Learning 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 based on both statistical and reinforcement learning techniques. We compare the different schemes empirically on a variety of games and show that they improve significantly upon the current state-of-the-art in simulation-control in GGP. For example, in the chess-like game Skirmish, which has proved a particularly challenging game for simulation-based GGP agents, an agent employing one of the proposed schemes achieves 97% winning rate against an unmodified agent.

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. These highly specialized game-playing system are engineered and optimized towards playing the particular game in question.

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, involving 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. The most successful GGP players used to be based on that approach (Schiffel and Thielscher 2007; Clune 2007; Kuhlmann, Dresner, and Stone 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, automatically 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).

In this paper we investigate several domain-independent search-control learning mechanisms in CADIAPLAYER. The main contributions are new search-control learning schemes for GGP agents as well as an extensive empirical evalua-

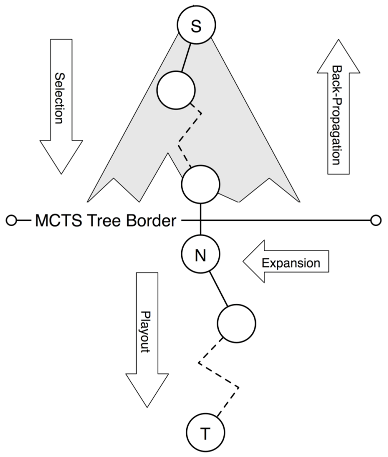

Figure 1: An overview of a single simulation.

tion of different search-control mechanisms, both old and new. There is no single scheme 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 its implementation in CADIAPLAYER. Next we introduce several different search-control mechanism, which we then empirically evaluate using four different games. Finally, we survey related work before concluding and discussing future work.

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 a 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 $\mathcal{Q}(s, a)$ 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 estimate with some confidence, the bonus of the suboptimal ones may have increased enough for them to get picked again for further exploration. If a suboptimal action is found 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. The C parameter is used to tune how much influence the UCT bonus has on the action selection calculations.

In the playout step there are no $\mathcal{Q}(s, a)$ values available for guiding the action selection, so in the most straightforward 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. A typical strategy is to append only one new node to the tree in each simulation: the first node encountered after stepping out of the tree (Coulom 2006). This is done to avoid using excessive memory, in particular if simulations are fast. In Fig. 1 the newly added node in this episode is labeled as N.

For details on the MCTS search in CADIAPLAYER see (Finnsson 2007; Björnsson and Finnsson 2009).

Search Controls

In this section we describe five search-control mechanisms 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 sophisticated about how it generalizes the learned control information. The fourth method, RAVE, is an established technique to expedite search-control learning in Go programs. Finally we describe a new method, FAST, that uses temporal-difference learning (Sutton 1988) to generate knowledge for search control.

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. 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, $\mathcal{Q}_h(a)$, is incrementally calculated and kept in a lookup table. 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 the Gibbs (or Boltzmann) distribution 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 ($\tau \to 0$ stretches the distribution, whereas higher values make it more uniform).

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 more robust and less variance data. In short this method prefers quality of data over sample quantity for controlling the search.

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 predicate-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 $\mathcal{P}(a)$ distribution $\mathcal{Q}_h(a)$ 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.

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

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.

Features-to-Action Sampling Technique

The aforementioned schemes use generic concepts such as actions and predicates as atomic features for their learning. This seems simplistic in contrast to human players, which also use high-level features such as piece types and board geometry in their reasoning. Although it has the benefit of allowing effective deployment over a large range of disparate games, the lack of understanding of high-level game concepts does severely handicap GGP players in certain types of games, for example chess-like games where a good understanding of the different piece type values is essential for competitive play. GDL, the language for writing GGP game descriptions, does not allow for pieces or board geometry to be explicitly stated; nonetheless, with careful analysis such concepts can be inferred from game descriptions. Now we describe a new search-control learning scheme that can be used in such cases.

