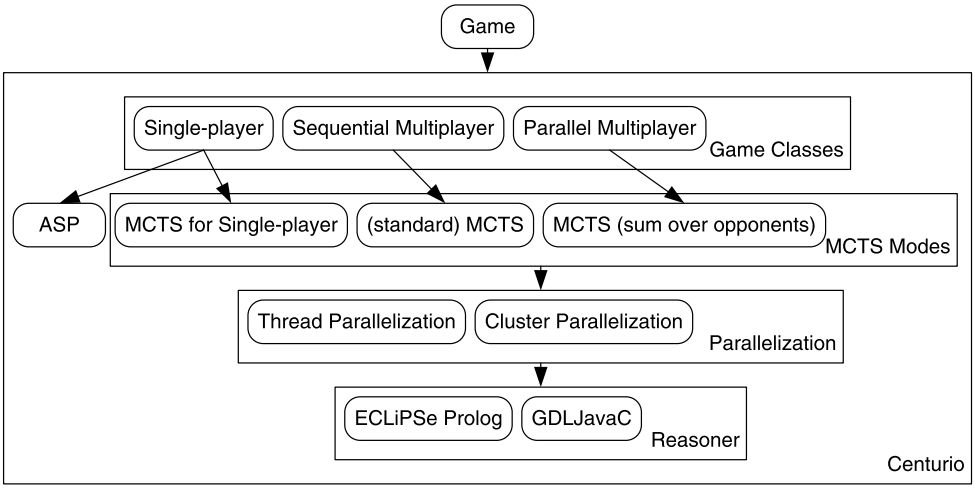
An effective use of MCTS in the domain of GGP requires a distinction of games into single-player, sequential, and parallel games.

#### *3.1.1 MCTS in Single-Player Games*

In single-player games no other players are involved and the game result is solely dependent on the own moves. This makes it practical to always store the moves of the random game that achieved the best result.

In single-player games the game tree is explored with the usual MCTS approach but Centurio does not use the winning probability $Q_{UCT}(s,a)$ of (1) to make the decision for the move that is transmitted to the Gamemaster. The fact that for each random game the true outcome is available makes it possible for Centurio to choose the best move based on the best played random game. This means that Centurio plays at anytime the best game it is currently aware of. In case

**Fig. 2** Design of Centurio




MCTS finds a maximum score game, Centurio sends the stored moves of the game move by move to the Gamemaster.

#### *3.1.2 MCTS in Sequential Multiplayer Games*

The MCTS/UCT approach is designed for sequential multiplayer games with 2 players. To be able to play sequential games with more than 2 players, every player has an own winning probability. If the opponent has to choose a move, Centurio assumes that the opponents intelligence is as strong as its own and its strategy is the one, Centurio would play instead. The selection of the move, which is transmitted to the Gamemaster, uses the selection strategy:

$$\pi(r) = \arg\max_{a \in \mathcal{A}}(Q_{UCT}(r, a)). \tag{2}$$

In (2), $\pi$ is the selection strategy and $r$ is the current game state. $\mathcal{A}$ is the set of all possible actions in this state.

Before this selection strategy is used, Centurio checks if the list of possible moves contains at least one move which causes Centurio to win with a maximum score of the game in the next step. In this case, Centurio immediately selects this victory move.

#### *3.1.3 MCTS in Parallel Multiplayer Games*

Parallel games are the most difficult ones because every player has several moves in each state. If each player has at most $k$ possible moves, for $n$ players $k^n$ is an upper bound for the number of moves. To determine Centurio's move maximizing its own score, all combinations of all other players must be taken into account. Under the assumption that there are $n$ players and Centurio is the $m$-th player ($1 \leq m \leq n$), the following formula determines the best move:

$$\pi(r) = \arg\max_{a_m \in \mathcal{A}} \sum_{a_{i \neq m} \in \mathcal{A}} Q_{UCT}(r, (a_1, \ldots, a_n)). \tag{3}$$

However, the calculation of (3) is too expensive and so Centurio only uses it for the choice of the move to send to the Gamemaster. The selection function of MCTS needs to be very efficient. This fact forces us to estimate the best move by taking the move with highest mean UCT value over all players:

$$Q^{\oplus}_{UCT}(s, a) = \frac{\sum_{m=1}^{k} Q_{UCT_m}(s, a)}{k} + c * \sqrt{\frac{\log(n(s))}{n(s, a)}}$$

where $Q_{UCT_m}(s, a)$ is the winning probability of the $m$-th player in state $s$ with move $a$.

