An Automatically-Generated Evaluation Function in General Game Playing

Karol Walędzik and Jacek Mańdziuk, *Senior Member, IEEE*

***Abstract*—General Game Playing competitions provide a framework for building multigame playing agents. In this paper, we describe an attempt at the implementation of such an agent. It relies heavily on our knowledge-free method of automatic construction of approximate state evaluation function based on game rules only. This function is then employed by one of the two game tree search methods: MTD(f) or Guided UCT, the latter being our proposal of an algorithm combining UCT with the usage of an evaluation function. The performance of our agent is very satisfactory when compared to a baseline UCT implementation.**

***Index Terms*—General Game Playing, UCT, MTD, evaluation function, autonomous game playing.**

I. Introduction

GAMES, some of them played by humans for thousands of years (Go and backgammon date back to 1000-2000 BC), are a natural and interesting topic for Artificial Intelligence (AI) and Computational Intelligence (CI) research. They provide well-defined environments for devising and testing possibly sophisticated strategies. Most of them are not simple enough to be solved by brute-force algorithms, yet they are limited enough to allow concentration on the development of an intelligence algorithm, instead of handling environment complexities.

Years of research on particular games led to a situation in which the goal of significantly outplaying top human players has been achieved in almost all the popular games (Go being the most notable exception). Several moderately sophisticated games, most notably Checkers, have been solved [23].

Yet, almost all of those so successful applications involved no learning mechanisms whatsoever, relying instead on fast and heavily optimized search algorithms coupled with manually-tuned game-specific heuristic state evaluation functions, defined by human players with significant experience in a given game. None of the top programs is actually able to transfer its game-playing abilities to the domain of another game, no matter how similar to the one it was intended for.

In this paper, continuing our research presented in [28], we deal with the topic of constructing a multigame playing agent, i.e. program capable of playing a wide variety of games, in our case defined by *Game Description Language* (GDL) – part of the *General Game Playing* (GGP) competition specification. The multitude of possible games prevents introduction of any significant amount of external knowledge or preconceptions into the application. It is expected that it will rely on some kind of learning algorithm, allowing it to reason about the game and come up with a successful playing strategy based solely on the rules (and possibly experience gathered during the play itself).

Faculty of Mathematic and Information Science, Warsaw University of Technology, Koszykowa 75, 00-662 Warsaw, Poland, e-mail: {k.waledzik, j.mandziuk}@mini.pw.edu.pl

Our GGP agent relies on a fully automatically generated state evaluation function. Its construction process interweaves two human-specific, cognitively-inspired processes of *generalization* and *specialization* for generating a pool of possible game state features. Those features are then combined into the final evaluation function depending on the results of their statistical analysis. This function is then employed by one of the two game tree search algorithms. The first one is a modification of the standard MTD(f) method. The second one, which we call *Guided UCT* (GUCT) is our own proposal of an algorithm, combining typical UCT with an evaluation function usage.

While the UCT algorithm itself can operate with no explicit evaluation function and some applications of it have proven successful in the context of multigame playing, we strongly believe that its quality can be improved by the introduction of a state evaluator. After all, game tree search algorithms combined with heuristic evaluation functions have remained the standard and most powerful approach to single game agents creation for years, and for a good reason. An intelligent mixture of both mentioned approaches should lead to the best of both worlds solution.

Recent, limited use of the evaluation functions in the context of GGP has probably been caused by the difficulties in their generation for arbitrary games. In this paper we prove, however, that this task is feasible, at least for some of the GGP games and, with further refinement, can form at least part of a successful GGP approach.

It is also worth noting that, as presented in the appendices, our approach generates evaluation functions in a form that can be relatively easily understood and analyzed by humans. This, in effect, is an important step in a transition from a black-box UCT algorithm towards a white-box approach to multigame playing, in which an agent's decisions can be analyzed in order to further tweak its operation and improve playing strength.

The remainder of this paper is divided into several sections. In the first three, we briefly introduce the concept of GGP and basic UCT and MTD(f) algorithms. Afterwards, we describe in detail the process of building the state evaluation function and incorporating it into our solution - this includes the specification of the GUCT method. Finally, we present the promising results of several experiments we have performed, conclude our research and provide suggestions for its further development.

```
;Roles:
(role x) (role y)
;Initial state:
(init (cell 1 1 b))
(init (cell 1 2 b))
...
(init (control x))
;Rules:
(<= (next (cell ?x ?y ?player))
     does (?player (mark ?x ?y)))
(<= (next (control x)) (true (control o)))
...
(<= (line ?player) (row ?x ?player))
;Legal moves:
(<= (legal ?player (mark ?x ?y))
     (true (cell ?x ?y b))
     (true (control ?player)))
...
;Goals:
(<= (goal ?player 100) (line ?player))
...
;Terminal:
(<= terminal (line ?player))
...
```

