Künstl Intell (2011) 25: 9–16
DOI 10.1007/s13218-010-0080-9

FACHBEITRAG

CadiaPlayer: Search-Control Techniques

Hilmar Finnsson · Yngvi Björnsson

Received: 31 July 2010 / Accepted: 11 September 2010 / Published online: 11 January 2011

Abstract Effective search control is one of the key components of any successful simulation-based game-playing program. In General Game Playing (GGP), learning of useful search-control knowledge is a particularly challenging task because it must be done in real-time during online play. In here we describe the search-control techniques used in the 2010 version of the GGP agent CADIAPLAYER, and show how they have evolved over the years to become increasingly effective and robust across a wide range of games. In particular, we present a new combined search-control scheme (RAVE/MAST/FAST) for biasing action selection. The scheme proves quite effective on a wide range of games including chess-like games, which have up until now proved quite challenging for simulation-based GGP agents.

Keywords General game playing · Cadiaplayer · Monte Carlo Tree Search · Search control

1 Introduction

CADIAPLAYER was the first General Game Playing (GGP) agent to use Monte-Carlo Tree Search (MCTS) for reasoning about its actions, an approach that has now become mainstream in GGP agents. However, in contrast to many of its contemporary MCTS-based players, which empathize massive search parallelism, the mainstream research in CADIAPLAYER has been on developing practical ways of extracting search-control knowledge during online play as well as developing informed search-control mechanisms for effectively using such knowledge in real-time. Using such an approach it won the 2007 and 2008 International GGP competitions and ended in the third place in 2010.

An effective search-control mechanism for simulation playouts is a key component in any MCTS-based game-playing program. In many game application domains where MCTS is successfully used, such as computer Go [7, 11] and Amazons [18], a typical approach is to manually define and encode game-specific playing patterns and other kind of domain knowledge that is useful for search control and then carefully tune its use, either by hand or by automated match play consisting of (possibly) thousands of games. In contrast, in GGP the search-control knowledge must be automatically discovered, extracted, represented, and tuned during online play consisting of (possibly) only a single match game—making the task a hand all the more challenging.

In this paper we describe the MCTS reasoning component of the 2010 version of CADIAPLAYER. The main contributions are: (1) a state-of-the-art online simulation search-control mechanism, which introduces a new technique for handling chess-like games as well as a novel integration with existing schemes; (2) empirical evaluation of the new search-control mechanism on a wide range of games and analysis of its strengths and weaknesses depending on several game properties; and (3) an overview of CADIAPLAYER's performance evolution since its first competition, thus providing one benchmark of how the field of GGP has advanced over the years.

The paper is structured as follows. In the next section we give an overview of the MCTS approach and its implementation in CADIAPLAYER. We next describe the search-control techniques used in CADIAPLAYER, including how they obtain and apply the relevant knowledge. We then empirically evaluate the techniques on a wide range of games, includ-

H. Finnsson · Y. Björnsson (✉)
Reykjavík University, Menntavegur 1, 101 Reykjavík, Iceland
e-mail: yngvi@ru.is

H. Finnsson
e-mail: hif@ru.is

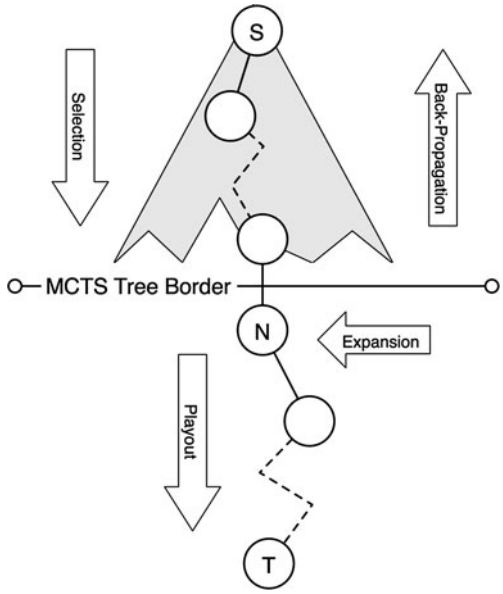

Fig. 1 An overview of a single simulation

ing some from the 2010 GGP competition. In particular, we examine how CADIAPLAYER has evolved in strength over the years as more sophisticated search-control schemes have been incorporated. Related work is then discussed before concluding and contemplating future work.

