Simulation-Based Approach to General Game Playing

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

Abstract

The aim of *General Game Playing (GGP)* is to create intelligent agents that automatically learn how to play many different games at an expert level without any human intervention. The most successful GGP agents in the past have used traditional game-tree search combined with an automatically learned heuristic function for evaluating game states. In this paper we describe a GGP agent that instead uses a Monte Carlo/UCT simulation technique for action selection, an approach recently popularized in computer Go. Our GGP agent has proven its effectiveness by winning last year's AAAI GGP Competition. Furthermore, we introduce and empirically evaluate a new scheme for automatically learning search-control knowledge for guiding the simulation playouts, showing that it offers significant benefits for a variety of games.

Introduction

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, provided 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, and machine learning.

The most successful GGP agents so far have been based on the traditional approach of using game-tree search augmented with an (automatically learned) heuristic evaluation function for encapsulating the domain-specific knowledge (Clune 2007; Schiffel & Thielscher 2007; Kuhlmann, Dresner, & Stone 2006). However, instead of using a set of carefully hand-crafted domain-specific features in their evaluation as high-performance game-playing programs do, GGP programs typically rely on a small set of generic features (e.g. piece-values and mobility) that apply in a wide range of games. The relative importance of the features is then automatically tuned in real-time for the game at hand.

There is an inherent risk with that approach though. In practice, because of how disparate the games and their playing strategies can be, the pre-chosen set of generic features may fail to capture some essential game properties. On top of that, the relative importance of the features can often be only roughly approximated because of strict online time constraints. Consequently, the resulting heuristic evaluations may become highly inaccurate and, in the worst case, even strive for the wrong objectives. Such heuristics are of a little (or even decremental) value for lookahead search.

In here we describe a simulation-based approach to general game playing that does not require any *a priori* domain knowledge. It is based on *Monte-Carlo (MC)* simulations and the *Upper Confidence-bounds applied to Trees (UCT)* algorithm for guiding the simulation playouts (Kocsis & Szepesvári 2006), thus bypassing the need for a heuristic evaluation function. Our GGP agent, CADIAPLAYER, uses such an approach to reason about its actions and has already proven its effectiveness by winning last year's annual GGP competition. The UCT algorithm has recently been used successfully in computer Go programs, dramatically increasing their playing strength (Gelly *et al.* 2006; Coulom 2006). However, there are additional challenges in applying it to GGP, for example, in Go pre-defined domain-knowledge can be used to guide the playout phase, whereas such knowledge must be automatically discovered in GGP.

The main contributions of the paper are as follows, we: (1) describe the design of a state-of-the-art GGP agent and establish the usefulness of simulation-based search approaches in GGP, in particular when used in combination with UCT; (2) empirically evaluate different simulation-based approaches on a wide variety of games, and finally (3) introduce a domain-independent enhancement for automatically learning search-control domain-knowledge for guiding simulation playouts. This enhancement provides significant benefits on all games we tried, the best case resulting in 90% winning ratio against a standard UCT player.

The paper is structured as follows. In the next section we give a brief overview of GGP, followed by a review of our GGP agent. Thereafter we detail the simulation-based search approach used by the agent and highlight GGP specific enhancements, including the new technique for improving the playout phase in a domain-independent manner. Finally, we present empirical results and conclude.

General Game Playing

Artificial Intelligence (AI) researchers have for decades worked on building game-playing systems capable of matching wits with the strongest humans in the world. The success of such systems has largely been because of improved search algorithms and years of relentless knowledge-engineering effort on behalf of the program developers, manually adding game-specific knowledge to their programs. The long-term aim of GGP is to take that approach to the next level with intelligent agents that can automatically learn to play skillfully without such human intervention.

The GGP Competition (Genesereth, Love, & Pell 2005) was founded as an initiative to facilitate further research into this area. Game-playing agents connect to a game server that conducts the matches. Each match uses two separate time controls: a *start-clock* and a *play-clock*. The former is the time the agent gets to analyze the game description until play starts, and the latter is the time the agent has for deliberating over each move decision. The server oversees play, relays moves, keeps time, and scores the game outcome.