Fig. 1. A condensed Tic-Tac-Toe game description in GDL [16]

II. General Game Playing

GGP is one of the latest and currently most popular approaches to multigame playing. It stems from Stanford University, where the annual General Game Playing Competition [10], [8] has been proposed for the first time. GGP agents are expected to be able to play a broad class of games whose rules can be expressed in GDL. Competitions typically encompass a number of varied games, including one-player puzzles, as well as multi-player competitive and cooperative games. Some of the games are based on real-world ones while others are designed from scratch specifically for the tournament.

GGP matches are coordinated by the so-called game server. Firstly, it sends the rules of the game (in the form of a set of GDL statements) to all participants, then contestants (programs) are allotted a limited amount of time (typically from several seconds to several minutes, depending on the complexity of the game) to study the rules and devise a reasonable playing strategy. Afterwards, the match is played with a time limit per move (again, typically of significantly less than one minute).

Since games are not assigned names and their rules are provided separately for each match, knowledge transfer between matches is severely limited. Additionally, a GGP agent knows nothing about its opponents and, therefore, cannot modify its strategy based on previous experience with the same opponent(s).

A. Game Description Language

GDL is a variant of Datalog which can, in principle, be used to define any finite, discrete, deterministic multi- or single-player game of complete information. In GDL, game states are represented by sets of facts. Logical inferences define the rules for computing the legal actions for all players, subsequent game states, termination conditions and final scores. Yet, GDL remains fairly simple and requires only a handful of predefined relations:

role(p)	defining p as a player;
init(f)	stating that f is true in the initial state of the game;
true(f)	stating that f holds in the current game state;
does(p, m)	stating that player p performs move m;
next(f)	stating that f will be true in the next state (reached after all actions defined by $does$ relations are performed);
legal(p, m)	stating that it is legal in the current state for the player p to perform move m;
goal(p, v)	stating that, should the current state be terminal, the score of player p would be v;
terminal	stating that the current state is terminal.

An excerpt from a simple game (Tic-Tac-Toe) description can be seen in figure 1. First, it defines players taking part in the game via the $role$ relation. Afterwards the initial game state is described by means of a predefined predicate $init$ and game-specific function $cell$ defining the contents of each square of the game board. Rules for state transitions and moves legality follow. Finally, game termination conditions are defined, again based on a game-specific predicate $line$.

GDL makes use of an easy-to-parse prefix notation and relies heavily on the usage of game-specific predicates and variables (prefixed with '?'). We encourage interested readers to consult [16] for a more detailed description of GDL.

GDL, being a variant of Datalog, is easily translatable to Prolog (or any other logic programming language). In fact, this is exactly what we do in our implementation of GGP agent, using Yap [30] as our main game engine.

Recently, some extensions to GDL have been proposed: GDL-II allows defining non-deterministic games and, furthermore, last year's GGP tournament games included two additional predefined relations, allowing easier identification of all possible game-state description facts. All research presented in this paper is, however, based on the original GDL specification.

III. Game Tree Analysis

Most of the approaches to the implementation of game-playing agents (be it a GGP agent, or a single-game one) rely on representing the game as a tree (or directed acyclic graph if equivalent positions reached by different sequences of moves are not differentiated). Nodes of such a tree represent positions (game states) and arcs – possible transitions between states (i.e. players' actions).

These trees are then analyzed by dedicated algorithms. Two of them are nowadays dominant: alpha-beta pruning [12] (including its various modifications) and UCT [13]. They are

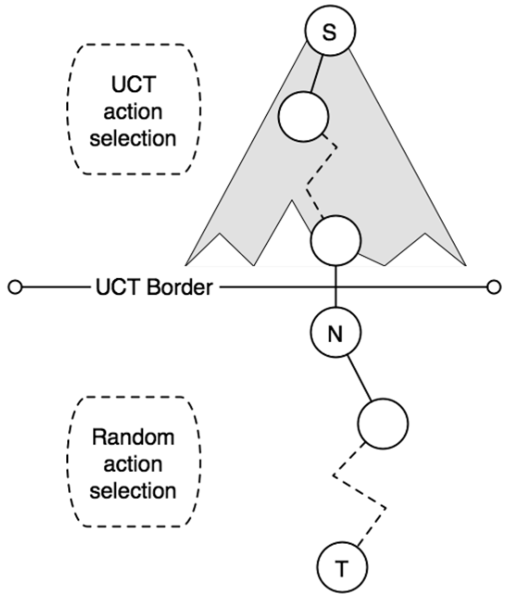

Fig. 2. Conceptual overview of a single simulation in UCT algorithm [4]

both employed by our GGP agent and described in the next subsections. For the sake of space savings the description is only brief. Please consult the original papers for more information.