2 MCTS General Game Player

Monte-Carlo Tree Search (MCTS) is at the core of CADIAPLAYER's reasoning engine. 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, $Q(s,a)$. When the deliberation time is up, the method plays the action at the root of the tree with the highest average return value.

Figure 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 and chooses actions while still in the tree (upper half of figure), while the playout step chooses 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 $Q(s,a)$ as well as other relevant information if applicable.

The *Upper Confidence Bounds applied to Trees (UCT)* algorithm [15] is 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:

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

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 $Q(s,a)$ 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 $Q(s,a)$ values available for guiding the action selection, so in the most straightforward case one would choose between those available uniformly at random. However, CADIAPLAYER uses more sophisticated techniques 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 [5]. This is done to avoid using excessive memory, in particular if simulations are fast. In Fig. 1 the node added in this episode is labeled as N.

3 Simulation Control

The search-control mechanism used in CADIAPLAYER for guiding the MCTS simulations combines three different schemes. The first one, MAST, was introduced in the 2008 version of CADIAPLAYER; the next one, RAVE, was added in the 2009 version and is borrowed from the computer Go community where it is an established technique to expedite search-control learning; and the third one, FAST, is a newly added scheme that uses temporal-difference learning to learn board-specific domain-knowledge for search control. In the 2010 version of CADIAPLAYER the three

schemes were used collectively, giving an effective and robust search-control guidance on a wide range of disparate games.

We first describe the general search-control framework used in CADIAPLAYER, followed by each of the individual schemes, and finally we show how the schemes can be combined.

3.1 Search-Control Framework

In CADIAPLAYER a single simulation plays out in two phases with a different search-control mechanism for each: first, in the selection step, actions are chosen according to the (possibly modified) UCT formula, and second, in the playout step, actions are chosen according to the Gibbs (or Boltzmann) distribution:

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

where $\mathcal{P}(a)$ is the probability that action a will be chosen in the current playout state and $\mathcal{Q}_h(a)$ is a merit-based value telling how promising move a is—actions with a higher $\mathcal{Q}_h(a)$ value being more likely to be chosen. This is a general and convenient framework for allowing non-uniform random action selection as the bias introduced by the merit values can be controlled by tuning the τ parameter. When $\tau \to 0$ the distribution stretches, whereas higher values make it converge to an uniform one.

3.2 Move-Average Sampling Technique

Move-Average Sampling Technique (MAST) [8] learns search-control information during the back-propagation step using the values of the simulation nodes both outside the tree as well as those within it. The information is then used to bias future playout steps towards more promising actions, using the aforementioned Gibbs distribution. This is done by setting up a lookup table over all actions independent of the states they are available in. When a return value of a simulation is backed up from T to S (see Fig. 1), the table is updated with incrementally calculated averages, $\mathcal{Q}_h(a)$, for each action a on the path. The rational is that some actions are likely to be good whenever they are available, e.g. placing a piece in the center cell in TicTacToe or one of the corner cells in Othello.

One of the main attractions of this scheme is its simplicity and generality, allowing useful search-control information to be efficiently computed in a game independent manner. The scheme's effectiveness has proved quite robust across a wide range of games.