With *Features-to-Action Sampling Technique* (FAST) we use template matching to identify common board game features, currently detecting two such: piece types and cells (squares). Piece type is the dominant feature set, but it is only judged relevant if it can take on more than one value;

if not, we instead consider cell locations as our feature set. We use TD(λ) (Sutton 1988) to learn the relative importance of the detected features, e.g. the values of the different type of pieces or the value of placing a piece on a particular cell. Each simulation, both on the start- and play-clock, generates an episode $s_1 \rightarrow s_2 \rightarrow ... \rightarrow s_n$ that the agent learns from by applying the delta rule to each state s_t in the episode:

$$\vec{\delta} = \vec{\delta} + \alpha[R_t^\lambda - V(s_t)] \bigtriangledown_{\vec{\theta}} V(s_t)$$

where R_t^λ is the λ-return (average of exponentially weighted n-step TD returns), $V(s)$ is our value function, and $\bigtriangledown_{\vec{\theta}} V(s)$ is its gradient. A reward is given at the end of the episode, as the difference of the players' goal values. The $\vec{\delta}$ is then used in between episodes to update the weight vector $\vec{\theta}$ used by the value function to linearly weigh and combine the detected features $\vec{f}(s)$:

$$V(s) = \sum_{i=1}^{|\vec{f}|} \theta_i * f_i(s)$$

In games with different piece types, each feature $f_i(s)$ represents the number of pieces of a given type in state s (we do not detect piece symmetry, so a white rook is considered different from a black one). In games where cell-based features are instead detected each feature is binary, telling whether a player has a piece in a given cell (i.e. a two-player game with N cells would result in $2N$ features).

The value function is not used directly to evaluate states in our playouts. Although that would be possible, it would require executing not only the actions along the playout path, but also all sibling actions. This would cause a considerable slowdown as executing actions is a somewhat time consuming operation in GGP . So, instead we map the value function into the same $Q(a)$ framework as used by the other schemes. This is done somewhat differently depending on type of detected features and actions. For example, for piece-type features in games where pieces move around the mapping is:

$$Q(a) = \begin{cases} -(2 * \theta_{Pce(to)} + \theta_{Pce(from)}), & \text{if capture move} \\ -100 & \text{otherwise} \end{cases}$$

where $\theta_{Pce(from)}$ and $\theta_{Pce(to)}$ are the learned values of the pieces on the $from$ and to squares, respectively. This way capture moves get added attention when available and capturing a high ranking piece with a low ranking one is preferred. For the cell features the mapping is:

$$Q(a) = c * \theta_{p,to}$$

where $\theta_{p,to}$ is the weight for the feature of player p having a piece on square to and c is a positive constant. Now that we have established a way to calculate $Q(a)$ the $\mathcal{P}(a)$ distribution may be used in the same way as for the other schemes to choose between actions.

Empirical Evaluation

We matched programs using the aforementioned search-control schemes against two baseline programs. They all share the same code base to minimize implementation-specific issues. One important difference between our current baseline agents and the ones used in our previous studies, is that they now have a more effective transposition table. The value of the UCT parameter C is set to 40 (for perspective, possible game outcomes are in the range 0-100). The τ parameter of the $\mathcal{P}(a)$ distribution in MAST, TO-MAST and PAST is set to 10, but to 1 for FAST. The sample threshold for a PAST value to be used is set to 3, and the *equivalence parameter* for RAVE is set to 1000. In FAST the λ parameter is set to 0.99, the step-size parameter α to 0.01, and c to 5. These parameters are the best known settings for each scheme, based on trial and error testing.

In the tables that follow, each data point represents the result of a either a 300-game (first two tables) or 200-game (latter two tables) match, with both a winning percentage and a 95% confidence interval shown. We tested the schemes on four different two-player turn-taking games: Checkers, Othello, Breakthrough, and Skirmish[2]. They were chosen because some variants of them are commonly featured in the GGP competitions, as well as some have proved particularly challenging for simulation-based agents to play well. The matches were run on Linux based 8 processor Intel(R) Xeon(R) E5430 2.66GHz CPU computer with 4GB of RAM. Each agent used a single processor. The startclock and the playclock were both set to 10 seconds.

Individual Schemes

Table 1 shows the result from matching the five search-control schemes individually against a base MCTS player. The base player uses UCT in the selection step and chooses actions uniformly at random in the playout step. All five schemes show an significant improvement over the base player in all games except Skirmish (with the exception that FAST has no effect in Checkers simply because the learning scheme was not initiated by the FAST agent as no template matched the way captures are done in that game). In the game of Skirmish, however, the FAST agent does extremely well, maybe not that surprising given that it was devised for use in such types of games.

