

CADIAPLAYER: A Simulation-Based General Game Player

Yngvi Björnsson and Hilmar Finnsson

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. The traditional design model for GGP agents has been to use a minimax-based game-tree search augmented with an automatically learned heuristic evaluation function. The first successful GGP agents all followed that approach. In this paper, we describe CADIAPLAYER, a GGP agent employing a radically different approach: instead of a traditional game-tree search, it uses Monte Carlo simulations for its move decisions. Furthermore, we empirically evaluate different simulation-based approaches on a wide variety of games, introduce a domain-independent enhancement for automatically learning search-control knowledge to guide the simulation playouts, and show how to adapt the simulation searches to be more effective in single-agent games. CADIAPLAYER has already proven its effectiveness by winning the 2007 and 2008 Association for the Advancement of Artificial Intelligence (AAAI) GGP competitions.

Index Terms—Artificial intelligence (AI), games, Monte Carlo methods, search methods.

I. INTRODUCTION

FROM the inception of the field of AI, over half a century ago, games have played an important role as a test-bed for advancements in the field. AI researchers have worked over the decades on building high-performance game-playing systems for games of various complexity capable of matching wits with the strongest humans in the world [1]–[3]. The importance of having such an objective measure of the progress of intelligent systems cannot be overestimated, nonetheless, this approach has led to some adverse developments. For example, the focus of the research has been driven to some extent by the quest for techniques that lead to immediate improvements to the game-playing system at hand, with less attention paid to more general concepts of human-like intelligence such as acquisition, transfer, and use of knowledge. Thus, the success of game-playing systems has been in part because of years of relentless knowledge-engineering effort on behalf of the program developers, manually adding game-specific knowledge to their programs. The aim of general game playing is to take that approach to the next level.

Manuscript received December 19, 2008; revised January 30, 2009; accepted February 07, 2009. First published March 24, 2009; current version published May 01, 2009. This work was supported in part 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.

The authors are with the School of Computer Science, Reykjavík University, IS-103 Kringlan 1, Reykjavík, Iceland (e-mail: yngvi@ru.is; hif@ru.is).

Digital Object Identifier 10.1109/TCIAIG.2009.2018702

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 AI subdisciplines such as knowledge representation, agent-based reasoning, heuristic search, computational intelligence, and machine learning.

The two core components of any game-playing program are *search* and *evaluation*. The former provides the ability to think ahead, whereas the latter provides a mechanism for assessing the merits of the game positions that arise during the search. Domain-dependent knowledge plays an important part in both components, in particular, for game-position evaluation, understandably, but also for providing effective search guidance. The main challenge faced by general game-playing systems, as opposed to game-playing systems for playing one specific game, is that the relevant game-specific knowledge required for expert-level play, whether for the search or the evaluation, must be effectively discovered during play.

The first successful GGP agents were all based on the traditional approach of using a minimax-based game-tree search augmented with an automatically learned heuristic evaluation function for encapsulating the domain-specific knowledge [4]–[6]. 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 to a wide range of games. The applicability and relative importance of the features are then automatically decided in real time for the game at hand. More recently, GGP agents using Monte Carlo simulations for reasoning about their actions have shown much promise [7].

In the last couple of years, Monte Carlo search as a decision mechanism for game-playing programs has been shown to be surprisingly effective in computer Go, dramatically increasing the playing strength of such programs [8], [9]. Such an approach also offers several attractive properties for general game-playing agents. For one, it relies on heuristic evaluation functions to a much lesser extent, even bypassing the need for them altogether. This can be a real asset in GGP as the different games the agents are faced with may require disparate playing strategies, some of which may have never been seen before. In such scenarios, automatically learned heuristic evaluation functions may fail to capture essential game properties, resulting in the evaluations becoming highly inaccurate, in the worst case, even causing the agent to strive for the wrong objectives. In compar-

ison to computer Go, there are additional challenges in applying Monte Carlo search to GGP. In particular, whereas in computer Go predefined domain knowledge is extensively used to guide the simulation payout phase, such knowledge must be automatically discovered in GGP.

In this paper, we describe CADIAPLAYER, our GGP agent. It does not require any *a priori* domain knowledge nor does it use heuristic evaluation of game positions. Instead, it relies exclusively on Monte-Carlo-based simulation search for reasoning about its actions, but it is guided by an effective search-control learning mechanism. The agent has already proven itself by winning the last two international GGP competitions, thus being the reigning GGP world champion.

The main contributions of this paper are as follows: 1) we describe the design and architecture of a state-of-the-art GGP agent, which established the usefulness of simulation-based search approaches in GGP; 2) we empirically evaluate different simulation-based approaches on a wide variety of games; 3) we introduce a domain-independent enhancement for automatically learning search-control domain knowledge for guiding simulation payouts, and finally; 4) we show how to adapt the simulation searches to be more effective in single-agent games.

This paper is structured as follows. In the next section, we give a brief overview of general game playing; thereafter, we discuss the architecture of our agent, followed by a detailed discussion of its core component: the simulation-based search. We highlight the GGP specific enhancements, including the search-control learning technique used for improving the payout phase in a domain-independent manner. Then, we present extensive empirical results, before discussing related work and finally concluding.

II. GENERAL GAME PLAYING

The Logic Group at Stanford University, Stanford, CA, initiated the General Game Playing Project a few years back to facilitate further research into the area, along with the annual GGP competition. For this purpose, they provide both a well-defined language for describing games and a web-based server for running and viewing general game playing matches.

A. Game Description Language

Games are specified in a game description language (GDL) [10], a specialization of knowledge interchange format (KIF) language [11], 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 special no operation move. A GDL game description uses keywords known as relations to specify 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. Only the relations have lexical meaning and during competitions everything else is obfuscated. Following is a brief overview of GDL, using the partial *Tic-Tac-Toe* description in

```
(role xplayer)
(role oplayer)

(init (cell 1 1 b))
(init (cell 1 2 b))
...
(init (control xplayer))

(<= (legal ?w (mark ?x ?y))
   (true (cell ?x ?y b))
   (true (control ?w)))
(<= (legal oplayer noop)
   (true (control xplayer)))
...
(<= (next (cell ?m ?n x))
   (does xplayer (mark ?m ?n))
   (true (cell ?m ?n b)))
(<= (next (control oplayer))
   (true (control xplayer)))
...
(<= (row ?m ?x)
   (true (cell ?m 1 ?x))
   (true (cell ?m 2 ?x))
   (true (cell ?m 3 ?x)))
...
(<= (line ?x)
   (row ?m ?x))
(<= (line ?x)
   (column ?m ?x))
(<= (line ?x)
   (diagonal ?x))
...
(<= (goal xplayer 100)
   (line x))
(<= (goal xplayer 0)
   (line o))
...
(<= terminal
   (line x))
```

Fig. 1. A partial Tic-Tac-Toe GDL description.

