Simulation Control in General Game Playing Agents

Hilmar Finnsson and Yngvi Björnsson
School of Computer Science
Reykjavík University
{hif,yngvi}@ru.is.

Abstract

The aim of *General Game Playing (GGP)* is to create intelligent agents that can automatically learn how to play many different games at an expert level without any human intervention. One of the main challenges such agents face is to automatically learn knowledge-based heuristics in real-time, whether for evaluating game positions or for search guidance. In recent years, GGP agents that use Monte-Carlo simulations to reason about their actions have become increasingly more popular. For competitive play such an approach requires an effective search-control mechanism for guiding the simulation playouts. In here we introduce several schemes for automatically learning search guidance, as well as comparing them empirically. We show that by combining schemes one can improve upon the current state-of-the-art of simulation-based search-control in GGP.

1 Introduction

From the inception of the field of Artificial Intelligence (AI), over half a century ago, games have played an important role as a test-bed for advancements in the field. Artificial intelligence researchers have over the decades worked on building high-performance game-playing systems for games of various complexity capable of matching wits with the strongest humans in the world [Campbell *et al.*, 2002; Schaeffer, 1997; Buro, 1999]. The importance of having such an objective measure of the progress of intelligent systems cannot be overestimated, nonetheless, this approach has led to some adverse developments. For example, the focus of the research has to some extent been driven by the quest for techniques that lead to immediate improvements to the game-playing system at hand, with less attention paid to more general concepts of human-like intelligence like acquisition, transfer, and use of knowledge. The success of game-playing systems has thus in part been because of years of relentless knowledge-engineering effort on behalf of the program developers, manually adding game-specific knowledge to their programs. The aim of general game playing is to completely automate this process.

In *General Game Playing (GGP)* the goal is to create intelligent agents that can automatically learn how to skillfully play a wide variety of games, given only the descriptions of the game rules. This requires that the agents learn diverse game-playing strategies without any game-specific knowledge being provided by their developers. A successful realization of this task poses interesting research challenges for artificial intelligence sub-disciplines such as knowledge representation, agent-based reasoning, heuristic search, computational intelligence, and machine learning.

The traditional way GGP agents reason about their actions is to use a minimax-based game-tree search along with its numerous accompanying enhancements [Knuth and Moore, 1975; Atkin and Slate, 1988; Schaeffer, 1989; Breuker, 1998]. The most successful GGP players all used to be based on that approach [Schiffel and Thielscher, 2007b; Clune, 2007; Kuhlmann *et al.*, 2006]. Unlike agents which play one specific board game, GGP agents are additionally augmented with a mechanism to automatically learn an evaluation function for assessing the merits of the leaf positions in the search tree. In recent years, however, a new paradigm for game-tree search has emerged, the so-called *Monte-Carlo Tree Search (MCTS)* [Coulom, 2006; Kocsis and Szepesvári, 2006]. In the context of game playing, Monte-Carlo simulations were first used as a mechanism for dynamically evaluating the merits of leaf nodes of a traditional minimax-based search [Abramson, 1990; Bouzy and Helmstetter, 2003; Brügmann, 1993], but under the new paradigm MCTS has evolved into a full-fledged best-first search procedure that can replace minimax-based search altogether. MCTS has in the past few years substantially advanced the state-of-the-art in several game domains where minimax-based search has had difficulties, most notably in computer Go. Instead of relying on an evaluation function for assessing game positions, pre-defined search-control knowledge is used for effectively guiding the simulation playouts [Gelly and Silver, 2007].