3.3 Rapid Action Value Estimation

Rapid Action Value Estimation (RAVE) [10] is a method borrowed from computer Go, which speeds up the learning process in the game tree. It uses returns associated with actions further down the simulation path to get more samples for duplicate actions available, but not selected, higher up in the tree. The method stores its values, $\mathcal{Q}_{\text{RAVE}}(s,a)$, separately from the actual MC state-action averages, $\mathcal{Q}(s,a)$, so when backing up the values of a simulation, we not only update the tree value for the action taken, $\mathcal{Q}(s,a)$, but also sibling action values, $\mathcal{Q}_{\text{RAVE}}(s,a')$, if and only if action a' occurs further down the path being backed up (s to T).

We use the $\mathcal{Q}_{\text{RAVE}}(s,a)$ to bias the $\mathcal{Q}(s,a)$ value in the UCT formula in the selection step. These rapidly learned estimates are mainly good initially when the sampled data is still unreliable and should only be used in conjunction with high variance $\mathcal{Q}(s,a)$ values. With more simulations the $\mathcal{Q}(s,a)$ averages become more reliable and should be trusted more than the $\mathcal{Q}_{\text{RAVE}}(s,a)$ values. This is accomplished by replacing the $\mathcal{Q}(s,a)$ term with the following formula that weighs the two values linearly based on number of samples collected:

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

where

$$\beta(s) = \sqrt{\frac{k}{3 \times N(s) + k}}$$

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.

3.4 Features-to-Action Sampling

The aforementioned schemes do not use any game-specific domain knowledge. Although this has the benefit of allowing effective deployment over a wide range of disparate games, this approach seems simplistic in contrast to human players, which use high-level features such as piece types and board geometry in their reasoning. The lack of understanding of such high-level game concepts does indeed severely handicap GGP players using simple search-control schemes 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. Although GDL does not explicitly represents items such as pieces and boards such game-specific concepts can often be inferred.

With the *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 only judged relevant if it can take on more than one value; if not, we consider cell locations as our feature set. We use TD(λ) [26] to learn the relative importance of the detected features, e.g. the values of the different types 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 $s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_n$ that the agent learns from by applying the delta rule to each state s_t in the episode:

$$\boldsymbol{\delta} = \boldsymbol{\delta} + \alpha \times [R_t^{\lambda} - V(s_t)] \times \nabla_{\boldsymbol{\theta}} V(s_t)$$

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

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

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. Instead we map the value function into the same $Q_h(a)$ framework as used by MAST. This is done differently depending on type of detected features and actions. For example, for piece-type features in games where pieces move around the mapping is:

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

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:

$$Q_h(a) = c \times \theta_{p,to}$$

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

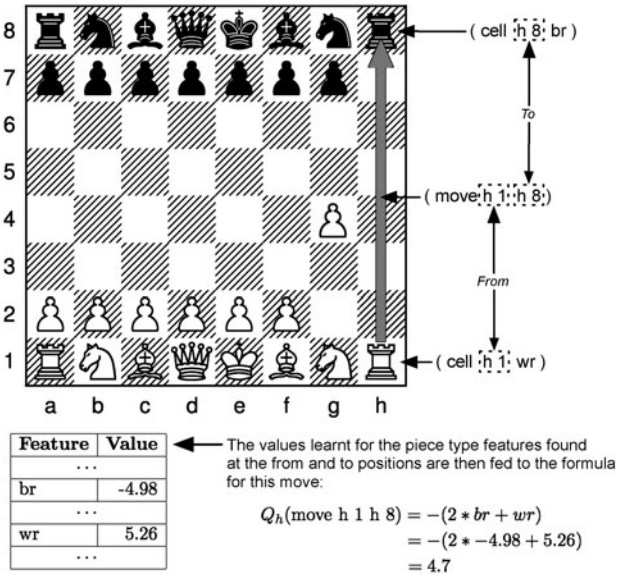

Fig. 2 FAST capture calculations in chess for (move h 1 h 8) in a state containing (cell h 1 wr) and (cell h 8 br)