Fig. 1 as a reference. A complete GDL description for *Tic-Tac-Toe* is provided in the Appendix.

The *role* relation lists the players participating in the game; arbitrarily many roles can be declared, that is, a game can be single player (i.e., a puzzle), two player, or multiplayer. However, once declared, the roles are fixed throughout the game. In our *Tic-Tac-Toe* example, two players are defined, *xplayer* and *oplayer*. The *init* relation states the facts that are true in the initial state, and they are added to the knowledge base. Here a game state is represented by the board position (initially all cells are empty), and whose turn it is to move. The *true* relation is used in GDL to check if a fact is in the knowledge base.

The *legal* and *next* relations are used to determine legal moves and execute them, respectively. In Fig. 1, the player having the turn can mark any empty cell on the board (variables in GDL are prefixed with a question mark), whereas the opponent can only play a *no-op* move (recall that GDL assumes simultaneous moves from all players). Moves are executed by temporarily adding a *does* fact to the knowledge base stating the move played, and then the *next* clause is called to determine how the resulting state looks like. Fig. 1 shows the *next* relations for updating a cell and the turn to move for one of the players; symmetrical *next* relations are needed for the other player, as well as relations for stating that the untouched cells retain their

mark (see the Appendix). The facts that the *next* call returns are added to the knowledge base, replacing all previous state information stored there (the temporary *does* fact is also removed). This knowledge representation formalism corresponds to the closed-world assumption; that is, the only true facts are those that are known to be so; others are assumed false.

The *terminal* relation tells if a game position is reached where the game is over, and the *goal* relation returns the value of the current game state; for terminal states, this corresponds to the game outcome. The goal scores are in the range [0–100], and if a game is coded to provide intermediate goal values for nonterminal positions, then GDL imposes the restriction that the values are monotonically nondecreasing as the game progresses. Examples of such a scoring system are games where players get points based on the number of pieces captured. In Fig. 1, user-defined predicates are also listed, for example, *line*, *row*, *column*, and *diagonal*; arbitrarily many such predicates are allowed.

The official syntax and semantics of GDL are described in [10].

B. GGP Communication Protocol

The game master (GM) is a server for administrating matches between GGP agents. It does so via the hypertext transfer protocol (HTTP). Before a game starts, the GGP agents register with the server. Each new match game gets a unique identification string. Play begins with a start message being sent to all the agents, containing the match identifier, the GDL game description, the role of the agent, and the time limits used. After all players have responded, play commences by the GM requesting a move from all players; subsequent move requests sent by the GM contain the previous round moves of all players. This way each player can update its game state in accordance with what moves the other players made. This continues until the game reaches a terminal state. If a player sends an illegal move to the GM, a random legal move is selected for that player.

The time limits mentioned above are positive integer values called *startclock* and *playclock*. The value for them is presented in seconds and the *startclock* indicates the time from receiving the rules until the game begins, and the *playclock* indicates the time the player has for deliberating each move.

More details, including the format of the HTTP message contents, can be found in [10]. A full description of the GM capabilities is given in [12].

III. ARCHITECTURE

An agent competing in the GGP competition requires at least three components: an HTTP server to interact with the GM, 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. An overview of its architecture is shown in Fig. 2.

The topmost layer of the figure is the HTTP server which runs the rest of the system as a child process and communicates with it via standard pipes. Every time a new game begins a new process is spawned and the old one is suspended.

The game-playing engine is written in C++ and can be split up into three conceptual layers: the *game-agent interface*, the *game-play interface*, and the *game-logic interface*. The game-agent interface handles external communications and manages the flow of the game. It queries the game-play interface for all intelligent behavior regarding the game. In the game-logic interface, the state space of the game is queried and manipulated.

A. Game-Agent Interface

This layer manages the game flow by interacting with and executing command requests from the GM. It also includes a game parser for building a compact internal representation for referencing atoms and producing KIF strings, both needed by the game-play interface. The parser also converts moves sent from the GM into the internal form. Upon receiving a new game start message, the agent saves the GDL description in the message to a file.

B. Game-Play Interface

This is the main AI part of the agent responsible for its move decisions. The design for the play logic—called game players—uses a well-defined interface allowing different game player implementations to conveniently plug into the layer and use its services. We have experimented with different search algorithms. For the two-player and multiplayer games, we have concentrated on simulation-based search approaches, but for the single-agent games, we have also experimented with different systematic search approaches, including A* [13] and memory-enhanced IDA* [14].

C. Game-Logic Interface

The game-logic interface encapsulates the state space of the game, provides information about available moves, and tells how a state changes when a move is made and whether the state is terminal and its goal value. It provides a well-defined interface for this called a game controller.

Once initialized by the game-agent layer, it spawns an external process for translating the previously saved GDL file description into Prolog code, using an external custom-made tool (see the File System box in Fig. 2). The generated code is—along with some prewritten Prolog code—compiled into a library responsible for all game-specific state-space manipulations. We use YAP Prolog [15] for this, mainly because it is free for academic use, reasonably efficient, and provides a convenient interface for accessing the compiled library routines from another host programming language. The game controller calls the YAP runtime engine via its C-code interface, including loading the compiled Prolog code at runtime and querying it for accessing and manipulating necessary game state information.