The MCTS approach offers several attractive properties for GGP agents, in particular, it avoids the need to construct a game-specific evaluation function in real-time for a newly seen game. Under this new paradigm the main focus is instead on online learning of effective search-control heuristics for guiding the simulations. Although still a challenging task, it is in some ways more manageable, because such heuristics do not depend on game-specific properties. In contrast, au-

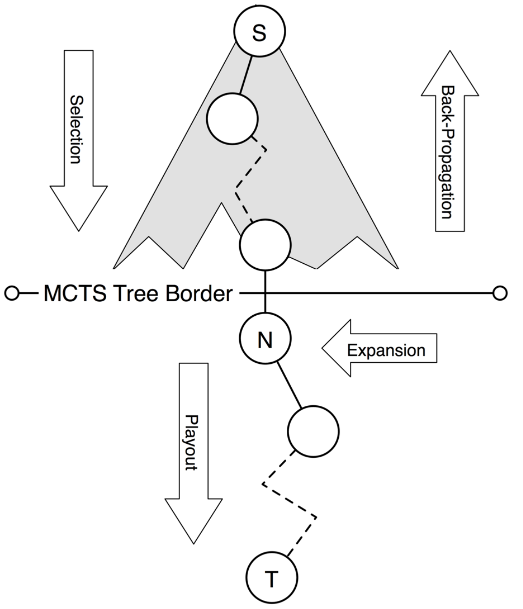

Figure 1: An overview of a single simulation.

tomatically learned heuristic evaluation functions that fail to capture essential game properties result in the evaluations becoming highly inaccurate and, in the worst case, even causing the agent to strive for the wrong objectives. GGP agents that apply the MCTS approach are now becoming increasingly mainstream, in part inspired by the success of CadiaPlayer [Björnsson and Finnsson, 2009] which won the last two AAAI GGP competitions.

In this paper we investigate several domain-independent search-control learning mechanisms in CadiaPlayer. The main contributions are an extensive empirical evaluation of different search-control mechanisms for GGP agents, both old and new. There is no single mechanism that dominates the others on all the games tested. We instead contrast their relative strengths and weaknesses and pinpoint game-specific characteristics that influence their effectiveness. We also show that by combining them one can improve upon the current state-of-the-art of simulation-based search-control in GGP.

The paper is structured as follows. In the next section we give a brief overview of the MCTS approach and how it is implemented in CadiaPlayer. Next we introduce several different search-control mechanism, that we then empirically evaluate using three different games. Finally, we survey related work before concluding and discussing future work.

2 Monte-Carlo Tree Search GGP Player

Monte-Carlo Tree Search (MCTS) continually runs simulations to play entire games, using the result to gradually build a game tree in memory where it keeps track of the average return of the state-action pairs played, $\mathcal{Q}(s, a)$. When the deliberation time is up, the method chooses between the actions at the root of the tree based on which one has the highest average return value.

Fig. 1 depicts the process of running a single simulation: the start state is denoted with S and the terminal state with T. Each simulation consists of four strategic steps: *selection*, *playout*, *expansion*, and *back-propagation*. The selection-step is performed at a beginning of a simulation for choosing actions while still in the tree (upper half of figure), while the playout-step is used for choosing actions once the simulated episode falls out of the tree and until the end of the game (bottom half of figure). The expansion-step controls how the game tree is grown. Finally, in the back-propagation-step, the result value of the simulation is used to update $\mathcal{Q}(s, a)$ as well as other relevant information if applicable.

The *Upper Confidence Bounds applied to Trees (UCT)* algorithm [Kocsis and Szepesvári, 2006] is commonly used in the selection step, as it offers an effective and sound way to balance the exploration versus exploration tradeoff. At each visited node in the tree the action a^* taken is selected by:

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

The $N(s)$ function returns the number of simulation visits to a state, and the $N(s, a)$ function the number of times an action in a state has been sampled. $A(s)$ is the set of possible actions in state s and if it contains an action that has never been sampled before it is selected by default as it has no estimated value. If more than one action is still without an estimate a random tie-breaking scheme is used to select the next action. The term added to the $\mathcal{Q}(s, a)$ function is called the UCT bonus. It is used to provide a balance between exploiting the perceived best action and exploring the less favorable ones. Every time an action is selected the bonus goes down for that action because $N(s, a)$ is incremented, while its siblings have their UCT bonuses raised as $N(s)$ is incremented. This way when good actions have been sampled enough to give an estimation with some confidence the bonus of the suboptimal ones may have increased enough to have this formula pick them for exploration. If a suboptimal action is discovered to be good, it needs a smaller bonus boost (if any) to be picked again, but if it still looks the same or worse it will have to wait longer now as the bonus will grow slower each time it is sampled. The C parameter is used to tune how much influence the UCT bonus has on the action selection calculations, but it can vary widely from one domain (or program) to the next.