Game descriptions are specified in a Game Description Language (GDL) (Love, Hinrichs, & Genesereth 2006), a specialization of KIF (Genesereth & Fikes 1992), a first-order logic based language for describing and communicating knowledge. It is a variant of Datalog that allows function constants, negation, and recursion (in a restricted form). The expressiveness of GDL allows a large range of deterministic, perfect-information, simultaneous-move games to be described, with any number of adversary or cooperating players. Turn-based games are modeled by having the players who do not have a turn return a *no-operation* move. A GDL game description specifies the initial game state, as well as rules for detecting and scoring terminal states and for generating and playing legal moves. A game state is defined by the set of propositions that are true in that state.

CadiaPlayer

An agent competing in the GGP competition requires at least three components: a HTTP server to interact with the GGP game server, the ability to reason using GDL, and the AI for strategically playing the games presented to it. In CADIAPLAYER the HTTP server is an external process, whereas the other two components are integrated into one game-playing engine. The HTTP server process is always kept running, monitoring the incoming port. It spawns an instance of the game-playing engine for each new game description it receives from the GGP game server.

The game engine translates the GDL description into Prolog code using an external home-made tool. The generated code is then — along with some pre-written Prolog code — compiled into a library responsible for all game-specific state-space manipulations, that is, the generation and executing of legal moves and the detection and scoring of terminal states. We use YAP Prolog (Costa *et al.* 2006) for this purpose, mainly because it is reasonably efficient and provides a convenient interface for accessing the compiled library routines from another host programming language. The game engine itself is written in C++, and its most important parts are the search algorithms used for making action decisions.

For single-agent games the engine uses an improved variation of the *Memory Enhanced IDA** (Reinefeld & Marsland 1994) search algorithm. The search starts immediately during the start-clock. If successful in finding at least a partial solution (i.e. a goal with a higher than 0 point reward) it continues to use the algorithm on the play-clock, looking for improved solutions. However, if unsuccessful, the engine falls back on using the UCT algorithm on the play-clock. The single-agent search module is still somewhat rudimentary in our agent (e.g. the lack of a heuristic function), although it ensures that the agent can play at least the somewhat less complex puzzles (and sometimes even optimally).

Apart from the single-agent case, all action decisions are made using UCT/MC simulation searches. The simulation approach applies to both two- and multi-player games, whether they are adversary or cooperative. In two-player games the agent can be set up to either maximize the difference in the players' score (games are not necessarily zero-sum), or maximize its own score, but with tie-breaking towards minimizing the opponent's score. In multi-player games the agent considers only its own score, ignoring the ones of the other players. This sacrifices the possibility of using elaborate opponent-modeling strategies, but is done for simplification purposes.

A detailed discussion of the architecture and implementation of CADIAPLAYER is found in (Finnsson 2007).

Search

The UCT algorithm (Kocsis & Szepesvári 2006) is the core component of our agent's action-selection scheme. It is a variation of the Upper Confidence Bounds algorithm (UCB1) (Auer, Cesa-Bianchi, & Fischer 2002) applied to trees, and offers a simple — yet sound and effective — way to balance exploration and exploitation.

The UCT algorithm gradually builds a game tree in memory where it keeps track of the average return of each state-action pair played, $\mathcal{Q}(s,a)$. During a simulation, when still within the tree, it selects the action to explore by:

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

The $\mathcal{Q}$ function is the action value function as in an MC algorithm, but the novelty of UCT is the second term — the so-called UCT bonus. The N function returns the number of visits to a state or the number of times a certain action has been sampled in a certain state, depending on the parameters. If there exists an action in $A(s)$, the set of possible actions in state s, that has never been sampled and has therefore no estimated value, the algorithm defaults to selecting it before any previously sampled action. The UCT term builds a level of confidence into the action selection, providing a balance between exploiting the perceived best action and exploring the suboptimal ones. When an action is selected its UCT bonus decreases (because $N(s,a)$ is incremented), whereas the bonus for all the other actions increases slightly (because $N(s)$ is incremented). The C parameter is used to tune how aggressively to consider the UCT bonus.