We look at a concrete example in Fig. 2 of how piece type features are used. In chess and similar board games a common encoding practice in GDL for representing a board configuration with state predicates will take on the general form (or variation thereof)

```
(<predicate-name><column>
 <row><piece>)
```

for each piece on the board. The name of the predicate may vary but a common feature is that two of its arguments indicate the cell location and the remaining one the piece type currently occupying the cell. For example, (*cell h 1 wr*) indicates a white rook on square `h1`. Similarly, actions in games where pieces move around are commonly encoded using the recurring format of indicating the from and to cell locations of the moving piece, e.g. (*move h 1 h 8*). This way both different piece types and piece captures can be identified. As soon as the program starts running its simulations, the TD-learning episodes provide learned values for the different piece types. For example, in chess we would expect the winning side to be material up more often and our own pieces thus getting positive values and the opponent's pieces negative ones. The more powerful pieces—such as queens and rooks—have higher absolute values (e.g., in Skirmish, a simplified chess-like game used in our experiments, the values typically learned for a pawn, knight, bishop, and rook were approximately 5, 10, 10, and 13, respectively). The learned piece values θ are stored in a lookup table and consulted when captures moves are made, as depicted in Fig. 2

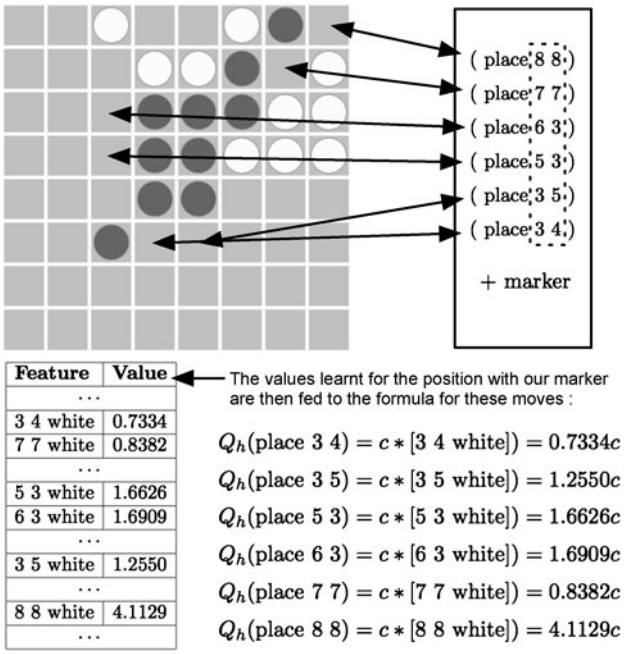

Fig. 3 FAST calculations in Othello for available moves

for the move *rook h1 captures on h8*. The more powerful the captured opponent's piece is, especially when captured with a low ranked piece, the higher the $Q_h(a)$ becomes and thus the likelihood that the move will be played. The learned piece values are constantly updated throughout the game, however, as a precaution, they are not used unless having a value far enough from zero.

In games with only one type of piece for each role, the piece locations become active features instead of piece types. This is shown for the game Othello in Fig. 3, where (3 4 *white*) is an example feature (i.e., a white disk on cell 3 4). During the playout step, in each state, we look up the $Q_h(a)$ values for all available actions in the feature table by their associated role and location, and bias play towards placing disks onto cells with high learned values, such as corners and edges.

3.5 Combining Schemes

The RAVE scheme can be easily combined with either MAST or FAST as it operates in the selection step of a simulation, opposed to in the playout step as the other two. The RAVE/MAST combination was implemented and used in CADIAPLAYER 2009.

The FAST scheme was devised later, but was not as easily integrated into the already existing combined RAVE/MAST scheme. The problem is that MAST and FAST both operate on the playout step and possibly bias the action selection differently. We opted for the following MAST/FAST integration in the playout step:

$$Q_h(a) = \begin{cases} Q_{\text{MAST}}(a) + w \times Q_{\text{FAST}}(a) & \text{if any features} \\ & \text{active in } A(s) \\ Q_{\text{MAST}}(a) & \text{otherwise} \end{cases}$$

where $Q_{\text{MAST}}(a)$ and $Q_{\text{FAST}}(a)$ are the $Q_h(a)$ values as calculated by the MAST and FAST schemes respectively, and the w parameter is a weighing constant deciding on their relative importance. If no features are active in the current state, meaning that either no features were detected in the game description or that no capture moves are available in such a game, the $Q_{\text{FAST}}(a)$ distribution becomes uniform and is omitted (as it would shift the final distribution without adding any information).