In the playout step there are no stored $\mathcal{Q}(s, a)$ values available for guiding the action selection, so in the most simplistic case one would choose between available actions uniformly at random. However, there exists several more sophisticated schemes for biasing the selection in an informed way, as discussed in the next section.

The expansion step controls how the search tree grows, and a typical strategy is to append only one new node to the tree for each simulation: the first node encountered after stepping out of the tree [Coulom, 2006]. This is done to prevent using excessive memory, in particular if the simulation are fast. In Fig. 1 the newly added node in this episode is labeled as N.

For more details about the MCTS search in CadiaPlayer see [Finnsson, 2007; Björnsson and Finnsson, 2009].

3 Search Controls

In this section we describe four different, although related, search-control mechanism for guiding simulation runs in MCTS. The first one, MAST, is the one used in the 2008 version of CADIAPLAYER; the next one, TO-MAST, is identical to the first except that it is more restrictive about its learning. The third one, PAST, is more informed about how to generalized the learned control information, that is, it affects mainly states with similar properties. The fourth method, RAVE, is a state-of-the-art technique originally designed to expedite search-control learning in Go programs.

3.1 Move-Average Sampling Technique

Move-Average Sampling Technique (MAST) [Finnsson and Björnsson, 2008] is the search-control method used by CADIAPLAYER when winning the AAAI 2008 GGP competition. It is somewhat related to the *history-heurstic* [Schaeffer, 1989], which is a well-established move-ordering mechanism in chess. The method learns search-control information during the back-propagations step, which it then uses in future playout steps to bias the random action selection towards choosing more promising moves. More specifically, when a return value of a simulation is backed up from T to S (see Fig. 1), then for each action a on the path a global (over all simulations) average for the action a is incrementally calculated, and kept in a lookup table $\mathcal{Q}_h(a)$. Moves found to be good on average, independent of a game state, will get higher values. The rational is that such moves are more likely to be good whenever they are available, e.g. placing a piece in one of the corner cells in Othello. In the playout step, the action selections are biased towards selecting such moves. This is done using Gibbs sampling as below:

$$\mathcal{P}(a) = \frac{e^{\mathcal{Q}_h(a)/\tau}}{\Sigma_{b=1}^{n} e^{\mathcal{Q}_h(b)/\tau}}$$

where $\mathcal{P}(a)$ is the probability that action a will be chosen in the current playout state and $\mathcal{Q}_h(a)$ is the average of all values backed up in any state when action a has been selected. This results in actions with a high $\mathcal{Q}_h(a)$ value becoming more likely to be chosen. One can stretch or flatten the above distribution using the τ parameter ($\tau \rightarrow 0$ stretches the distribution, whereas higher values make it more uniform).

3.2 Tree-Only MAST

Tree-Only MAST (TO-MAST) is a slight variation of MAST. Instead of updating the $\mathcal{Q}_h(a)$ for an entire simulation episode, it does so only for the part within the game tree (from state N back to S). This scheme is thus more selective about which action values to update, and because the actions in the tree are generally more informed than those in the playout part, this potentially leads to decisions based on overall a more robust and less variance data. In short this method prefers quality of data over sample quantity for controlling the search.

3.3 Predicate-Average Sampling Technique

Predicate-Average Sampling Technique (PAST) has a finer granularity of its generalization than the previous schemes. As the name implies, it uses the predicates encountered in the states to discriminate how to generalize.[1]

This method works as MAST except that now prediate-action pair values are maintained, $\mathcal{Q}_p(p, a)$, instead of action values $\mathcal{Q}_h(a)$. During the back-propagation, in a state s where action a was taken, $\mathcal{Q}_p(p, a)$ is updated for all $p \in P(s)$ where $P(s)$ is the set of predicates that are true in state s. In the playout step, an action is chosen as in MAST except that in the Gibbs sampling $\mathcal{Q}_h(a)$ is substituted with $\mathcal{Q}_p(p', a)$, where p' is the predicate in the state s with the maximum predicate-action value for a.