As the MAST scheme has been in use in CADIAPLAYER for some time, and as such represented the state-of-the-art in simulation search-control in GGP in 2008, 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 first three schemes improve upon MAST in the game of Checkers. In the game Othello, RAVE and FAST also more or less hold their own against MAST. However, in the game Breakthrough, MAST clearly outperforms both RAVE and FAST. This is not of a surprise because the MAST scheme was originally motivated to overcome problems surfacing in that particular game.

It is also of interest to contrast TO-MAST's performance on different games. The only difference between MAST and

[2]Skirmish is a chess-like game. We used the variation played in the 2007 GGP finals where each player has two knights, two bishops, two rooks, and four pawns (can only move by capturing). The objective is to capture as many of the opponent pieces as possible.

Table 1: Tournament against the MCTS agent.

Game	**MAST** win %	**TO-MAST** win %	**PAST** win %	**RAVE** win %	**FAST** win %
Breakthrough	90.00 (± 3.40)	85.33 (± 4.01)	85.00 (± 4.05)	63.33 (± 5.46)	81.67 (± 4.39)
Checkers	56.00 (± 5.37)	82.17 (± 4.15)	57.50 (± 5.36)	82.00 (± 4.08)	50.33 (± 5.36)
Othello	60.83 (± 5.46)	50.17 (± 5.56)	67.50 (± 5.24)	70.17 (± 5.11)	70.83 (± 5.10)
Skirmish	41.33 (± 5.18)	48.00 (± 5.29)	42.33 (± 5.16)	46.33 (± 5.30)	96.33 (± 1.86)

Table 2: Tournament against the MAST agent.

Game	**TO-MAST** win %	**PAST** win %	**RAVE** win %	**FAST** win %
Breakthrough	52.33 (± 5.66)	45.67 (± 5.65)	20.33 (± 4.56)	39.67 (± 5.55)
Checkers	82.00 (± 4.18)	55.83 (± 5.35)	78.17 (± 4.36)	46.17 (± 5.33)
Othello	40.67 (± 5.47)	49.17 (± 5.60)	58.17 (± 5.49)	56.83 (± 5.55)
Skirmish	56.00 (± 5.31)	43.33 (± 5.26)	59.83 (± 5.15)	97.00 (± 1.70)

Table 3: Tournament: MCTS vs. Scheme Combinations.

Game	**RM** win %	**RF** win %
Breakthrough	89.00 (± 4.35)	76.50 (± 5.89)
Checkers	84.50 (± 4.78)	77.00 (± 5.37)
Othello	79.75 (± 5.52)	81.00 (± 5.32)
Skirmish	45.00 (± 6.55)	96.00 (± 2.34)

Table 4: Tournament: MAST vs. Scheme Combinations.

Game	**RM** win %	**RF** win %
Breakthrough	50.50 (± 6.95)	38.50 (± 6.76)
Checkers	83.50 (± 4.87)	74.00 (± 5.81)
Othello	73.75 (± 6.01)	66.00 (± 6.43)
Skirmish	53.00 (± 6.47)	97.00 (± 2.04)

TO-MAST is that the former updates action values in the entire 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. TO-MAST significantly improves upon MAST in the game of Checkers, whereas it has decremental effects in the game of Othello. 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.

PAST seems to have an advantage on MAST in only Checkers and then just slightly, but the main problem with PAST is its overhead resulting in fewer simulations per second; it seems that we do not gain enough from the better information to offset the simulation loss. FAST still clearly remains the best scheme to use for Skirmish.

Combined Schemes

The MAST, TO-MAST, PAST, and FAST values are for action selection in the playout step, whereas RAVE's action selection is for the selection step. It thus makes sense to try to combine RAVE with the others. The results of a combined RAVE/MAST (RM) and RAVE/FAST(RF) schemes playing against the same base players as in previous experiments are given in Tables 3 and 4. The result shows that combining schemes this way offers overall a genuine improvement over using the schemes individually. Especially noteworthy are the improvement gains in Othello.

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 2007) 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, MALIGNE and ARY (which won the 2009 competition).

As for learning simulation guidance in GGP the MAST scheme was proposed in (Finnsson and Björnsson 2008). A preliminary comparison study of some of the search-control schemes discussed here is found in (Finnsson and Björnsson 2009), whereas FAST and RF are first introduced here. In (Sharma, Kobti, and Goodwin 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 their sum. Most recently in (Kirci, Schaeffer, and Sturtevant 2009) a method is introduced that generates features for the game being played by examining state differences to pinpoint predicates that result in moves beneficial towards winning or for fending off attacks. The move and the relevant predicates constitute a feature. During the playout the agent scans for these features and if present uses them to guide the search.