IV. SEARCH

The search procedure is at the heart of CADIAPLAYER. In contrast to previous successful GGP agents, which build on minimax-based alpha–beta game-tree search, our agent uses Monte Carlo simulations for its move decisions (for games we use the terms move and action interchangeably). Thus, instead of doing a look-ahead search and evaluating the resulting leaf nodes, it plays out a large number of complete games and, based on the

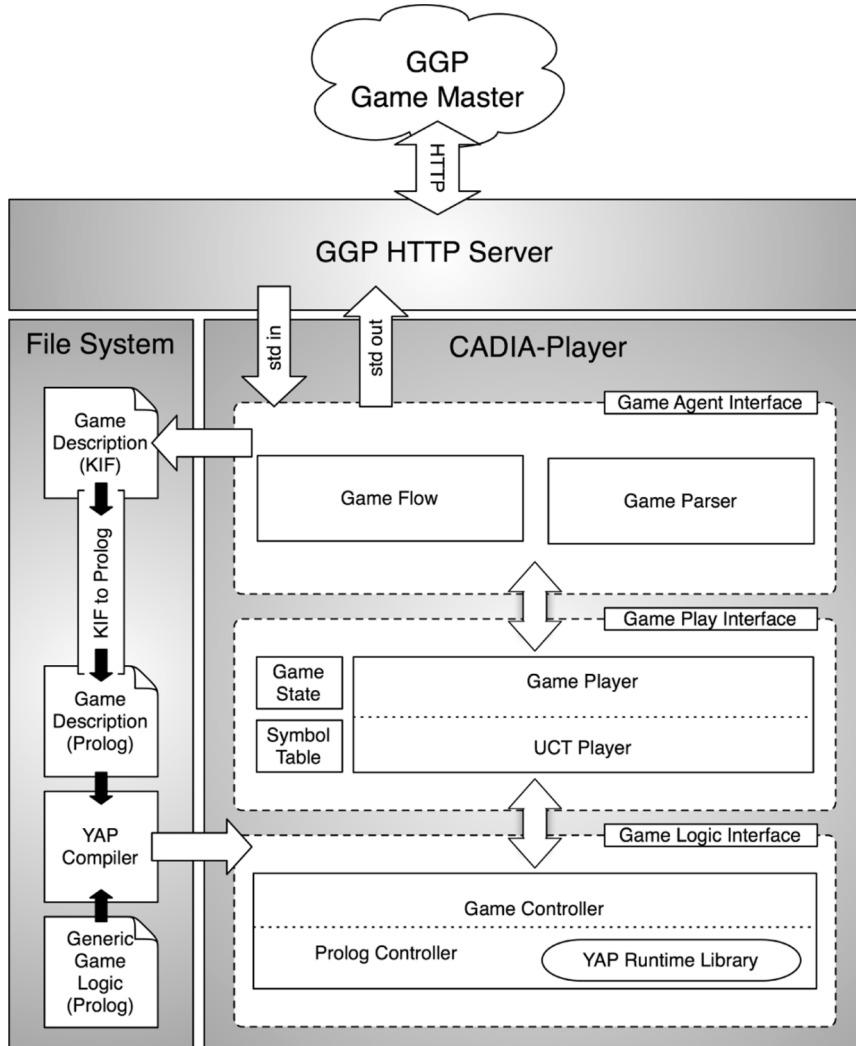


Fig. 2. Overview of the architecture of CADIAPLAYER.

result, picks the most promising continuation to play. However, as this approach is potentially computationally costly, it is critical to focus the simulations on the most relevant lines of play while still allowing for adequate exploration.

A. UCT Game Tree

The upper confidence bounds applied to trees (UCT) algorithm [16] is a generalization of the upper confidence bound (UCB1) [17] algorithm that can be applied to game trees. The algorithm uses simulations to gradually build a game tree in memory where it keeps track of the average return of each state–action pair played, $Q(s, a)$. It offers an effective and sound way to balance the exploration versus exploration tradeoff.

During a simulation, when still within the tree, it selects the action to explore by

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

The 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. The parameter can vary widely from one domain (or program) to the next, in the extreme even being set to zero. Because some programs do not rely on the UCT bonus at all, the term Monte Carlo tree search (MCTS), instead of UCT search, is now commonly used to refer collectively to methods building a game tree in this manner.

The Monte Carlo game 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 nonsimulated action is played. Also, for every simulated episode, only the first new node encountered is stored [9]. An overview of a single UCT simulation is given in Fig. 3. The start state is denoted by S , the terminal state is denoted 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.

Algorithm 1: search(ref qValues[])

```

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

```

B. Opponent Modeling

To get the best performance out of MCTS we must model not only the role CADIPLAYER plays, but also the ones of the other players. So for each opponent in the game a separate game-tree model is used. 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 CADIPLAYER. The function *make* advances the game, *retract* reverts the game to its previous state, and *search* is a recursive call. The discount factor γ is set slightly less than one, or 0.999, for the algorithm to prefer earlier rather than later payoffs, as longer playouts have higher uncertainty. A

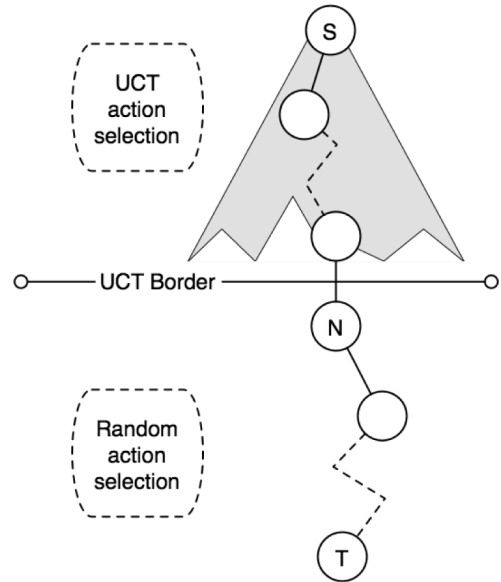


Fig. 3. Conceptual overview of a single UCT simulation.

conservative discounting value was chosen as more aggressive discounting increases the risk of shorter lines of play becoming more attractive than superior, although longer, lines of play. 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, CADIPLAYER’s model is queried for the action in the current state with the highest $Q(s, a)$ value.

C. Playouts

The UCT algorithm provides an effective mechanism for focusing the Monte Carlo simulations towards relevant lines of play. However, except towards the end of the game, the largest part of each simulation playout occurs outside the tree (below the UCT border in Fig. 3), where by default a move is chosen uniformly at random because of a lack of criteria for discriminating between the moves. The same event holds true in the UCT tree when choosing among unexplored actions for which no information has been gathered. In general, because all models play equally uninformed there is useful information to be gained even from such random simulations: the player having the better position will win more often in the long run. However, this can be problematic for games characterized by having only one good move, and can lead to a behavior where a simulation-based agent plays too “optimistically,” even relying on the opponent to make a mistake.