Whereas MAST concentrates on moves that are good on average, PAST can realize that a given move is good only in a given context, e.g. when there is a piece on a certain square. To ignore PAST values with unacceptably high variance, they are returned as the average game value until a certain threshold of samples is reached.

3.4 Rapid Action Value Estimation

Rapid Action Value Estimation (RAVE) [Gelly and Silver, 2007] is a method to speed up the learning process inside the game tree. In *Go* this method is known as *all-moves-as-first* heuristic because it uses returns associated with moves further down the simulation path to get more samples for duplicate moves available, but not selected, in the root state. When this method is applied to a tree structure as in MCTS the same is done for all levels of the tree. When backing up the value of a simulation, we update in the tree not only the value for the action taken, $\mathcal{Q}(s, a)$, but also sibling action values, $\mathcal{Q}_{RAVE}(s, a')$, if and only if action a' occurs further down the path being backed up (s to T).

As this presents bias into the average values, which is mainly good initially when the sampled data is still unreliable, these rapidly learned estimates should only be used for high variance state-action values. With more simulations the state-action averages $\mathcal{Q}(s, a)$ become more reliable, and should be trusted more than the RAVE value $\mathcal{Q}_{RAVE}(s, a)$. To accomplish this the method stores the RAVE value separately from the actual state-action values, and then weights them linearly as:

$$\beta\,(s) \times \mathcal{Q}_{RAVE}(s, a) + (1 - \beta\,(s)) \times \mathcal{Q}(s, a)$$

where

$$\beta\,(s) = \sqrt{\frac{k}{3n\,(s) + k}}$$

The parameter k is called the *equivalence parameter* and controls how many state visits are needed for both estimates to be weighted equal. The function $n(s)$ tells how many times state s has been visited.

4 Empirical Evaluation

We matched programs using the the four aforementioned search-control schemes against two baseline programs. They all share the same code base to minimize implementation-specific issues. The value of the UCT parameter C is set to

[1] A game positions, i.e. a state, is represented as a list of predicates that hold true in the state.

Table 1: Tournament against the MCTS agent.

Game	**MAST** win %	**TO-MAST** win %	**PAST** win %	**RAVE** win %
Checkers	54.83 (± 5.42)	80.67 (± 4.23)	61.33 (± 5.20)	61.50 (± 5.27)
Othello	58.67 (± 5.48)	54.00 (± 5.55)	61.33 (± 5.42)	57.17 (± 5.50)
Breakthrough	88.67 (± 3.59)	86.67 (± 3.85)	89.67 (± 3.45)	60.67 (± 5.54)

Table 2: Tournament against the MAST agent.

Game	**TO-MAST** win %	**PAST** win %	**RAVE** win %
Checkers	74.83 (± 4.64)	57.67 (± 5.27)	61.33 (± 5.16)
Othello	37.00 (± 5.35)	49.67 (± 5.51)	46.50 (± 5.51)
Breakthrough	49.33 (± 5.67)	42.33 (± 5.60)	13.00 (± 3.81)

40 (for perspective, possible game outcomes are in the range 0-100). The τ parameter of the Gibbs sampling in MAST and TO-MAST was set to 10, but for PAST it was set to 8. The sample threshold for a PAST value to used was set to 3, and the *equivalence parameter* for RAVE was set to 1000. These parameters are the best settings for each scheme, based on trial an error testing.

In the result tables that follow each data point represents the result of a 300-game match, with both a winning percentage and a 95% confidence interval shown. We tested the schemes on only two-player turn-taking games. The matches were run on Linux based dual processor Intel(R) Xeon(TM) 3.20GHz CPU computers with 2GB of RAM. Each agent used a single processor. The startclock and the playclock were both set to 10 seconds.