For maximum efficiency the influence of each scheme must be carefully weighted, possibly varying depending on the game type. For the 2010 competition a combined RAVE/MAST/FAST scheme was used, however, we simply fixed $w = 1.0$ because of a lack of time for thoroughly tuning a more appropriate value.

4 Empirical Evaluation

The aforementioned search-control schemes were developed and added to CADIAPLAYER at different time points, often in a response to an observed inefficiency of the agent in particular types of games. The main difference in the agent between subsequent GGP competitions has thus been the sophistication level of the search control. Although there have been other more minor improvements made to the agent from year to year—e.g., in the form of a more effective transposition table, implementation efficiency improvements, and better tuned parameter settings—in here we are interested in quantifying the effects of the increasingly more effective search-control schemes. We nullify the effect of other changes by having all agent versions share the newest and most effective code base, thus differing only in the type of search-control scheme used.

4.1 Setup

We empirically evaluate and contrast four versions of CADIAPLAYER, each representing the type of search-control used in the four GGP competitions the agent has competed in (it placed 1st in 2007 and 2008, 6th in 2009, and 3rd in 2010). The 2007 version is used as a baseline player that all other versions are matched against. In the table that follows, each data point represents the result of a 200-game match, showing both a win percentage and a 95% confidence interval. The matches were run on Linux based 8 processor Intel(R) Xeon(R) 2.66 GHz CPU computer with 4 GB of RAM. Each agent used a single processor.

Table 1 Tournament: CADIAPLAYER 2007 (MCTS) vs. its Descendants

Game	CADIAPLAYER 2008 win % (MAST)	CADIAPLAYER 2009 win % (RAVE/MAST)	CADIAPLAYER 2010 win % (RAVE/MAST/FAST)
Breakthrough	86.50 (±4.74)	89.00 (±4.35)	77.50 (±5.80)
Checkers	53.25 (±6.45)	84.50 (±4.78)	79.50 (±5.30)
Othello	56.75 (±6.80)	79.75 (±5.52)	84.50 (±4.93)
Skirmish	42.25 (±6.29)	45.00 (±6.55)	96.75 (±2.31)
Connect 5	70.25 (±6.33)	94.00 (±3.23)	93.00 (±3.55)
3D Tic-Tac-Toe	66.50 (±6.56)	92.00 (±3.77)	96.50 (±2.55)
Chinook	82.00 (±5.25)	73.00 (±6.13)	73.25 (±6.13)
9BoardTTT	60.00 (±6.35)	71.50 (±6.27)	70.50 (±6.34)
TTCC4	72.75 (±5.89)	77.25 (±5.24)	51.00 (±6.88)
Average	65.58	78.44	80.28

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 is set to 10 for all agents. The *equivalence parameter* for RAVE is set to 500. In FAST the λ parameter is set to 0.99, the step-size parameter α to 0.01, c to 5, and w to 1. These parameters are the best known settings for each scheme, based on trial and error testing. The startclock and the playclock were both set to 10 seconds.