A. UCT

Upper Confidence bounds applied to Trees (UCT) is a simulation-based game tree analysis algorithm [13]. It repeatedly proves to be relatively successful in the case of some very difficult games, including Go [6]. Many of the GGP tournament winners so far have relied heavily on the UCT algorithm [4], [20].

UCT is based on the UCB1 [1] algorithm for solving a K-armed bandit problem. The K-armed bandit problem deals with the issue of finding an optimal strategy for interacting with K gambling machines (or, alternatively, one machine with K play modes - arms). The rewards of each machine/arm are presumed to be nondeterministic but, nevertheless, stable (their distributions should not change in time) and pairwise independent.

The UCT algorithm assumes that the problem of choosing an action to perform in a given game state is actually equivalent to solving a K-armed bandit problem. While this may not be strictly true, it is expected to be a sufficiently good approximation.

UCT consists of performing multiple simulations of possible continuations of the game from the current state. Instead of performing fully random rollouts, as in the case of Monte-Carlo sampling, it proposes a more intelligent approach to choosing continuations worth analyzing so that the obtained results are more meaningful.

In each state, following the principles of UCB1, UCT advises to first try each action once and then, whenever the same position is analyzed again, choose the move a^* according

```
**function** MTD
1: $f^+ := MAX$; $f^- := MIN$;
2: **repeat**
3:     $b := (f^+ + f^-)/2 + \frac{\varepsilon}{2}$;
4:     $g := alphabeta(s, b - \varepsilon, b)$;
5:     **if** ( $b - \varepsilon < g < b$ ) **then return** $g$;
6:     **if** ( $g < b$ ) **then** $f^+ := g$ **else** $f^- := g$;
7: **until** forever
```

Fig. 3. MTD(f) algorithm for real-valued evaluation function [17]

to the following formula:

$$a^* = argmax_{a \in A(s)} \{Q(s,a) + C\sqrt{\frac{\ln N(s)}{N(s,a)}}\}, \qquad (1)$$

where A(s) is a set of all actions possible in state s, $Q(s,a)$ denotes the average result of playing action a in state s in the simulations performed so far, N(s) – number of times state s has been visited and $N(s,a)$ – number of times action a has been sampled in this state. Constant C controls the balance between exploration and exploitation, since the formula postulates choosing actions with the highest expected rewards, however, at the same time, avoiding repetitive sampling of only one action while others might yet prove more beneficial.

A direct implementation of the approach described above is, of course, infeasible due to memory limitations. With sufficient time it would lead directly to storing the whole game tree in memory, which is impossible in the case of all but the simplest games. Therefore, each simulation actually consists of two-phases: strict UCT phase and Monte-Carlo (random rollout) phase (see figure 2). An in-memory game tree is built iteratively and in each simulated episode the UCT strategy is applied only until the first not-yet-stored game state is reached. This state is then added to the in-memory game tree, its first action to be tested is chosen and a strictly random Monte Carlo rollout is performed further down the tree. No data is retained about the positions encountered during this phase of the simulation. All the nodes visited during the strict UCT phase have their $N(s,a)$ and $N(s)$ values incremented and their $Q(s,a)$ values updated with the final value of the rollout.

Additionally, as the game progresses, the part of the in-memory game tree not reachable from the current game state is discarded. In our implementation, in order to further optimize the algorithm, we make use of the fact that some states may be reached in several ways (by performing different move sequences) and, therefore, as mentioned earlier, the game should be represented as a directed acyclic graph rather than a tree. This does not, however, influence the basis of UCT algorithm operation.

With UCT analysis finished, choosing the next move is simply a matter of finding the action with the highest $Q(s,a)$ value in the current state.

B. MTD(f)

MTD(f) [21] (a member of the family of Memory Enhanced Test Driver algorithms) is one of the popular algorithms based on alpha-beta pruning [12].

MTD(f) consists of multiple zero-window alpha-beta-with-memory searches. Every such search is going to fail high or low, thus providing, respectively, a low or high bound on the actual value of the node. Convergence of the bounds indicates that the true value has been found. In the case of real-valued evaluation functions, true zero-window search is, obviously, not possible. Instead, the searches are performed with a narrow (but not-null) window and a bisection method is used to converge on the final value. A pseudocode for this algorithm is presented in figure 3. Our GGP agent employs an iterative-deepening version of the real-valued MTD(f) method.