In Fig. 4, we see an example position from the game *Breakthrough*. The game is played on an 8×8 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

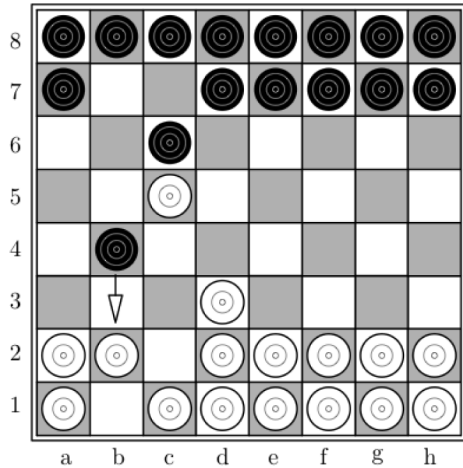


Fig. 4. Breakthrough game position.

rank. It is black’s turn to move. 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 choose white’s reply at random (or highly exploratory) have problems with a move like this one because most of the simulation play-outs 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 the playout phase, it would continue to score well (there is no memory) and erroneous information would propagate back into the UCT tree. Special preprogrammed move patterns, or ones learned offline, are used in computer Go to detect many of such only-reply moves. In GGP programs, however, this must be learned effectively during online play.

One way to add bias into the action–selection 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 heuristic* [18], which is a well-established move-ordering mechanism in chess, is based on this same principle. In an improved version of our agent [7], in addition to the action values $Q(s, a)$, the agent also keeps its average return for each action encountered independent of the state where it was played, i.e., $Q_h(a)$. 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

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

where $\mathcal{P}(a)$ is the probability that action a will be chosen in that state—actions with a high $Q_h(a)$ value are more likely. The $Q_h(a)$ 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).

D. Single-Agent Games

In the 2007 GGP competition, a rudimentary version of the memory-enhanced IDA* [14] search algorithm was used as the game player for the single-agent games. However, if no (partial) solution was found on the *startclock*, then the agent fell back on using the UCT algorithm on the *playclock*. For the 2008 GGP competition, a scheme for automatically deriving search-guidance heuristics for single-agent games was developed, using a relaxed planning graph in a similar manner as heuristic search-based planners do [19]. The heuristic was used to guide a time-bounded A*-like algorithm on both the *startclock* and the *playclock*. In the preliminaries, we initially used this scheme alongside UCT, picking the move from whichever method promised the higher score. However, the scheme was not yet in its prime and proved neither sufficiently efficient nor robust enough across different games, and was consequently suspended. Instead, we made several small adjustments to UCT to make it better suited to handle single-agent games.

First, the absence of an adversary makes play deterministic in the sense that the game will never take an unexpected turn because of an unforeseen move by the opponent. The length of the solution path, therefore, becomes irrelevant and the discount parameter unnecessary and possibly even harmful; no discounting was thus done ($\gamma = 1.0$).

Second, when deciding on a move to send to the GM, the best one available may not be the one with the highest average return. The average can hide a high goal if it is surrounded with low goals, while leading the player down paths littered with medium goals. Therefore, we also keep track of the maximum simulation score returned for each node in the UCT tree. The average score is still used for action selection during simulation play-outs, but the move finally played at the root will be the one with the highest maximum score.

Finally, we add the entire simulation path leading to a better or equal solution than previously found to the game tree, as opposed to only growing the tree one node at a time as the multiplayer UCT variant does. This guarantees that a good solution, once found, is never forgotten. The effect of this is clearly visible in Fig. 5.

Overall UCT does a reasonable job on simple single-agent puzzles, or where many (or partial) solutions are available. However, on more complex puzzles with large state spaces, it is helpless; there is really no good replacement for having a well-informed heuristics for guidance. The work on automatically finding informed heuristics by problem relaxation is thus still a work in progress.

E. Parallelization

One of the appeals of simulation-based searches is that they are far easier to perform in parallel than a traditional game-tree search because of fewer synchronization points. CADIAPLAYER supports running simulations in parallel, and typically uses 4–12 central processing units (CPUs) concurrently. The UCT tree is maintained by a master process and whenever crossing the UCT