Our experiments used nine two-player turn-taking games: Breakthrough, Skirmish, Checkers, Othello, Connect 5, 3D Tic-Tac-Toe, Chinook, Nine Board Tic-Tac-Toe, and TTCC4.[1]

[1] *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's pieces as possible. *Breakthrough* is played on a chess board where the players, armed with two ranks of pawns, try to be the first to break through to reach the opponent's back rank. The pawns can move forward and diagonally, but can only capture diagonally. *Chinook* is a variant of Breakthrough played with checkers pawns and the added twist that two independent games are played simultaneously, one on the white squares and another on the black ones. *3D Tic-Tac-Toe* and *Nine Board Tic-Tac-Toe* are variants of Tic-Tac-Toe; the former is played on a $4 \times 4 \times 4$ cube, and in the latter 9 Tic-Tac-Toe boards are placed in a 3×3 grid formation, and on each turn a player can play only on the board having the same coordinate as the last cell marked by the opponent (e.g., if the opponent marked a center cell in one of the boards, the player must on his turn play in the Tic-Tac-Toe board in the center). *TTCC4* is a hybrid of several games where each player has a chess pawn, a chess knight, and a checkers king—these pieces respawn on their start square if captured. Instead of moving a piece a player can choose to drop a disk onto the board like in Connect 4 (captured disks do not respawn). The goal is to form a 3-in-a-row formation with your pieces anywhere on the 3×3 center squares of the table. Full game descriptions can be found on the Dresden GGP server (http://euklid.inf.tu-dresden.de:8180/ggpserver).

4.2 Result

Table 1 shows the result of the matches. The 2007 baseline player uses MCTS but chooses actions uniformly at random in the playout phase. The main improvement in the following year's player, CADIAPLAYER 2008, was MAST for choosing actions in a more informed manner in the playout phase. We can see from column two that the MAST scheme offers an genuine improvement over the MCTS player in almost all the games, particularly in Breakthrough and related games. This is not surprising as the scheme was motivated specifically to improve the agents performance in such type of games. The only game where the scheme has non-positive effects on performance is in Skirmish.

In CADIAPLAYER 2009 the combined RAVE/MAST scheme was introduced, again improving upon the previous year's version in almost all games (except Chinook). The new version is though still inferior to the baseline player in the game of Skirmish. The problem MCTS-based players have with chess-like games like Skirmish is that they do not realize fast enough that the value of the pieces are radically different, e.g. that a rook is worth more than a pawn. Even though MCTS might towards the end of the game start to realize this then it is far to late to save the game.

We specifically targeted this inefficiency in the 2010 version of the agent by incorporating FAST. From the rightmost column we see that the new combined RAVE/MAST/FAST scheme is very effective in Skirmish, getting close to perfect score against the baseline player. Although the new combination does incur reduced performance in a few other games, the overall average winning percentage slightly improves over the 2009 version. More importantly, the performance of the 2010 agent is more robust across a wider range of games, which is important for GGP agents.

We believe that additional performance gains are possible by further tuning the RAVE/MAST/FAST scheme. For example, in our experiments RAVE shows decremental performance effects in the game Chinook, a simultaneous move

game recently added to our test suite. We have verified with additional experiments that decremental behavior is not observed in a turn-taking (but otherwise identical) variant of that game. The relative importance of RAVE in the combined scheme could be controlled by tuning the w parameter based on game properties. The feature-detection templates could also be made more selective. In the above experiments the piece-type feature was activated in Skirmish with a great success, but in TTCC4 with counter effective results. The problem with the latter game was that the number of pieces varied only for disks as the other pieces respawn on capture, and the agent thus learned a significant value for that piece only, making our agent too aggressive in adding such pieces to the board. As for the location based feature, it was activated in Breakthrough, Othello, Connect 5, and Chinook. The only game where this helped was Othello, but playing on specific locations is particularly important in that game, e.g. on corner and edge squares. As for the other games piece formations are more important than placing pieces on particular locations, and the FAST scheme thus adds noise which may sometimes reduce the effectiveness of MAST. This is in particular noticeable in Breakthrough, where the learning preferred advanced piece locations. This results in a playing behavior where the agent advances its pieces prematurely without necessary backup from other pieces. From these experiments, it is clear that a deeper analysis is needed into which game properties must be present to apply the FAST scheme in the most effective way.