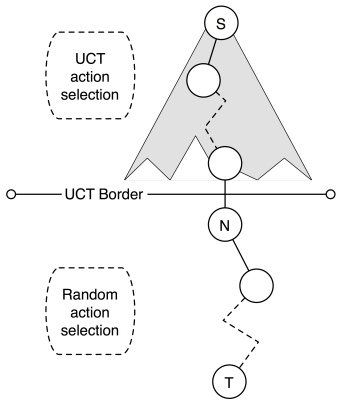

Figure 1: Conceptual overview of a single UCT simulation

The game tree (UCT tree) that is built in memory stores the necessary statistics. However, its size must be managed to counteract running out of memory. The parts of the tree that are above the current state are deleted each time a non-simulated action is played. Also, for every simulated episode, only the first new node encountered is stored (Coulom 2006). An overview of a single UCT simulation is given in Figure 1. The start state is denoted by S, the terminal state with T, and N is the new state added to the model after the simulation finishes. As GDL rules require a move from all players for each state transition, the edges in the figure represent a set of moves. When the UCT border has been passed the default tie-breaking scheme results in random play to the end of the episode. Because better actions are selected more often than suboptimal ones, the tree grows asymmetrically. Consistently good lines of play are grown aggressively, sometimes even to the end of the game, whereas uninteresting branches are rarely explored and will remain shallow.

Opponent Modeling

To get the best performance out of the UCT algorithm we must model not only the role CADIAPLAYER plays, but also the ones of the other players. So for each opponent in the game a separate game-tree model is set up estimating the returns it receives. Because GGP is not limited to two-player zero-sum games, the opponents cannot be modeled simply by using the negation of our return value. Any participant can have its own agenda and therefore needs its own action-value function. All these game-tree models work together when running simulations and control the UCT action selection for the player they are modeling.

Algorithm 1 shows how the opponent modeling is combined with UCT/MC in CADIAPLAYER. The discount factor γ is set to 0.99 and makes the algorithm prefer earlier rather than later payoffs. The *StateSpaces* array stores the different models. The functions *selectMove* and *update* use the corresponding model to make move selections and updates (based on the UCT rule). The *update* function builds the game-tree model and is responsible for adding only one node per simulation. When the time comes to select the best action CADIAPLAYER's model is queried for the action with the highest $\mathcal{Q}(s, a)$ value.

Algorithm 1 search(ref qValues[])

```
1: if isTerminal() then
2:    for all r_i in getRoles() do
3:       qValues[i] ← goal(i)
4:    end for
5:    return
6: end if
7: playMoves ← ∅
8: for all r_i in getRoles() do
9:    moves ← getMoves(r_i)
10:   move ← selectMove(moves, StateSpaces[i])
11:   playMoves.insert(move)
12:   moves.clear()
13: end for
14: make(playMoves)
15: search(qValues)
16: retract()
17: for r_i in getRoles() do
18:   qValues[i] ← γ * qValues[i]
19:   update(playMoves[i], qValues[i], StateSpaces[i])
20: end for
21: return
```

Choosing among Unexplored Actions

During the GGP competition we noticed that our agent sometimes played too "optimistically", even relying on the opponent making a mistake. This was particularly visible in the game *Breakthrough*. In Figure 2 we see an example position from this game (a slight variation of it with four squares blocked was used in the semi-finals of the GGP competition). The game is played on an 8 by 8 chess or checkers board. The pieces are set up in the two back ranks, Black at the top and White at the bottom. White goes first and the players then alternate moving. The pieces move one step forward into an empty square either straight or diagonally, although captures are done only diagonally (i.e. as in chess). The goal of the game is to be the first player to reach the opponent's back rank.

It is Black's turn to move in Figure 2. Our agent would initially find it most attractive to move the far advanced black piece one square forward (b4-b3). However, this is obviously a bad move because White can capture the piece with a2-b3; this is actually the only good reply for White as all the others lead to a forced win for Black (b3-a2 followed by a2-b1). Simulations that chose White's reply at random (or highly exploratory) have problems with a move like this one because most of the simulation playouts give a positive return. The UCT algorithm would gradually start to realize this, although a number of simulations may be required. However, if this move were played in a MC simulation playout it would continue to score well (there is no memory) and erroneous information would propagate back into the UCT tree. Special pre-programmed move patterns are used in computer Go to detect many of such only-reply moves. In GGP programs, however, this must be learned automatically.

When encountering a state with unexplored actions CADIAPLAYER used to select the next unexplored action to ex-