#### *3.1.4 MCTS in Cooperative Games*

Centurio cannot detect cooperative games and it seems that it is hard to extract this information from a GDL description. We do not follow the paranoid assumption [16] that every opponent is against Centurio, but we assume that every player tries to maximize its own score. We also assume that the best action for Centurio is also the best action for Centurio's team and so Centurio's score should be somewhat equal to the partners' ones.

### 3.2 Parallelization

MCTS is designed for high scalability. We introduce two parallelization approaches. The threadsafe parallelization (see Sect. 3.2.1) is only reasonable and usable on a multicore system. While it profits from computations on a single game tree, in the alternative cluster parallelization (see Sect. 3.2.2) each cluster node works on its own game tree.

#### *3.2.1 Thread Parallelization*

Thread parallelization uses local mutexes (soft barriers) to ensure that all threads can access the game tree without any

race conditions. In other words, every node has its own mutex. When a thread accesses a node, it reduces the node's MCTS value. The decision which action is chosen is based on the following equation:

$$\pi(s) = \arg\max_{a \in \mathcal{A}} \left( \frac{Q_{MCTS}(s,a)}{\sqrt{t(s,a)+1}} \right).$$

$t(s,a)$ is the number of threads with state $s$ including action $a$ and $\mathcal{A}$ is the set of all possible actions in this state. In most cases, the UCT values are small and a reduced UCT value causes other threads to choose a node with a higher value. The result of this case is that every thread chooses another way through the game tree. A very promising way consists of nodes with much higher UCT values than other nodes have. In this case, it is possible that threads take the same way through the game tree. With this strategy the UCT heuristic does not lose much influence and the processor cores are distributed over the most promising branches.

#### *3.2.2 Cluster Parallelization*

An easy way to parallelize a Java-based general game player for a cluster is provided by Terracotta [11]. Terracotta is an open source infrastructure software that offers a Network-Attached Memory (NAM) where a subset of the application's data structures can be stored. Every operation (read, write) on a structure, which is stored in the NAM and is therefore available for several nodes of the cluster, must be thread-safe. Terracotta is able to generalize thread-safe operations to the cluster.

In the case of cluster parallelization, MCTS is running independently on each cluster node because it depends on random games. Each node creates its own game tree that is independent from the other game trees investigated by other nodes. In each game tree, a root represents the current state of the game. We call its children the decision level that represents the states, which can be reached with one legal move in the current state. Every node expedites the convergence to the real winning probability of the states from the decision level if the results are shared between them. The idea is to merge the decision levels of all cluster nodes weighted by each tree node's visits by writing it to the NAM. Now the procedure responsible for sending the best move to the Gamemaster can take the decision on basis of the merged heuristic values of the decision level.

MCTS's strength depends on the number of random games it has performed. The more nodes are executing MCTS, the less Centurio is dependent from fortuity and the better the move can be. So for Centurio's approach there is no need to parallelize the algorithm itself. This brought us nearly linear speedup $(7, 42)$ in the number of random games on a cluster with up to 8 available nodes.

The effect of nodes working on the same game tree should be that each node profits from the others heuristic values in the selection phase of MCTS on each level (not only on the decision level). We tried to generalize the thread parallelization, introduced in Sect. 3.2.1, to the cluster but in fact, the access to the data in the NAM was too expensive and we were far away from linear speedup because the selection and backpropagation phase of MCTS have extensive read access to the game tree.