4.1 Individual Schemes

Table 1 shows the result when the four search-control schemes were individually matched against a base MCTS player, using UCT in the selection step but choosing actions uniformly at random in the playout step. All four schemes show an significant improvement against the base player on all three games where PAST seems to be doing overall the best, coming close to dominate all the others across all the games.

As the MAST scheme has been in use in CADIAPLAYER for some time, and as such represents the state-of-the-art in simulation search-control in GGP, we also matched the other schemes against CADIAPLAYER as a baseline player. The result in shown in Table 2. There are several points of interest. First of all, the other schemes improve upon MAST only in the game of Checkers. In the game Othello, PAST and RAVE also more or less holds their own against MAST and we cannot state with statistical significance that one is better than the other. However, in the game Breakthrough, MAST clearly outperforms both PAST and RAVE. This is maybe not of a surprise because the MAST scheme was originally motivated to overcome problems surfacing in that particular game. Of particular interest though is how badly RAVE is doing against MAST in this game, despite that is showed a significant improvement against the other base player.

Also of interest is to contrast TO-MAST's performance on different games. The only difference between MAST and TO-MAST is that the former updates action values in the entire playout episode, whereas the latter only updates action values when back-propagating values in the top part of the episode, that is, when in the tree. Doing so seems to have significant improvements in the game of Checkers, where it has decremental effects in the game of Othello against MAST. A possible explanation is that actions generalize better between states in different game phases in Othello than in Checkers, that is, an action judged good towards the end of the game is more often also good early on if available. For example, placing a piece on the edge of the board is typically always good and such actions, although not available early on in the game, start to accumulate credit right away.

Table 3: Tournament between MCTS and RAVE/MAST.

Game	**RAVE/MAST** win %
Checkers	62.83 (± 5.25)
Othello	66.83 (± 5.29)
Breakthrough	89.00 (± 3.55)

4.2 Combined Schemes

The MAST, TO-MAST, and PAST values are applied for action selection in the playout step only, whereas RAVE affects the action selection in the selection step only. It thus makes sense to try to combine RAVE with the others The results of a combined RAVE/MAST scheme playing against the same base players as in previous experiments is given in Tables 3 and 4.

The result shows that such a combined scheme offers overall a genuine improvement. For example, against the MCTS base player the combined schemes scores higher than either of the schemes individually on all three games. Also, against the MAST baseline player the combined scheme offers substantial benefits in Checkers and Breakthrough, although it appears to be slightly worse in Breakthrough (although within the significance confidence bounds). Again, we should keep in mind that MAST was specifically inspired to overcome drawbacks that simulations have in that game.

Table 4: Tournament between MAST and RAVE/MAST.

Game	**RAVE/MAST** win %
Checkers	74.50 (± 4.80)
Othello	60.33 (± 5.43)
Breakthrough	46.33 (± 5.65)

5 Related Work

One of the first general game-playing systems was Pell's METAGAMER [Pell, 1996], which played a wide variety of simplified chess-like games. CLUNEPLAYER [Clune, 2007] and FLUXPLAYER [Schiffel and Thielscher, 2007b; 2007a] were the winners of the 2005 and 2006 GGP competitions, respectively. UTEXAS LARG was also a prominent agent in those two competitions, and novel in that it used knowledge transfer to expedite the learning process. The aforementioned agents all use a traditional game-tree search with a learned heuristic evaluation function. The most recent version of CLUNEPLAYER also has a Monte-Carlo simulation module, and the agent decides at the beginning of a game which search approach to use [Clune, 2008]. Besides CADIAPLAYER, which won the 2007 and 2008 GGP competitions, two other strong agents were also simulation-based, ARY and MALIGNE.

As for learning simulation guidance in GGP the MAST scheme has already been presented [Finnsson and Björnsson, 2008]. Furthermore, in [Sharma *et al.*, 2008] a method to generate search-control knowledge for GGP agents based on both action and state predicate values is presented. The action bias consists of the sum of the action's average return value and the value of the state being reached. The value of the state is computed as the sum of all state predicate values, where the value of each state predicate is learned incrementally using a recency weighted average. Our PAST method learns state predicates as simple averages and uses them quite differently, in particular, we found that a bias based on a maximum state predicate value is more effective than using their sum.