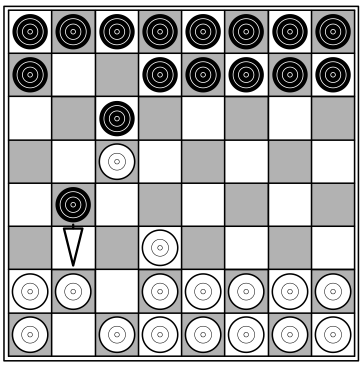

Figure 2: Breakthrough game position

plore uniformly at random, because it had no criteria for knowing which ones are more likely to be good. One way to add such criteria in a domain independent way is to exploit the fact that actions that are good in one state are often also good in other states. For example, in our example above White capturing on b3 will likely continue to be the best action even though the remaining pieces would be positioned slightly differently. The *history-heurstic* (Schaeffer 1989), which is a well-established move-ordering mechanism in chess, is based on this same principle. In an improved version of our agent, in addition to the action-values $\mathcal{Q}(s,a)$, the agent also keeps for each action encountered its average return independent of the state where it was played, i.e. $\mathcal{Q}_h(a)$. This value is used to bias which unexplored action to investigate next, both in the MC playout phase and when encountering nodes in the UCT tree having unexplored actions. This is done using Gibbs sampling as below:

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

where $\mathcal{P}(a)$ is the probability that action a will be chosen in that state — actions with a high $\mathcal{Q}_h(a)$ value are more likely. The $\mathcal{Q}_h(a)$ value of an action that has not been explored yet is set to the maximum GGP score (100) to bias towards similar exploration as is default in the UCT algorithm. 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).

Empirical Evaluation

The UCT simulation-based approach has already proved its effectiveness against traditional game-tree search players. Our agent won the annual 2007 GGP competition. In the preliminary rounds of the competition, played over a period of 4 weeks, the agent played a large number of match games using over 40 different types of games. It won the preliminaries quite convincingly. For example, in the second half of the tournament where somewhat more complex games were used (and the technical difficulties had been ironed out) it scored over 85% of the available points, whereas the closest competitors all scored well under 70%. At the GGP competition finals held at the AAAI conference a knockout format with only a handful of games was used. Such a format — although providing some excitement for the audience — does not do justice towards finding the best overall player, and although our agent did win there was some luck involved. The main difference between the version of CADIAPLAYER used in the preliminaries and the one in the finals was that the former ran on a single CPU whereas in the latter the simulation searches had been parallelized and ran on 8-12 CPUs.

In the remainder of this section we evaluate the effectiveness of different simulation-based search approaches. The objective of the experiments is threefold: to demonstrate the benefits of UCT over standard MC in the context of GGP, evaluate the effectiveness of our improved action-selection scheme, and investigate how increasing the number of simulations affects the quality of play.

Experimental Setup

We matched different variants of the agent against each other. They were all built on the same software framework to minimize the impact of implementation details, and differed only in the simulation approach being evaluated. We refer to the GGP competition version of CADIAPLAYER as CP$_{uct}$ where the UCT parameter C is set to 40 (for perspective, possible game outcomes are in the range 0-100). In the result tables that follow each data point represents the result of a 250-game match between two players alternating roles; both the winning percentage and a 95% confidence interval are provided. 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. For each game, both the start- and the play-clocks were set to 30 seconds. Four different two-player games were used in the experiments: *Connect-4*, *Checkers*, *Othello*, and *Breakthrough* (the same variant as was used in the GGP competition semi-finals). The GDL descriptions of the games can be downloaded from the official GGP game server.

UCT vs. MC

In here we contrast the performance of our UCT player against two different MC players. The benefits of UCT over standard MC are twofold: a more informed action-selection rule and caching of already expanded game-tree nodes and actions. We investigate the contributions of these two factors independently, thus the two baseline MC players. The former, MC$_{org}$, uses a uniform random distribution for action selection for the entire playout phase, and then chooses the action at the root with the highest average return. The latter, MC$_{mem}$, uses identical action-selection mechanism to the first (i.e. highest average return) but is allowed to build a top-level game tree one node per simulation, as CP$_{uct}$ does.

The match results are shown in Table 1. The UCT player outperforms both baseline players by a large margin in all four games, with an impressive average winning percentage ranging from 77% to over 91%. It is also of interest to note that the added memory is helpful, although the main benefit still comes from the UCT action-selection rule. However, the usefulness of retaining the game tree in memory differs between games and is most beneficial in Checkers. This is likely because of its low branching factor (because of the forced-capture rule), resulting in large parts of the game tree being kept between moves. Another benefit of