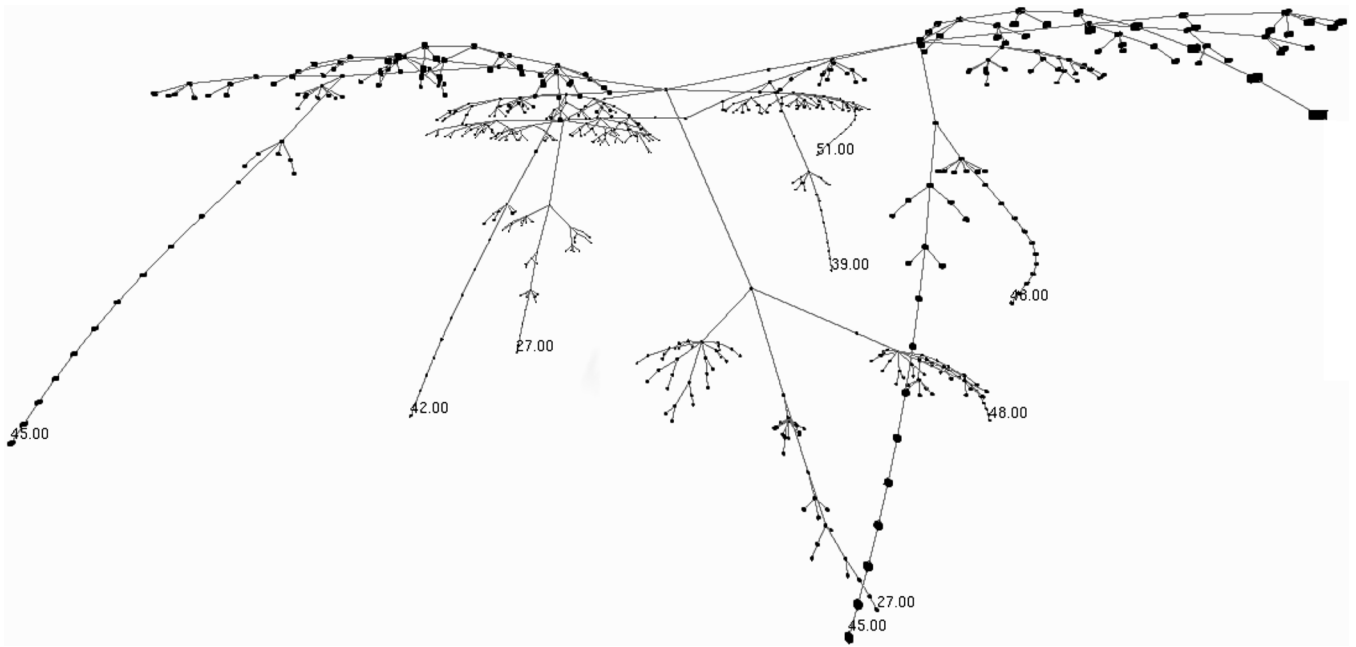


Fig. 5. Visualization (3-D) of our enhanced UCT tree for single-player games made from real data when playing *Knights Tour*. The numbers show goals at terminal states with better or equal score to the best one currently known at the time.

border, falling out of the tree, it generates and delegates the actual playout work to a client process if available, otherwise the master does the playout. To better balance the workload, a client may be asked to perform several simulations from the leaf position before returning. This parallelization scheme is based on ideas presented in [20]. Further work on parallelization schemes for Monte Carlo tree search are presented in [21] and [22].

As we typically run CADIAPLAYER on a cluster of workstations using distributed memory, one would ideally have to broadcast the $Q_h(a)$ values to all the slave processes. However, for more complex games, we have found this not to be cost effective, and thus each slave maintains its own set of $Q_h(a)$ values calculated only based on results from local simulations.

V. EMPIRICAL EVALUATION

The UCT simulation-based approach has already proved its effectiveness against traditional game-tree search players. In the preliminary rounds of the 2007 competition, played over a period of four weeks, the agent played a large number of match games using over 40 different types of game. 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%. CADIAPLAYER also came out on top in the 2008 preliminaries. Fewer games were played that year and the scores were closer. The GGP competition finals held at the Association for the Advancement of Artificial Intelligence (AAAI) conference use a knockout format with only a handful of games each round, in part to provide some excitement for the audience. As mentioned before, CADIAPLAYER won the finals both in 2007 and 2008, although a luck factor inevitably plays some role when only a handful of games are played in a knockout tournament.

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

A. 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 use CP_{uct} to refer to a version of CADIAPLAYER using random action selection when the UCT border is crossed, and CP_{imp} to refer to the current CADIAPLAYER with the playout enhancements. The value of 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 dedicated Linux-based dual processor Intel(R) Xeon(TM) 3.20-GHz CPU computers with 2 GB of RAM. Each agent used a single processor. For each game, both the start- and the playclocks were set to 10 s. Most of the GDL descriptions of the games can be downloaded from the official GGP game server at Stanford University.

B. UCT Versus MC

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

TABLE I
RESULTS OF UCT AND MC MATCHES IN PERCENT

Game	Player	MC _{org}	MC _{mem}	CP _{uct}	Average
<i>Connect-4</i>	MC _{org}	N/A	29.0 (± 5.00)	6.8 (± 3.08)	17.9 (± 3.33)
	MC _{mem}	71.0 (± 5.59)	N/A	15.8 (± 4.44)	43.4 (± 4.31)
	CP _{uct}	93.2 (± 3.08)	84.2 (± 4.44)	N/A	88.7 (± 2.73)
<i>Checkers</i>	MC _{org}	N/A	14.6 (± 4.08)	6.8 (± 2.87)	10.7 (± 2.51)
	MC _{mem}	85.4 (± 4.08)	N/A	31.2 (± 5.37)	58.3 (± 4.12)
	CP _{uct}	93.2 (± 2.87)	68.8 (± 5.37)	N/A	81.0 (± 3.22)
<i>Othello</i>	MC _{org}	N/A	39.8 (± 5.99)	23.2 (± 5.12)	31.5 (± 3.78)
	MC _{mem}	60.2 (± 5.99)	N/A	26.6 (± 5.36)	43.4 (± 4.28)
	CP _{uct}	76.8 (± 5.12)	73.4 (± 5.36)	N/A	75.1 (± 3.71)
<i>Breakthrough</i>	MC _{org}	N/A	38.0 (± 6.03)	9.6 (± 3.66)	23.8 (± 3.74)
	MC _{mem}	62.0 (± 6.03)	N/A	18.4 (± 4.81)	40.2 (± 4.30)
	CP _{uct}	90.4 (± 3.66)	81.6 (± 4.81)	N/A	86.0 (± 3.04)
<i>Amazons Small</i>	MC _{org}	N/A	41.2 (± 6.09)	15.6 (± 4.37)	28.4 (± 3.91)
	MC _{mem}	58.8 (± 6.09)	N/A	21.8 (± 4.99)	40.3 (± 4.25)
	CP _{uct}	84.4 (± 4.37)	78.2 (± 4.99)	N/A	81.3 (± 3.32)
<i>Pentago</i>	MC _{org}	N/A	38.4 (± 5.99)	11.0 (± 3.74)	24.7 (± 3.73)
	MC _{mem}	61.6 (± 5.99)	N/A	26.6 (± 5.39)	44.1 (± 4.31)
	CP _{uct}	89.0 (± 3.74)	73.4 (± 5.39)	N/A	81.2 (± 3.35)
<i>Quarto</i>	MC _{org}	N/A	40.6 (± 5.69)	9.8 (± 2.82)	25.2 (± 3.45)
	MC _{mem}	59.4 (± 5.69)	N/A	11.4 (± 2.94)	35.4 (± 3.83)
	CP _{uct}	90.2 (± 2.82)	88.6 (± 2.94)	N/A	89.4 (± 2.03)
<i>TCCC</i>	MC _{org}	N/A	44.0 (± 6.14)	10.8 (± 3.86)	27.4 (± 3.90)
	MC _{mem}	56.0 (± 6.14)	N/A	18.6 (± 4.78)	37.3 (± 4.22)
	CP _{uct}	89.2 (± 3.86)	81.4 (± 4.78)	N/A	85.3 (± 3.09)

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 but is allowed to build a top-level game tree, adding one node per simulation, as CP_{uct} does.