Monte Carlo Tree Search (MCTS) has been used successfully to advance the state-of-the-art in computer Go, and is used by several of the strongest Go programs, e.g. MOGO [Gelly *et al.*, 2006] and CRAZYSTONE [Coulom, 2006]. Experiments in Go showing how simulations can benefit from using an informed playout policy are presented in [Gelly and Silver, 2007]. The method, however, requires game-specific knowledge which makes it difficult to apply to GGP. In the paper the authors also introduced RAVE. Progressive Strategies [Chaslot *et al.*, 2007] is another methods used by Go programs to improve simulation guidance in the MCTS's selection step. The UCT action selection is biased progressively from using mainly heuristic knowledge about states early on, towards using regular action return values once the sampling size has increased sufficiently. A method that progressively unprunes search trees with a large branching factor based on domain knowledge is also presented.

Search-control learning in traditional game-tree search has been studied for example in [Björnsson, 2002; Björnsson and Marsland, 2003].

6 Conclusions and Future Work

In this paper we empirically evaluate several search-control schemes for simulation-based GGP agents: MAST, TO-MAST, and PAST control action selection in the MCTS playout step, whereas RAVE biases the action selection in the MCTS selection step.

It is clear that the design and choice of a search-control scheme greatly affects the playing strength of an agent. By combining schemes that work on disparate parts of the simulation rollout, even further performance improvements can be gained, as we showed by RAVE/MAST. Also, it is important to consider both where the learning experiences come from (e.g. the performance difference of MAST vs. TO-MAST), and how they are generalized. For example, the PAST scheme is capable of generalizing based on context, and that gives significant benefits in some games. Overall, none of the schemes is dominating in the sense that of improving upon all the others on all three test-bed games. Not surprisingly, the diverse properties of the different games favor some schemes more than others.

For future work there are still many interesting research avenues to explore for further improving simulation-based search control. All the methods we present here do not, or only very indirectly, take the game structure into account. Agents that infer game-specific properties from a game description, could potentially use better informed schemes for generalizing the learned search-control knowledge. For example, an agent that explicitly understands that a move is a capture move, could potentially learn more quickly whether captures are in general beneficial for the game at hand (PAST kind of learns this, but implicitly over time). There is also specific work that can be done to further improve the schemes discussed, for example, currently PAST introduces considerable overhead in games where states contain many predicates, resulting in up to 10-20% slowdown. By updating the predicates more selectively we believe that most of this overhead can be eliminated while still maintaining the benefits, likely resulting in PAST subsequently dominating the other individual schemes. Also, we do not fully understand which game properties determine why MAST is so much better than TO-MAST on certain games but clearly worse on others. The only difference between the two schemes is that the former learns from an entire episode whereas the latter learns only from the beginning of an episode. This difference is interesting and warrants a further investigation. There is also scope for combining the search-control schemes differently — one possibility that comes to mind is PAST and RAVE. Finally, we believe that there is still room for improvements by simply tuning the various different parameters used by the methods, especially if one could automatically tailor them to the game at hand.

Acknowledgments

This research was supported by grants from The Icelandic Centre for Research (RANNÍS) and by a Marie Curie Fellowship of the European Community programme *Structuring the ERA* under contract MIRG-CT-2005-017284.

References

[Abramson, 1990] B. Abramson. Expected-outcome: A general model of static evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(2):182–193, 1990.

[Atkin and Slate, 1988] L. Atkin and D. Slate. Chess 4.5-the northwestern university chess program. *Computer chess compendium*, pages 80–103, 1988.

[Björnsson and Finnsson, 2009] Yngvi Björnsson and Hilmar Finnsson. Cadiaplayer: A simulation-based general game player. *IEEE Transactions on Computational Intelligence and AI in Games*, in press, 1(1), 2009.