IV. Evaluation Function

A. Evaluation Function Construction

Building an evaluation function for a GGP player is inherently different and much more difficult than building an evaluation function for any single-game application. While in the latter case a lot is usually known about the game at hand (or can be deduced by the function developer), in the former, one has to create an agent able to deal with a vast number of possible games that can be described in GDL. There is virtually no way of incorporating any domain-specific (game) knowledge directly into the program and the evaluation function must be somehow autonomously generated, probably by an AI- or CI-based routine.

Nevertheless, some researchers attempted (with some success) to identify (within the game rules) concepts typical for many human-played games, such as boards and pieces [14], [24], [11], [18], [17]. While there is no GGP-inherent reason for this approach to work, many of the GGP games are naturally based on real-world human games and, therefore, these concepts are actually fairly abundant in many of them.

In our application, though, we decided to avoid any such preconceptions and employ a totally knowledge-free evaluation function generation approach, based solely on simulations and statistical analysis of a game description. Our evaluation function is represented as a linear combination of *features*. Features are game-specific numerical values describing game positions. Typical feature examples would be checker or king counts in the game of Checkers or the presence of a particular piece on a particular square in Chess. While a set of features is not expected to uniquely and fully describe a game state, it is expected to provide enough information to judge the most likely outcome of the game with reasonable accuracy.

In order to build our evaluation function, we needed, therefore, two mechanisms: one for generating the set of features to be incorporated into our function and another one for assigning weights to them. The following subsections describe the process in detail.

1) Features Identification: The first step of our algorithm consisted in generation of a (possibly huge) set of features that could be used for building evaluation function for a particular game at hand. While designing this process, we were inspired by the general ideas and concepts presented in some of the earlier papers dealing with GGP, [14], [24] and [2].

Especially [2] is the source of many of the concepts described below. It incorporates the ideas of extracting expressions from the game description and identifying function argument domains to generate more specialized features. It also introduces the concept of stability, albeit not calculated according to the same formula we use. On the other hand, the GGP agent presented in [2] does not employ a feature generalization phase or build compound features in a way present in our solution. Additionally, we do not attempt to impose any predefined interpretations to the expressions or build a simplified model of the game. Instead, we employ a custom algorithm to build an evaluation function directly from the generated expressions.

Features used by our agent are represented by expressions similar to those of a GDL description of the analyzed game. For instance, the expression $(cell\ ?x\ ?y\ b)$ would represent the number of empty fields in the current position in a game of Tic-Tac-Toe. In general, the value of a feature (in a particular game state) is calculated by identifying the number of possible combinations of values for variables of the expression, so that the resulting expression is true in a given state. In the case of features with constants only (and no variables) only two values would be possible: 0 and 1.

The initial set of features is extracted directly from the game rules. This set is expected to already be reasonable as the game description contains mainly expressions that have direct significance either for further development of the game, or for its score. In both cases, they make good candidates for evaluation function components.

Still, we do not settle for this relatively small number of candidate features and try to identify more of them. For this purpose we employ two procedures, characteristic for human cognitive processes: *generalization* and *specialization*. We believe their combination should lead us to a better understanding of the game and finding features possibly crucial for a success.

We start with generalization, which is a relatively straightforward process. Namely, in each of the expressions identified so far, we replace constants with variables, generating all possible combinations of constants and variables. For instance, for an expression of $(cell\ 1\ ?\ b)$, we would generate 3 new candidate features: $(cell\ ?\ ?\ b)$, $(cell\ 1\ ?\ ?)$ and $(cell\ ?\ ?\ ?)$. Obviously, we may arrive at a feature that is already in our population - it is then ignored as we do not need any repetitions.

The aim of the specialization phase is, as the name suggests, opposite to the generalization phase. We try to generate features that contain fewer variables than those found so far. For the feature $(cell\ 1\ 1\ ?)$ we would ideally like to generate 3 new expressions $(cell\ 1\ 1\ x)$, $(cell\ 1\ 1\ o)$ and $(cell\ 1\ 1\ b)$. This phase, however, requires a slightly more sophisticated algorithm than generalization. Theoretically, we could simply attempt to replace variables in all the features with all known constants, but it would lead to an explosion in the number of features that would be impossible to handle.

Therefore, we attempt to at least approximately identify the domains of all parameters of all predicates. In order to achieve that, we employ an algorithm first described in [24]. This method is not guaranteed to generate precise results but rather supersets of the actual domains. Still, it is capable of severely limiting the number of generated features. Basically, it consists in analysis of all inferences and, within each of them, identifying variables present in several predicates (or as several arguments of a single predicate). Whenever such a case is discovered, it is assumed that the parameters in question have identical domains. A simultaneous analysis of all constant occurrences allows a quick assignment of constants to all the shared domains found by the algorithm.