Eight different two-player games were used in the experiments: *Connect-4*, *Checkers*, *Othello*, *Breakthrough*, *Amazons Small* (6×6), *Pentago*, *Quarto*, and *TCCC* (an interesting hybrid of *Tic-Tac-Toe*, *Checkers*, *Chess*, and *Connect-4* that was played at the 2008 GGP competition finals).

The match results are shown in Table I. The UCT player outperforms both baseline players by a large margin in all eight games, with an impressive average winning percentage ranging from just over 75% up to about 90%. 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 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 on a few games, and CP_{uct} and MC_{mem} perform on average around 35% (*Othello*) to approximately 100% (*Checkers*, *Connect-4*) more simulations than MC_{org} does. The added number of simulations explains some of the performance increase.

C. Biased Action-Selection Enhancement

The following experiment evaluates the biased action-selection enhancement. The τ parameter of the Gibbs distribution

TABLE II
TOURNAMENT BETWEEN CP_{uct} AND CP_{imp}

Game	CP _{imp} win %
Connect-4	49.2 (± 6.03)
Checkers	55.4 (± 5.93)
Othello	59.0 (± 6.05)
Breakthrough	88.8 (± 3.92)
Amazons Small	55.0 (± 6.04)
Pentago	56.0 (± 6.01)
Quarto	50.6 (± 4.50)
TCCC	75.6 (± 5.25)

was set to 10 (based on trial-and-error on a small number of games). The result is shown in Table II.

The new action-selection scheme offers some benefits for almost all the games. 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. Also, we noticed in *Breakthrough*, in particular, that focussing the simulations resulted in shorter episodes, allowing up to three times as many simulations to be performed in the same amount of time. The scheme also offers significant improvements in the game of *TCCC* and *Othello*, but in the latter 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 a deciding factor. The ineffectiveness in *Connect-4* and *Quarto* is most likely because of their rule's simplicity and the absence of consistently good moves.

D. 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 III. Moreover, there are no signs yet of diminishing performance improvement as the time controls are

TABLE III
TIME-CONTROL COMPARISON FOR CP_{uct}

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

TABLE IV
COMPARISON OF CP_{imp} WITH AND WITHOUT SINGLE PLAYER ENHANCEMENTS

Game	CP_{imp} Avg.	Enh. CP_{imp} Avg.
Asteroids	53.50 (\pm 2.51)	74.00 (\pm 4.92)
Peg	87.30 (\pm 0.87)	90.30 (\pm 0.34)
Pancakes	86.25 (\pm 2.16)	94.75 (\pm 1.23)
Asteroids Serial	55.75 (\pm 2.19)	78.00 (\pm 1.60)
State Space Large	28.00 (\pm 0.00)	31.01 (\pm 1.33)
Average	62.16 (\pm 2.12)	73.61 (\pm 2.27)

raised (although inevitably they will at some point show up). This is positive and indicates that simulation-based approaches will probably continue to gain momentum with more massive multicore CPU technology. It is worth noting that a simulation-based search is much easier to parallelize than traditional game-tree search algorithms.

E. UCT Single-Agent Enhancements

To examine the effect of the single-agent enhancements, we ran comparison experiments using five different games, where each data point consists of 100 matches. We used the CP_{imp} player both with and without the enhancements. The start- and playclocks were both set to 10 s. The games all have partial goals and are difficult to solve perfectly given these time controls. As can be seen in Table IV, UCT does a reasonable job solving the games, but more importantly, the enhancements add about ten points on average, and for some games even over 20 points. The improved UCT version does even slightly outperform our A*-based solver.

VI. RELATED WORK

One of the first general game-playing systems was Pell's METAGAMER [23], which played a wide variety of simplified chess-like games. The winners of the 2005 and 2006 GGP competition were CLUNEPLAYER and FLUXPLAYER, respectively. The main research focus of both agents has been on automatic feature discovery for building a heuristic evaluation function for use by a traditional game-tree search.

The CLUNEPLAYER [4] agent creates abstract models from the game descriptions that incorporate essential aspects of the original game, such as payoff, control, and termination. The agent then identifies stable features through sampling, which are then used for fitting the models using regression. Since winning the GGP competition in 2005, the agent has finished second in all subsequent GGP competitions. In the 2008 competition, the agent had dual capabilities such that it could choose between using either minimax-based or Monte Carlo simulation search,

based on game properties. A thorough analysis of the relative effectiveness of Monte Carlo simulations versus minimax search in CLUNEPLAYER is provided for different games in [24].

The FLUXPLAYER agent [5], [25] uses fluent calculus (an extension of situated calculus) for reasoning about actions and for generating game states. Standard game-tree search techniques are used for the planning phase, including nonuniform depth-first search, iterative deepening, transposition tables, and history heuristic. The heuristic function evaluation is based on fuzzy logic where semantic properties of game predicates are used for detecting static structures in the game descriptions. The system won the 2006 GGP competition, and came second in 2005.

UTEXAS LARG is another prominent GGP agent using a traditional game-tree method [6]. The agent is reinforcement-learning based and is capable of transferring knowledge learned in one game to expedite learning in other unseen games via so-called value-function transfer (general features are extracted from the state space of one game and used to seed the learning in a state space of a different game) [26], [27]. Part of the problem is to recognize which games are similar enough for the transfer learning to be applicable; this is done with graph-based domain mapping [28]. The system is also highly parallel, capable of running concurrently on many processors. It was among the top-place finishers in the first few GGP competitions, although it has never placed first.

Besides CADIPLAYER [7], [29], two agents that participated in the 2007 GGP competition, ARY and JIGSAWBOT, used Monte Carlo simulation search, although not UCT according to their authors (personal communication, July 2007). In the 2008 competition, ARY had incorporated the UCT enhancement, as well as a new entry called MALIGNÉ.