5 Related Work

One of the first general game-playing systems was Pell's METAGAMER [20], which played a wide variety of simplified chess-like games. The introduction of the AAAI GGP competition [12] brought about an increased interest in general game-playing systems. CLUNEPLAYER [4] and FLUXPLAYER [23] were the winners of the 2005 and 2006 GGP competitions, respectively. CADIAPLAYER [2] won the competition in 2007 and 2008, and the agent ARY won in 2009 and 2010. The former two agents employ traditional game-tree search (the most recent version of CLUNEPLAYER also had a MC simulation module), whereas the latter two are MCTS based. Another strong MCTS-based agent is MALIGNE.

GGP as a field presents many challenging research topics, which is reflected in the various agents focusing on somewhat different aspects of GGP. Some use traditional game-tree search where the focus is on learning evaluation functions [4, 16, 22], whereas the focus of the simulation-based agents is more on learning search control [8, 9, 14, 25]. Other important research topics include: representations [17], efficient GDL reasoning engines [29], detecting state-space properties such as symmetries [21] and factorability [6, 13], proving game properties [24, 28], and knowledge-transfer learning [1]. There is also interest in extending the expressive power of GDL [19], and a proposal for GDL-II [27] was recently introduced, adding support for both non-deterministic and incomplete information games.

Monte Carlo Tree Search (MCTS) was pioneered in computer Go, and is now used by several of the strongest Go programs, including MOGO [11] and FUEGO [7]. Experiments in Go showing how simulations can benefit from using an informed playout policy are presented in [10]. This, however, requires game-specific knowledge which makes it difficult to apply in GGP. The paper also introduced RAVE. Progressive Strategies [3] are also used by Go programs to improve simulation guidance in the MCTS's selection step.

6 Conclusions and Future Work

Informed search-control for guiding simulations playouts is a core component of CADIAPLAYER. In here we described the search-control scheme in CadiaPlayer 2010 and empirically evaluated the performance gains achieved over the years as it evolved.

Generic search-control schemes using only implicit domain knowledge, such as MAST and RAVE, are able to provide impressive performance across a large collection of games. However, such a generic approach fails to take advantage of higher-level game concepts that are important for skillful play in many game domains. By using search-control methods that incorporate such concepts, like FAST, we were able to improve the performance of CADIAPLAYER in a chess-like game that had previously proved problematic for our agent. Challenges in using such game-type specific approaches include knowing when to apply them and how to balance them with possibly existing more generic search-control approaches. We made initial steps in that direction.

As for future work there is still further research needed of how best to incorporate higher-level game concepts into simulation control, e.g. to understand a much broader range of games and game concepts. We believe that to take GGP simulation-based agents to the next level such reasoning is essential.

References

1. Banerjee B, Stone P (2007) General game learning using knowledge transfer. In: Veloso MM (ed) 20th IJCAI, pp 672–677
2. Björnsson Y, Finnsson H (2009) CADIAPlayer: a simulation-based general game player. IEEE Trans Comput Intell AI in Games 1(1):4–15
3. Chaslot G, Winands M, van den Herik JH, Uiterwijk J, Bouzy B (2007) Progressive strategies for Monte-Carlo tree search. In: 10th JCIS, heuristic search and computer game playing session