### 3.3 GDLJavaC—A GDL to Java Compiler

Every general game player, and in fact every artificial game player, is based on a low level mechanism for state manipulation, determining initial state, legal moves, or next state (called reasoner). The strength of the tree search algorithm heavily depends on the reasoners efficiency. Typically a general game player makes use of a Prolog-based reasoner. The Game Description Language (GDL; [4]) has its roots in the logic programming language Datalog and therefore parsing the game description to Prolog source code is the most obvious and widespread approach.

While for concrete games like Connect4, the state manipulation can be customized to reach speed improvements, in the general game discipline there is no information about the game available until Centurio has to play it. So tuning algorithms to a concrete game is not quite easy and contradicts the idea of generality.

Centurio uses the Prolog system $ECL^iPS^e$ [1], which provides an interface to Java. If several Prolog instances are needed for parallelization, the player is forced to use a socket-based communication interface, producing some communication overhead. To keep this overhead as small as possible, the random games are played autonomously by the reasoner and Centurio has no influence on a random game once it is triggered. Waugh [18] presents an approach for improving the speed of state manipulation by generating C++-Code out of a game description, which we have adapted to Java. Besides performance improvements, the approach gives Centurio the possibility to regain control over random games because it makes communication through external interface obsolete.

The central idea is to parse each game description into Java source code, in view of the semantics of special-purpose GDL syntax. The resulting code is able to calculate all derivations of each predicate that are true in a given state.

The basic components of the system are covered first. A game description consists of two parts, a static and a dynamic one. GDL facts like $(role\ ?x)$ build the static part, because they do not depend on the game's progress and are true in the derivation of each state. They have to be calcu-

```
public static LinkedList<step> step_0(HashMap<String,LinkedList<GDLPredicate>> tempFacts) {
    LinkedList<step> results = new LinkedList<step>();
    LinkedList<GDLPredicate> res0 = tempFacts.get("step");
    if(res0 != null) {
        for(GDLPredicate res0Item : res0) {
            step res0Itemstep = (step) res0Item;
            if(true) {
                LinkedList<succ> res1 = succ.succ_10(res0Itemstep.var0,tempFacts,moves);
                for(GDLPredicate res1Item : res1) {
                    succ res1Itemsucc = (succ) res1Item;
                    step temp2step = new step(res1Itemsucc.var1);
                    results.add(temp2step);
                }
            }
        }
    }
    return results;
}
```

**Fig. 3** Example method source code for predicate *step*

lated only once. Dynamic rules like

$$(<= \ (next \ (step \ ?y)) \quad (true \ (step \ ?x)) \quad (succ \ ?x \ ?y)) \tag{4}$$

depend on what is true in the current state and its derivations, that might differ in various states, build the dynamic part.

For each predicate symbol (e.g. *step*), a Java class is created. The dynamic part of the game description consists of class instances that represent, for instance, a fully instantiated *step* predicate. This means the variable in the predicate is replaced by a term from all possible assignments. All legally instantiated *step* predicates are derived by the class's method.

For illustration, we take a closer look at the method generation for rule (4): we get to game step $y$, if the game step is currently $x$ and $y$ is the successor of $x$. This predicate counts moves. Ignoring the meta predicate *next*, predicate *step* results in the Java method in Fig. 3. The methods' name *step*_0 indicates that the corresponding predicate has a single variable as argument, which is unbound, and the method shall provide all possible bindings of it as result. A method called *step*_1 would indicate, there is one bound variable that is passed as method argument and the method only determines, whether the predicate is satisfiable with it as argument. The hashmap *tempFacts* corresponds to the dynamic part of the game description. For every instance of *step* that exists in *tempFacts*, we have to determine the successor of the first and only property of *step* and create a new *step* instance with the successor as new property. To determine the successor we call the method *succ*_10 of the class *succ*. The first variable of *succ* is bound through the variable $x$ of the *step* predicate. So we want only all possible bindings for the second variable $y$ as the result and based on that, we create a new *step* instance with the new property.