MCTS has been used successfully to advance the state-of-the-art in computer Go, and is used by several of the strongest Go programs, e.g., MOGO [8] and CRAZYSTONE [9]. Experiments in Go showing how simulations can benefit from using an informed playout policy are presented in [30]. 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 MCTS by using a search-control technique called rapid action value estimation (RAVE). It is based on the *all-moves-as-first* principle, where actions from the entire playout episode may invoke changes in the top-level tree. There are fundamental differences between that method and the one we propose here for search control, the main one being that data are propagated differently: in our method from the top-level tree down to the playout phase, whereas in the aforementioned method, it is the other way around. Search-control learning in traditional game-tree search has been studied, for example, in [31].

Other work on general game playing include a coevolution approach that allows algorithm designers to both minimize the amount of domain knowledge built into the system and model opponent strategies more efficiently [32], and a logic program approach where the game descriptions are translated into a specialized evaluator that works with decomposed logical features for improving accuracy and efficiency [33]. However, neither of the approaches mentioned above has yet developed into a fully fledged competitive general game-playing system.

VII. CONCLUSION

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 a 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* and *Amazons*), where each move advances the game towards the end, as this bounds the length of each simulation payout. 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 nonterminal 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 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 (yet) show diminishing returns in performance improvement as thinking time is increased, thus promising to take full advantage of future massively multicore CPUs.

Search-control heuristics are important for guiding the simulation payouts. We introduced one promising domain-independent search-control method that increased our agent’s playing strength on most games we tried it on, in the best case defeating a standard UCT player with close to a 90% winning score, and currently we have not seen any indication of the method being detrimental to any game. Also, the agent’s effectiveness on single-agent games was improved with a couple of minor adjustments to the UCT search.

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 payouts, 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 payouts 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 nonterminal states when simulations are terminated prematurely.

APPENDIX
COMPLETE GDL FOR TIC-TAC-TOE

```
(role xplayer)
(role oplayer)
(init (cell 1 1 b))
(init (cell 1 2 b))
(init (cell 1 3 b))
(init (cell 2 1 b))
(init (cell 2 2 b))
(init (cell 2 3 b))
(init (cell 3 1 b))
(init (cell 3 2 b))
(init (cell 3 3 b))
(init (control xplayer))
(<=(next (cell ?m ?n x))
  (does xplayer (mark ?m ?n))
  (true (cell ?m ?n b)))
(<=(next (cell ?m ?n o))
  (does oplayer (mark ?m ?n))
  (true (cell ?m ?n b)))
(<=(next (cell ?m ?n ?w))
  (true (cell ?m ?n ?w))
  (distinct ?w b))
(<=(next (cell ?m ?n b))
  (does ?w (mark ?j ?k))
  (true (cell ?m ?n b))
  (distinct ?m ?j))
(<=(next (cell ?m ?n b))
  (does ?w (mark ?j ?k))
  (true (cell ?m ?n b))
  (distinct ?n ?k))
(<=(next (control xplayer))
  (true (control oplayer)))
(<=(next (control oplayer))
  (true (control xplayer)))
(<=(row ?m ?x)
  (true (cell ?m 1 ?x))
  (true (cell ?m 2 ?x))
  (true (cell ?m 3 ?x)))
(<=(column ?n ?x)
  (true (cell 1 ?n ?x))
  (true (cell 2 ?n ?x))
  (true (cell 3 ?n ?x)))
(<=(diagonal ?x)
  (true (cell 1 1 ?x))
  (true (cell 2 2 ?x)))
```

```

      (true (cell 3 3 ?x)))
(<=(diagonal ?x)
  (true (cell 1 3 ?x))
  (true (cell 2 2 ?x))
  (true (cell 3 1 ?x)))
(<=(line ?x)
  (row ?m ?x))
(<=(line ?x)
  (column ?m ?x))
(<=(line ?x)
  (diagonal ?x))
(<=open
  (true (cell ?m ?n b)))
(<=(legal ?w (mark ?x ?y))
  (true (cell ?x ?y b))
  (true (control ?w)))
(<=(legal xplayer noop)
  (true (control oplayer)))
(<=(legal oplayer noop)
  (true (control xplayer)))
(<=(goal xplayer 100)
  (line x))
(<=(goal xplayer 50)
  (not (line x))
  (not (line o))
  (not open))
(<=(goal xplayer 0)
  (line o))
(<=(goal oplayer 100)
  (line o))
(<=(goal oplayer 50)
  (not (line x))
  (not (line o))
  (not open))
(<=(goal oplayer 0)
  (line x))
(<=terminal
  (line x))
(<=terminal
  (line o))
(<=terminal
  (not open))