4. Clune J (2007) Heuristic evaluation functions for general game playing. In: Holte RC, Howe A (eds) 22nd AAAI. AAAI Press, Menlo Park, pp 1134–1139
5. Coulom R (2006) Efficient selectivity and backup operators in Monte-Carlo tree search. In: van den Herik HJ, Ciancarini P, Donkers HHLM (eds) CG2006. Springer, Berlin, pp 72–83
6. Cox E, Schkufza E, Madsen Ryan MG (2009) Factoring general games using propositional automata. In: GIGA'09 the IJCAI workshop on general game playing
7. Enzenberger M, Müller M (2009) Fuego—an open-source framework for board games and Go engine based on Monte-Carlo tree search. Tech Rep 09-08, Dept of Computing Science, University of Alberta
8. Finnsson H, Björnsson Y (2008) Simulation-based approach to general game playing. In: Fox D, Gomes CP (eds) 23rd AAAI. AAAI Press, Menlo Park, pp 259–264
9. Finnsson H, Björnsson Y (2010) Learning simulation-control in general game-playing agents. In: Fox M, Poole D (eds) 24th AAAI. AAAI Press, Menlo Park, pp 954–959
10. Gelly S, Silver D (2007) Combining online and offline knowledge in UCT. In: Ghahramani Z (ed) 24th ICML, vol 227. ACM, New York, pp 273–280
11. Gelly S, Wang Y, Munos R, Teytaud O (2006) Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA
12. Genesereth MR, Love N, Pell B (2005) General game playing: overview of the AAAI competition. AI Mag 26(2):62–72
13. Günther M, Schiffel S, Thielscher M (2009) Factoring general games. In: GIGA'09 the IJCAI workshop on general game playing
14. Kirci M, Schaeffer J, Sturtevant N (2009) Feature learning using state differences. In: GIGA'09 the IJCAI workshop on general game playing
15. Kocsis L, Szepesvári C (2006) Bandit based Monte-Carlo planning. In: ECML, pp 282–293
16. Kuhlmann G, Dresner K, Stone P (2006) Automatic heuristic construction in a complete general game player. In: 21st AAAI, pp 1457–1462
17. Kuhlmann G, Stone P (2007) Graph-based domain mapping for transfer learning in general games. In: 18th ECML
18. Lorentz RJ (2008) Amazons discover Monte-Carlo. In: van den Herik HJ, Xu X, Ma Z, Winands MHM (eds) CG2008. Lecture notes in computer science, vol 5131. Springer, Berlin, pp 13–24
19. Love N, Hinrichs T, Genesereth M (2006) General game playing: game description language specification. Technical Report, Stanford University, 4 April 2006
20. Pell B (1996) A strategic metagame player for general chess-like games. Comput Intell 12:177–198
21. Schiffel S (2010) Symmetry detection in general game playing. In: Fox M, Poole D (eds) 24th AAAI, pp 980–985. AAAI Press, Menlo Park
22. Schiffel S, Thielscher M (2007) Automatic construction of a heuristic search function for general game playing. In: Veloso MM (ed) 7th IJCAI workshop on nonmontonic reasoning, action and change (NRAC07)
23. Schiffel S, Thielscher M (2007) Fluxplayer: a successful general game player. In: Holte RC, Howe A (eds) 22nd AAAI, AAAI Press, Menlo Park, pp 1191–1196
24. Schiffel S, Thielscher M (2009) Automated theorem proving for general game playing. In: Boutilier C (ed) 21st IJCAI. Morgan Kaufmann, San Francisco, pp 911–916
25. Sharma S, Kobti Z, Goodwin S (2008) Knowledge generation for improving simulations in UCT for general game playing. In: AI 2008: advances in artificial intelligence. Springer, Berlin, pp 49–55
26. Sutton RS (1988) Learning to predict by the methods of temporal differences. Mach Learn 3:9–44
27. Thielscher M (2010) A general game description language for incomplete information games. In: Fox M, Poole D (eds) 24th AAAI. AAAI Press, Menlo Park, pp 994–999
28. Thielscher M, Voigt S (2010) A temporal proof system for general game playing. In: Fox M, Poole D (eds) 24th AAAI. AAAI Press, Menlo Park, pp 1000–1005
29. Waugh K (2009) Faster state manipulation in general games using generated code. In: GIGA'09 the IJCAI workshop on general game playing

Hilmar Finnsson is a Ph.D. student at the School of Computer Science at Reykjavik University working on General Game Playing. He is a co-author of the GGP agent CADIAPLAYER.

Yngvi Björnsson is an associate professor at the School of Computer Science at Reykjavik University and the director of Reykjavik University's Center for Analysis and Design of Intelligent Agents (CADIA). He is a co-author of the GGP agent CADIAPLAYER.