**Table 1** Benchmark results, comparison of Centurio V2.1 (Prolog) and Centurio (`GDLJavaC`), metrics are taken after a startclock of 30 seconds

| Versions | Random games | Factor |
|---|---|---|
| Tic Tac Toe | | |
| `Centurio V2.1` | 3994 | 2.66 |
| `Centurio GDLJavaC` | 10611 | |
| Connect 4 | | |
| `Centurio V2.1` | 4864 | 2.48 |
| `Centurio GDLJavaC` | 12057 | |
| Checkers | | |
| `Centurio V2.1` | 1232 | 2.16 |
| `Centurio GDLJavaC` | 2659 | |
| Skirmisch | | |
| `Centurio V2.1` | 126 | 2.46 |
| `Centurio GDLJavaC` | 312 | |
| Blobwars | | |
| `Centurio V2.1` | 29 | 2.9 |
| `Centurio GDLJavaC` | 85 | |

### 3.3.1 Benchmarks

To measure the performance of GDLJavaC a benchmark was performed. Four games: Tic Tac Toe, Connect4, Checkers, and Blobwars were played with a startclock of 30. Two versions of Centurio compete against each other. `Centurio V2.1` has $ECL^iPS^e$ Prolog as reasoner and `Centurio GDLJavaC` is backed by GDLJavaC. After the startclock, the achieved number of random games are recorded. The

results are presented in Table 1. GDLJavaC performed better than the corresponding Prolog reasoner. In every game, the new reasoning mechanism can double up the executed random games. The more random games are performed by Centurio's Monte Carlo Tree Search, the better is its playing ability.

With GDLJavaC, Centurio is able to control the random game process and we concentrate on random game policies for MCTS in future work.

### 3.4 Answer Set Programming

A new approach in General Game Playing is the use of Answer Set Programming (ASP; [2]). ASP is a declarative problem solving paradigm offering a rich modeling language [6] along with highly efficient inference engines based on Boolean constraint solving technology [7]. The basic idea of ASP is to encode a problem as a logic program such that its answer sets represent solutions. Because ASP is designed for problems in NP, it is a perfect support in General Game Playing for problems at the corresponding complexity. Centurio uses the ASP systems (`Clingo` and `iClingo`) from the University of Potsdam, which are available in the `potassco` project under [6].

First approaches, described in Sect. 3.4.1 and [17], use ASP to solve single-player games. Another application of ASP is automated theorem proving [15] to prove automatically and efficiently game characteristics. In the next section, the topic of single-player games is discussed. Thereafter an evaluation of this approach and a discussion of the future work is touched.

#### *3.4.1 ASP as Single-Player Game Solver*

The use of MCTS is a typical approach in General Game Playing although it is not always the best choice. If there is only one solution with 100 points and a large state space, the probability to find it is near 0 (e.g. Lightsout and 8puzzle). Because MCTS weights actions according to their winning probability or their average goal score, it can be easily fooled by certain games (e.g. Firefighters [14]).

In the case of single-player games, where no other players have effect on the game, ASP is a perfect way to solve such games. Because ASP is designed for problems in NP, which is the same worst case complexity of single-player games, and the syntax of GDL is similar to the syntax of ASP, ASP can be used in a straightforward way. Only a mapping from the Game Description Language into the input language of ASP is necessary. The following mapping is invented in [17] and only extended from Centurio in some points. A detailed and technical view on the mapping can be found in [13].

1. Disjunctions are removed and equality added.[2]
2. A time parameter to all meta predicates and time dependent predicates is added.
   (e.g. $next(\phi) \mapsto holds(\phi, t+1)$)
3. Constraints are added that guarantee the game semantics and winning properties.
   (e.g. :- $terminal(t)$, $not\ goal(R, 100, t)$, $role(R)$.
   is an integrity constraint which ensures the answer set has a terminal state with 100 points)

One main bottleneck is the memory consumption of grounding. It can happen that the grounding fails because of shortage of memory. One approach to solve this problem is the use of an incremental grounding system, e.g. `iClingo`. For each time step, grounding and solving alternates. Hence it is not necessary to ground the whole problem in advance.