[Björnsson and Marsland, 2003] Yngvi Björnsson and T. A. Marsland. Learning extension parameters in game-tree search. *Information Sciences*, 154(3-4):95–118, 2003.

[Björnsson, 2002] Yngvi Björnsson. *Selective Depth-First Game-Tree Search*. PhD dissertation, University of Alberta, Canada, Department of Computing Science, 2002.

[Bouzy and Helmstetter, 2003] B. Bouzy and B. Helmstetter. Monte-Carlo Go Developments. In H.J. van den Herik, H. Iida, and E.A. Heinz, editors, *Advances in Computer Games 10: Many Games, Many Challenges*, pages 159–174. Kluwer Academic Publishers, Boston, MA, USA, 2003.

[Breuker, 1998] Dennis M. Breuker. *Memory versus Search in Games*. PhD dissertation, Maastricht University, Department of Computing Science, 1998.

[Brügmann, 1993] B. Brügmann. Monte Carlo Go. Technical report, Physics Department, Syracuse University, 1993.

[Buro, 1999] Michael Buro. How machines have learned to play Othello. *IEEE Intelligent Systems*, 14(6):12–14, November/December 1999. Research Note.

[Campbell *et al.*, 2002] Murray Campbell, A. Joseph Hoane, Jr., and Feng-Hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1–2):57–83, 2002.

[Chaslot *et al.*, 2007] G. M. J. B. Chaslot, Winands M. H. M. Winands, J. W. H. M. Uiterwijk, H. J. van den Herik, and B. Bouzy. Progressive strategies for Monte-Carlo tree search. Draft, submitted to JCIS workshop 2007, 2007.

[Clune, 2007] James Clune. Heuristic evaluation functions for General Game Playing. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, pages 1134–1139, 2007.

[Clune, 2008] James Edmond Clune. *Heuristic Evaluation Functions for General Game Playing*. PhD dissertation, University of California, Los Angeles, Department of Computer Science, 2008.

[Coulom, 2006] Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *The 5th International Conference on Computers and Games (CG2006)*, pages 72–83, 2006.

[Finnsson and Björnsson, 2008] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 259–264. AAAI Press, 2008.

[Finnsson, 2007] Hilmar Finnsson. CADIA-Player: A General Game Playing Agent. Master's thesis, Reykjavík University, December 2007.

[Gelly and Silver, 2007] Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In Zoubin Ghahramani, editor, *ICML*, volume 227, pages 273–280. ACM, 2007.

[Gelly *et al.*, 2006] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA, 2006.

[Knuth and Moore, 1975] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

[Kocsis and Szepesvári, 2006] Levante Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *European Conference on Machine Learning (ECML)*, pages 282–293, 2006.

[Kuhlmann *et al.*, 2006] Gregory Kuhlmann, Kurt Dresner, and Peter Stone. Automatic heuristic construction in a complete general game player. In *Proc. of the Twenty-First National Conference on Artificial Intelligence*, pages 1457–62, July 2006.

[Pell, 1996] Barney Pell. A strategic metagame player for general chess-like games. *Computational Intelligence*, 12:177–198, 1996.

[Schaeffer, 1989] Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-11(11):1203–1212, 1989.

[Schaeffer, 1997] Jonathan Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer-Verlag New York, Inc., 1997.

[Schiffel and Thielscher, 2007a] Stephan Schiffel and Michael Thielscher. Automatic construction of a heuristic search function for General Game Playing. In *Seventh IJCAI International Workshop on Nonmontonic Reasoning, Action and Change (NRAC07)*, 2007.

[Schiffel and Thielscher, 2007b] Stephan Schiffel and Michael Thielscher. Fluxplayer: A successful general game player. In *Proc. of the Twenty-Second AAAI Conference on Artificial Intelligence*, pages 1191–1196, 2007.

[Sharma *et al.*, 2008] Shiven Sharma, Ziad Kobti, and Scott Goodwin. Knowledge generation for improving simulations in UCT for general game playing. In *AI 2008: Advances in Artificial Intelligence*, pages 49–55. Springer, 2008.