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, including MoGo (Gelly et al. 2006) and Fuego (Enzenberger and Müller 2009). 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 in GGP. The paper also introduced RAVE.

Conclusions and Future Work

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

It is clear that the design and choice of a search-control scheme greatly affects the playing strength of a GGP agent. By combining schemes that work on disparate parts of the simulation rollout, further performance improvements can be gained, as showed by RAVE/MAST and RAVE/FAST. It is also 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 of improving upon all the others on all four test-bed games. Not surprisingly, the diverse properties of the different games favor some schemes more than others. Also, agents that infer game-specific properties from a game description, like FAST, offer great potentials for games that have until now been known to be problematic for the MCTS approach.

For future work there are still many interesting research avenues to explore for further improving simulation-based search control. For example, there is still a lot of work needed to further improve FAST-like schemes, e.g. for them to understand a much broader range of games and game concepts. Nonetheless, looking at the finals of the 2007 and 2008 GGP competitions (which our agent won), FAST would have detected feature candidates in all the games. There is also specific work that can be done to further improve the other schemes discussed, for example, currently PAST introduces considerable overhead in games where states contain many predicates, resulting in up to 20% slowdown. By updating the predicates more selectively we believe that most of this overhead can be eliminated while still maintaining the benefits. There is also scope for combining the search-control schemes differently: one promising possibility that comes to mind is to combine FAST and MAST. Finally, we believe that there is still room for improvements by 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).

References

Abramson, B. 1990. Expected-outcome: A general model of static evaluation. *IEEE Trans. PAMI* 12(2):182–193.

Björnsson, Y., and Finnsson, H. 2009. Cadiaplayer: A simulation-based general game player. *IEEE Trans. on Computational Intelligence and AI in Games* 1(1):4–15.

Bouzy, B., and Helmstetter, B. 2003. Monte-Carlo Go Developments. In van den Herik, H.; Iida, H.; and Heinz, E., eds., *Advances in Computer Games 10: Many Games, Many Challenges*, 159–174. Kluwer, Boston, MA, USA.

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

Clune, J. 2007. Heuristic evaluation functions for General Game Playing. In *22nd AAAI*, 1134–1139.

Clune, J. E. 2008. *Heuristic Evaluation Functions for General Game Playing*. PhD dissertation, University of California, Los Angeles, Department of Computer Science.

Coulom, R. 2006. Efficient selectivity and backup operators in Monte-Carlo tree search. In *CG2006*, 72–83.

Enzenberger, M., and Müller, M. 2009. Fuego - an open-source framework for board games and go engine based on monte-carlo tree search. Technical Report 09-08, Dept. of Computing Science, University of Alberta.

Finnsson, H., and Björnsson, Y. 2008. Simulation-based approach to general game playing. In *23rd AAAI*, 259–264.

Finnsson, H., and Björnsson, Y. 2009. Simulation control in general game playing agents. In *GIGA'09 The IJCAI Workshop on General Game Playing*.

Finnsson, H. 2007. CADIA-Player: A General Game Playing Agent. Master's thesis, Reykjavík University.

Gelly, S., and Silver, D. 2007. Combining online and offline knowledge in UCT. In Ghahramani, Z., ed., *ICML*, volume 227, 273–280. ACM.

Gelly, S.; Wang, Y.; Munos, R.; and Teytaud, O. 2006. Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA.

Kirci, M.; Schaeffer, J.; and Sturtevant, N. 2009. Feature learning using state differences. In *GIGA'09 The IJCAI Workshop on General Game Playing*.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In *ECML*, 282–293.

Kuhlmann, G.; Dresner, K.; and Stone, P. 2006. Automatic heuristic construction in a complete general game player. In *21st AAAI*, 1457–62.

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

Schiffel, S., and Thielscher, M. 2007. Fluxplayer: A successful general game player. In *22nd AAAI*, 1191–1196.

Sharma, S.; Kobti, Z.; and Goodwin, S. 2008. Knowledge generation for improving simulations in UCT for general game playing. In *AI 2008: Advances in Artificial Intelligence*. Springer. 49–55.

Sutton, R. S. 1988. Learning to predict by the methods of temporal differences. *Machine Learning* 3:9–44.