Table 1: Results of UCT and MC matches in %

Game	Player	MC_{org}	MC_{mem}	CP_{uct}	Total
Connect-4	MC_{org}	*N/A*	41.8 (± 5.93)	8.8 (± 3.38)	25.3 (± 3.71)
	MC_{mem}	58.2 (± 5.93)	*N/A*	8.4 (± 3.26)	33.3 (± 4.03)
	CP_{uct}	91.2 (± 3.38)	91.6 (± 3.26)	*N/A*	91.4 (± 2.35)
Checkers	MC_{org}	*N/A*	15.4 (± 4.18)	8.2 (± 2.95)	11.8 (± 2.57)
	MC_{mem}	84.6 (± 4.18)	*N/A*	28.6 (± 5.08)	56.6 (± 4.10)
	CP_{uct}	91.8 (± 2.95)	71.4 (± 5.08)	*N/A*	81.6 (± 3.07)
Othello	MC_{org}	*N/A*	35.0 (± 5.86)	16.0 (± 4.49)	25.5 (± 3.78)
	MC_{mem}	65.0 (± 5.86)	*N/A*	26.2 (± 5.39)	45.6 (± 4.33)
	CP_{uct}	84.0 (± 4.49)	73.8 (± 5.39)	*N/A*	78.9 (± 3.53)
Breakthrough	MC_{org}	*N/A*	47.6 (± 6.20)	20.0 (± 4.97)	33.8 (± 4.15)
	MC_{mem}	52.4 (± 6.20)	*N/A*	25.2 (± 5.39)	38.8 (± 4.28)
	CP_{uct}	80.0 (± 4.97)	74.8 (± 5.39)	*N/A*	77.4 (± 3.67)

Table 2: Tournament between CP_{uct} and CP_{imp}

Game	CP_{imp} win %
Connect-4	54.2 (± 6.08)
Checkers	54.4 (± 5.77)
Othello	65.0 (± 5.83)
Breakthr.	90.0 (± 3.73)

Table 3: Time-control comparison for CP_{uct}

Game	10 / 5 sec	20 / 10 sec	40 / 20 sec
Connect-4	64.2 (± 5.81)	63.2 (± 5.83)	65.4 (± 5.79)
Checkers	76.2 (± 4.85)	72.2 (± 4.96)	77.8 (± 4.33)
Othello	67.0 (± 5.75)	64.0 (± 5.86)	69.0 (± 5.68)
Breakthr.	66.8 (± 5.85)	67.6 (± 5.81)	64.8 (± 5.93)

having a game tree in memory is that we can cache legal moves. This speeds up the simulations when still in the tree, because move generation — a relatively expensive operation in our GGP agent — is done only once for the corresponding states. We measured the effect of this, and CP_{uct} and MC_{mem} manage on average around 35% (Othello) to approximately twice (Checkers, Connect-4) as many simulations as MC_{org} does. The added number of simulations explains some of the performance increase.

Unexplored Action Selection Enhancement

The following experiment evaluates the unexplored action-selection enhancement. The τ parameter of the Gibbs distribution was set to 10 (based on trial-and-error on a small number of games). The result is shown in Table 2.

The new action-selection scheme offers some benefits for all the games, although for Connect-4 and Checkers we can say with only 91% and 93% confidence, respectively, that the new enhanced player is the better one (using a one-tailed test). Most noticeable is though how well this improvement works for the game Breakthrough, maybe not surprising given that it was UCT's behavior in that game that motivated the scheme. It also offers significant improvements in the game of Othello, but in that game a move that is good in one position — e.g. place a piece in a corner or on an edge — is most likely also good in a different position. This seems to be the deciding factor.

Time-Control Comparison

To find out how increased number of simulations affects UCT's performance, we ran experiments with two identical players where one player was given twice the thinking time of the other. The player with more time won all matches convincingly as seen in Table 3. Moreover, there are no signs of diminishing performance improvement as the time controls are raised. This is positive and indicates that simulation-based approaches will probably continue to gain momentum with more massive multi-core CPU technology. It is worth noting that a simulation-based search is much easier to parallelize than traditional game-tree search algorithms.

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. The introduction of the GGP competition renewed the interest in the field, and several research groups world-wide are now actively participating.

The winners of the 2005 and 2006 GGP competition were CLUNEPLAYER (Clune 2007) and FLUXPLAYER (Schiffel & Thielscher 2007), respectively. They both use automatic feature discovery to build a heuristic evaluation function for a traditional game-tree search. Another strong GGP agent using the traditional method is UTEXAS LARG (Kuhlmann, Dresner, & Stone 2006) and its authors, among others, research knowledge transfers between GGP games (Banerjee, Kuhlmann, & Stone 2006; Banerjee & Stone 2007). Two agents that participated in the 2007 GGP competition, ARY and JIGSAWBOT, used MC simulations although not UCT, according to their authors (personal comm., July, 2007).