In addition, the use of `Clingo`, as a non-incremental grounding system, has a drawback. It is necessary to estimate the length of the move sequence to reach the maximum score. In most cases, only an approximation of this length is possible. If it is an underapproximation no solution is found. And if it is an overapproximation the instantiated problem gets to big and the solving process gets slower. In contrast, `iClingo` does not need this approximation because of its iterative grounding and solving for each time step.

#### *3.4.2 Benchmarks*

In Table 2, we can see the performance of ASP systems to solve single-player games. `Clingo` is an out-of-the-box combination of the grounder `gringo` and the solver `clasp`. `iClingo` is the extension to incremental grounding. Every "–" in the first two columns marks a time out (more than 600 s) or a shortage of memory (2 GB). The benchmarks were executed on an Intel i7 CPU 940 (using one core). The last two rows show the sum of running times of all single-player games (see [14]) and the number of games which could not be solved with ASP.

The last column shows the possibility to solve the game with the Centurio implementation of MCTS. Every ✓ means that the MCTS approach can solve it whereas – means that it cannot be solved with MCTS.

We see that the choice between `Clingo` and `iClingo` can be crucial to be able to solve a game. In addition the conclusion of these tests are that ASP can be a useful extension to MCTS in cases where MCTS fails.

#### *3.4.3 Future*

To overcome the grounding problem there exist some new approaches. An extension of `iClingo` takes advantage of

[2] There is no direct way to use equality in GDL.

**Table 2** Time in sec, Clingo (complete grounding with heuristic for game depth), iClingo (incremental grounding), possibility to solve it with MCTS (✓/−)

| Game | Clingo | iClingo | MCTS |
|---|---|---|---|
| 8puzzle | **18.09** | 164.44 | – |
| asteroidsserial | 5.93 | **0.19** | ✓ |
| buttons | 0.00 | 0.00 | ✓ |
| god | 109.04 | **10.24** | ✓ |
| hanoi | 374.73 | – | – |
| lightsout | 0.95 | **0.91** | – |
| pancakes | 77.35 | **14.49** | ✓ |
| queens | 5.23 | 5.53 | ✓ |
| uf20-01.cnf.SAT | **0.28** | – | – |
| snake_2009_big | 237.51 | – | ✓ |
| twisty-passages | **1.63** | 7.79 | – |
| 46 single-player games | | | |
| $\sum$ | **6317.77** | 7669.55 | – |
| Outs | **9** | 12 | 22 |

the fact that the domain of each `next` and `true` predicate is the same. Hence the instantiated encoding can be much smaller.

On the other hand, it is possible to optimize the running time of solver with an automatic configuration of the parameters. For example, resolution based pre-processing of `clasp` can extremely speed up the running time in some games.

In addition, it is not clear in which other application of General Game Playing ASP can also be used. One approach could be to decompose a two player game into two independent single-player games [10]. Or to use it in games where all player have to cooperate to reach the same goal.

In all scenarios where the task can be mapped to an NP-problem ASP is an efficient way to solve it. But it is not limited to it. ASP can be used to calculate Monte Carlo simulations, too. If it is faster than Prolog and GDLJavaC, it could be a third possible choice as a reasoner.

## 4 Discussion

To test the strength and interaction of all approaches, Centurio took part in the competition General Intelligence in Game-Playing Agents (GIGA'09). This competition was in the scope of the International Joint Conferences on Artificial Intelligence (IJCAI) held in Pasadena, California, USA. Centurio completed the preliminaries as the group leader. In the preliminaries Centurio won against CadiaPlayer the double world champion in 2007 and 2008 from Reykjavík (Iceland). In the finals Centurio achieved a respectable fourth place.

Nevertheless, Centurio has some known weaknesses:

A very large branching factor of the game tree, often found in parallel multiplayer games, is a critical weakness of MCTS. In most cases, progressive widening cannot avoid this because the start- and playclock are mostly to short and the branching factor is too large.

The performance of MCTS is based on the number of random games played and so the time for one random game must be a fraction of a second. One random game for games like chess or othello requires more than one second. For such games, MCTS performs very bad with current GDL-based state manipulation approaches.