```

REFERENCES

- [1] M. Campbell, A. J. Hoane Jr., and F.-H. Hsu, "Deep blue," *Artif. Intell.*, vol. 134, no. 1–2, pp. 57–83, 2002.
- [2] J. Schaeffer, *One Jump Ahead: Challenging Human Supremacy in Checkers*. New York: Springer-Verlag, 1997.
- [3] M. Buro, "How machines have learned to play Othello," *IEEE Intell. Syst.*, vol. 14, no. 6, pp. 12–14, Nov./Dec. 1999 [Online]. Available: <http://www.cs.ualberta.ca/~mburo/ps/IEEE.pdf>
- [4] J. Clune, "Heuristic evaluation functions for general game playing," in *Proc. 22nd AAAI Conf. Artif. Intell.*, 2007, pp. 1134–1139.
- [5] S. Schiffel and M. Thielscher, "Fluxplayer: A successful general game player," in *Proc. 22nd AAAI Conf. Artif. Intell.*, 2007, pp. 1191–1196.
- [6] G. Kuhlmann, K. Dresner, and P. Stone, "Automatic heuristic construction in a complete general game player," in *Proc. 21st Nat. Conf. Artif. Intell.*, Jul. 2006, pp. 1457–1462.
- [7] H. Finnsson and Y. Björnsson, "Simulation-based approach to general game playing," in *Proc. 23rd AAAI Conf. Artif. Intell.*, D. Fox and C. P. Gomes, Eds., Chicago, IL, Jul. 13–17, 2008, pp. 259–264.
- [8] S. Gelly, Y. Wang, R. Munos, and O. Teytaud, "Modification of UCT with patterns in Monte-Carlo Go," INRIA, Orsay Cedex, France, Tech. Rep. 6062, 2006.
- [9] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *Proc. 5th Int. Conf. Comput. Games*, 2006, pp. 72–83 [Online]. Available: <http://remi.coulom.free.fr/CG2006/>
- [10] N. Love, T. Hinrichs, and M. Genesereth, "General game playing: Game description language specification," Stanford Univ., Stanford, CA, Tech. Rep. April 4 2006 [Online]. Available: <http://games.stanford.edu/>
- [11] M. R. Genesereth and R. E. Fikes, "Knowledge interchange format, version 3.0 reference manual," Stanford Univ., Stanford, CA, Tech. Rep. Logic-92-1, 1992.
- [12] M. R. Genesereth, N. Love, and B. Pell, "General game playing: Overview of the AAAI competition," *AI Mag.*, vol. 26, no. 2, pp. 62–72, 2005.
- [13] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. SSC-4, no. 2, pp. 100–107, Jul. 1968.
- [14] A. Reinefeld and T. A. Marsland, "Enhanced iterative-deepening search," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 16, no. 7, pp. 701–710, Jul. 1994 [Online]. Available: citeseer.ist.psu.edu/reinefeld94enhanced.html
- [15] V. S. Costa, L. Damas, R. Reis, and R. Azevedo, "YAP prologue user's manual," 2006 [Online]. Available: <http://www.ncc.up.pt/~vsc/Yap/documentation.html>, Retrieved Jan. 27, 2008
- [16] L. Kocsis and C. Szepesvari, "Bandit based Monte-Carlo planning," in *Proc. Eur. Conf. Mach. Learn.*, 2006, pp. 282–293.
- [17] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Mach. Learn.*, vol. 47, no. 2/3, pp. 235–256, 2002 [Online]. Available: citeseer.ist.psu.edu/auer00finite-time.html
- [18] J. Schaeffer, "The history heuristic and alpha-beta search enhancements in practice," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-11, no. 11, pp. 1203–1212, Nov. 1989 [Online]. Available: citeseer.ist.psu.edu/schaeffer89history.html
- [19] J. Hoffmann and B. Nebel, "Fast plan generation through heuristic search," *J. Artif. Intell. Res.*, vol. 14, pp. 2001–, 2001.
- [20] T. Cazenave and N. Jouandeau, "On the parallelization of UCT," in *Proc. Comput. Games Workshop*, Jun. 2007, pp. 93–101 [Online]. Available: <http://www.ai.univ-paris8.fr/~n/pub/indexPub.html#cgw07>
- [21] G. Chaslot, M. H. M. Winands, and H. J. van den Herik, "Parallel Monte-Carlo tree search," *Comput. Games*, vol. 5131, pp. 60–71, 2008.
- [22] T. Cazenave and N. Jouandeau, "A parallel Monte-Carlo tree search algorithm," *Comput. Games*, vol. 5131, pp. 72–80, 2008.
- [23] B. Pell, "A strategic metagame player for general chess-like games," *Comput. Intell.*, vol. 12, pp. 177–198, 1996 [Online]. Available: citeseer.ist.psu.edu/pell94strategic.html
- [24] J. E. Clune, "Heuristic evaluation functions for general game playing," Ph.D. dissertation, Dept. Comput. Sci., Univ. California, Los Angeles, CA, 2008.
- [25] S. Schiffel and M. Thielscher, "Automatic construction of a heuristic search function for general game playing," in *Proc. 7th IJCAI Int. Workshop Nonmonotonic Reasoning, Action Change*, 2007.
- [26] B. Banerjee, G. Kuhlmann, and P. Stone, "Value function transfer for general game playing," in *Proc. ICML Workshop Structural Knowl. Transfer for ML*, Jun. 2006.
- [27] B. Banerjee and P. Stone, "General game learning using knowledge transfer," in *Proc. 20th Int. Joint Conf. Artif. Intell.*, Jan. 2007, pp. 672–677.
- [28] G. Kuhlmann and P. Stone, "Graph-based domain mapping for transfer learning in general games," in *Proc. 18th Eur. Conf. Mach. Learn.*, Sep. 2007, vol. 4701, pp. 188–200.

- [29] H. Finnsson, "CADIA-Player: A general game playing agent," M.S. thesis, School Comput. Sci., Reykjavík Univ., Reykjavík, Iceland, Dec. 2007.
- [30] S. Gelly and D. Silver, "Combining online and offline knowledge in UCT," in *Proc. Int. Conf. Mach. Learn.*, Z. Ghahramani, Ed., 2007, vol. 227, pp. 273–280.
- [31] Y. Björnsson, "Selective depth-first game-tree search," Ph.D. dissertation, Dept. Comput. Sci., Univ. Alberta, Edmonton, AB, Canada, 2002.
- [32] J. Reisinger, E. Bahceci, I. Karpov, and R. Miikkulainen, "Coevolving strategies for general game playing," in *Proc. IEEE Symp. Comput. Intell. Games*, 2007, pp. 320–327.
- [33] T. Kaneko, K. Yamaguchi, and S. Kawai, "Automatic feature construction and optimization for general game player," in *Proc. Game Programm. Workshop*, 2001, pp. 25–32.



Yngvi Björnsson received the Ph.D. degree in computer science from the Department of Computing Science, University of Alberta, Edmonton, AB, Canada, in 2002.

Currently, he is an Associate Professor at the School of Computer Science, Reykjavík University, Reykjavík, Iceland, as well as a Co-Director (and co-founder) of the Center for Analysis and Design of Intelligent Agents (CADIA) research lab. He is a coauthor of the general game-playing agent CADIAPLAYER. His research interests are in heuristic

search methods and search-control learning, and the application of such techniques for solving large scale problems in a wide range of problem domains, including computer games and industrial process optimization.

Dr. Björnsson regularly serves on program committees of major artificial intelligence and computer games conferences, including the senior program committee of the 2009 International Joint Conference on Artificial Intelligence.



Hilmar Finnsson received the M.Sc. degree from the School of Computer Science, Reykjavík University, Reykjavík, Iceland, in 2008, where he is currently working towards the Ph.D. degree in computer science.

His research focus is on heuristic search and learning in the context of general game playing. He is a coauthor of the general game-playing agent CADIAPLAYER.