The UCT algorithm has been used successfully to advance the state-of-the-art in computer Go, and is now used by several of the strongest Go programs, e.g. *MoGo* (Gelly *et al.* 2006) and *Crazy Stone* (Coulom 2006). Experiments in Go showing how UCT can benefit from using an informed playout policy are presented in (Gelly & Silver 2007). The method, however, requires game-specific knowledge which makes it difficult to apply to GGP. In the paper the authors also show how to speed up the initial stages of the learning process in UCT by using a so-called *Rapid Action Value Estimation (RAVE)*, which is closely related to the history heuristic.

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.

Conclusions

We have established UCT simulations as a promising alternative to traditional game-tree search in GGP. The main advantages that UCT has over standard MC approaches are a more informed action-selection rule and game-tree memory, although MC can too be enriched with memory. The main benefit comes from UCT's action-selection rule though.

Simulation methods work particularly well in "converging" games (e.g. Othello, Amazons, and Breakthrough), where each move advances the game closer towards the end, as this bounds the length of each simulation playout. However, simulation-based methods may run into problems in games that converge slowly — we have observed this in some chess-like games (e.g. *Skirmish* played in the GPP competition). Both players can keep on for a long time without making much progress, resulting in many simulation runs becoming excessively long. To artificially terminate a run prematurely is of a limited use without having an evaluation function for assessing non-terminal states; such a function may be necessary for playing these games well.

In general, however, there are many promising aspects that simulations offer over traditional game-tree search in GGP. The main advantage is that simulations do implicitly capture in real-time game properties that would be difficult to explicitly learn and express in a heuristic evaluation function. Also, simulation searches are easily parallelizable and do not show diminishing returns in performance improvement as thinking time is increased, thus promising to take full advantage of future massively multi-core CPUs.

Search-control heuristics are important for guiding the simulation playouts. We introduced one promising domain-independent search-control method, that increased our agent's playing strength on all games we tried it on, in the best case defeating a standard UCT player with 90% winning score. It is worthwhile pointing out that it is not as critical for UCT search to learn accurate search-control heuristics, as it is for traditional game-tree search to have a good evaluation function. In both cases performance will degrade when using bad heuristics, but the UCT approach will recover after some number of playouts, whereas the game-tree search will chase the wrong objective the entire game.

As for future work there are many interesting research avenues to explore to further improve the UCT approach in GGP. There are still many parameter values that can be tuned (e.g. C and τ), preferably automatically for each game at hand. The simulation playouts can be improved further, and we have already started exploring additional schemes for automatically learning search control. Also, an interesting line of work would be to try to combine the best of both worlds — simulations and traditional game-tree search — for example, for evaluating final non-terminal states when simulations are terminated prematurely.

References

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine Learning* 47(2/3):235–256.

Banerjee, B., and Stone, P. 2007. General game learning using knowledge transfer. In *The 20th International Joint Conference on Artificial Intelligence*, 672–677.

Banerjee, B.; Kuhlmann, G.; and Stone, P. 2006. Value function transfer for General Game Playing. In *ICML Workshop on Structural Knowledge Transfer for ML*.

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

Costa, V. S.; Damas, L.; Reis, R.; and Azevedo, R. 2006. YAP Prolog user's manual. Retrieved January 27, 2008, from http://www.ncc.up.pt/˜vsc/Yap/documentation.html.

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

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.

Genesereth, M. R., and Fikes, R. E. 1992. Knowledge interchange format, version 3.0 reference manual. Technical Report Technical Report Logic-92-1, Stanford University.

Genesereth, M. R.; Love, N.; and Pell, B. 2005. General Game Playing: Overview of the AAAI competition. *AI Magazine* 26(2):62–72.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In *European Conference on Machine Learning (ECML)*, 282–293.

Kuhlmann, G.; Dresner, K.; and Stone, P. 2006. Automatic heuristic construction in a complete general game player. In *Proc. of the Twenty-First National Conference on Artificial Intelligence*, 1457–62.

Love, N.; Hinrichs, T.; and Genesereth, M. 2006. General Game Playing: Game description language specification. Technical Report April 4 2006, Stanford University.

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

Reinefeld, A., and Marsland, T. A. 1994. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16(7):701–710.

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

Schiffel, S., and Thielscher, M. 2007. Fluxplayer: A successful general game player. In *Proc. of the Twenty-Second AAAI Conference on Artificial Intelligence*, 1191–1196.