Some game descriptions include two distinct games. For example a sequential multiplayer game which includes two single-player games for each player. The move choice of player 1 would have no direct influence on player's 2 winning probability. Without detection of such game properties, the state space blows up unnecessarily. In future versions, Centurio should be able to detect such games.

## References


1. *ECL$^i$PS$^e$* (2010) http://www.eclipseclp.org/
2. Baral C (2003) Knowledge representation, reasoning and declarative problem solving. Cambridge University Press, Cambridge
3. Brooks R (1991) Intelligence without representation. Artif Intell 47(1–3):139–159
4. Ceri S, Gottlob G, Tanca L (1990) Logic programming and databases. Springer, Berlin
5. Chaslot G, Winands M, Szita I, van den Herik H (2008) Parameter tuning by the cross-entropy method. In: European workshop on reinforcement learning
6. Gebser M, Kaminski R, Kaufmann B, Ostrowski M, Schaub T, Thiele S A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`. Available at http://potassco.sourceforge.net
7. Gebser M, Kaufmann B, Neumann A, Schaub T (2007) clasp: a conflict-driven answer set solver. In: Baral C, Brewka G, Schlipf J (eds) Proceedings of the ninth international conference on logic programming and nonmonotonic reasoning (LPNMR'07). Lecture notes in artificial intelligence, vol 4483. Springer, Berlin, pp 260–265
8. Gelly S, Silver D (2007) Combining online and offline knowledge in uct. In: Ghahramani Z (ed) Proceedings of the international conference of machine learning (ICML 2007), pp 273–280
9. Genesereth M, Love N, Pell B (2005) General game playing: overview of the AAAI competition. AI Mag 26(2):62–72
10. Günther M, Schiffel S, Thielscher M (2009) Factoring general games. In: Proceedings of the IJCAI-09 workshop on general game playing (GIGA'09), pp 27–34
11. Inc T Terracotta (2010) http://www.terracotta.org/
12. Kocsis L, SzepesvÃri C (2006) Bandit based Monte-Carlo planning. In: ECML-06. LNCS, vol 4212. Springer, Berlin, pp 282–293
13. Möller FM, Schneider TM, Wegner M (2010) GGP University of Potsdam. http://www.ggp-potsdam.de
14. Schiffel S (2009) Dresden GGP server. http://euklid.inf.tu-dresden.de:8180/ggpserver/
15. Schiffel S, Thielscher M (2009) Automated theorem proving for general game playing. In: Proceedings of IJCAI'09

16. Sturtevant NR, Korf RE (2000) On pruning techniques for multi-player games. In: Proceedings of the seventeenth national conference on artificial intelligence and twelfth conference on innovative applications of artificial intelligence. AAAI Press, Menlo Park, pp 201–207
17. Thielscher M (2009) Answer set programming for single-player games in general game playing. In: Proceedings of the international conference on logic programming (ICLP). Springer, Berlin
18. Waugh K (2009) Faster state manipulation in general games using generated code. In: Proceedings of the 1st general intelligence in game-playing agents (GIGA)

**Maximilian Möller** is a M.Sc. student at University of Potsdam, where he received his B.Sc. in Informatics in 2009. His current research interests are General Game Playing and Machine Learning.

**Marius Schneider** is a Ph.D. student at University of Potsdamy, where he received his M.Sc. in Informatics in 2010. His research interests lie in the field of applied AI, specifically General Game Playing, Maschine Learning and Automatic Parameter Configuration for Boolean Constraint Solvers.

**Martin Wegner** is a M.Sc. student at University of Potsdam, where he received his B.Sc. in Informatics in 2009. His current research interests are General Game Playing and RFID-Chips.

**Torsten Schaub** had the pleasure to supervise the Centurio project and learned a lot from his students. Otherwise, he is heading the Knowledge Representation and Reasoning group at the University of Potsdam. He received his Diploma and Dissertation in Informatics in 1990 and 1992, respectively, from the Technical University of Darmstadt. He received his Habilitation in Informatics in 1995 from the University of Rennes I in France.