Still, in some cases it is possible for the domain sizes to be prohibitively big, possibly leading to an explosion in the candidate features count. Therefore, whenever a parameter's domain size exceeds a configured limit, it is ignored and takes no part in the specialization process. In all our experiments this threshold was set to 30, which led to up to several thousand candidate features in moderately sophisticated games (4343 in Checkers, being the greatest value in this category) and several dozen thousand in the extreme cases (such as Chess with 28163 candidate features).

Finally, after the specialization phase is completed, even more features are generated by the process of combining existing features. The only type of compound feature currently in use in our application is the difference of two similar features, where two simple features are considered similar if and only if they are based on the same predicate and its arguments differ on one position only which in both instances contains a constant. An example of a pair of such similar features would be $(cell\ ?\ ?\ x)$ and $(cell\ ?\ ?\ o)$. A compound feature built of those two simple components would, therefore, calculate the difference in the count of x and o markers on the board.

2) Features Analysis: Since building evaluation function with the use of several thousand features (many of them redundant or unimportant for the final score) is not feasible, the next step of our algorithm involves some statistical analysis of the features in order to allow selecting the most promising ones and assigning weights to them.

The assessment is made by means of random simulations. First, we generate a set of some number (typically 100) of random short game-state sequences that will form the basis for further analysis. Each sequence starts in a position generated by randomly making a number of moves from the initial position of the game. This initial depth is selected randomly for each sequence from an interval depending on the expected length of a game - estimated by random simulations as well.

Each random-play-generated sequence consists of 2 to 5 positions, either consecutive or almost consecutive (with a distance of 2 or 3 plies). Once the sequences are generated, up to 10 Monte Carlo rollouts are performed from the last position of each of them. These allow an estimation of the final scores of all the players for each sequence.

The data obtained so far serves as the basis for calculating some typical statistics for all the features. These include their average values, average absolute values, variance, minimum and maximum values, etc. While these figures are useful for some minor optimizations of the algorithm, two statistics are crucial for the overall effectiveness of the features selection process: *correlation* of the feature's value with the scores of each player and its *stability*. While the former is obvious (and calculated on the basis of random rollouts mentioned earlier), the latter requires some explanation.

Stability is intuitively defined as the ratio of the feature's variance across all game states to its average variance within a single game (or, in our case, game subsequence). More precisely, it is calculated according to the formula:

$$S = \frac{TV}{TV + 10SV}, \tag{2}$$

where TV denotes the total variance of the feature (across all game states, in our implementations: initial states of all generated sequences) and SV – the average variance within sequences. Continuing with the Tic-Tac-Toe game example we would expect the feature $(cell\ ?\ ?\ b)$ (the number of empty squares) to have a low stability value, since it changes with each and every ply. On the other hand, the feature $(cell\ 2\ 2\ x)$ (x on the center square) would be characterized by relatively high stability, as its value changes at most once per game.

We expect more stable features to be better candidates for the usage in heuristic evaluation of game states. Low stability due to high SV value would signal significant oscillations of the feature value between successive positions and, therefore, strong horizon effect (and, as a consequence, low prognostic quality). Relatively low TV, on the other hand, is typical for features that remain constant or change slowly and are not useful for distinguishing between even significantly different states. Although some false negatives are in general possible, we try to avoid using features with a low stability value.

3) Evaluation Function Generation: Once the set of candidate features is built and analyzed, it is time to define the evaluation function. As mentioned earlier, it is constructed as a linear combination of selected features. Two steps are required to create it: selection of the most promising features and assigning weights to them.

At the moment we employ a relatively simple strategy for realization of these tasks. Although simplistic, it turned out to be surprisingly robust and so far preliminary attempts at using more advanced CI-based approaches have not proven to be superior.

The procedure relies on the statistics described in the previous subsection, namely correlation with the expected score of the player (c) and stability of the feature (s). We order the features descending according to the minimum of absolute value of their score correlations and stability (i.e. $min(|c|, s)$) and choose the first 30 of them as components of the evaluation function. While those two values ($|c|$ and s) are not directly comparable, they both fall in the same range ($[0, 1]$) and although in most cases absolute values of score correlations are lower than stability, both of them actually influence the choice, and the formula cannot be simplified to using one of them only. The decision to have exactly 30 components in each evaluation function was based on a number of preliminary tests, which proved this size of the evaluation function to be quite robust across several test games.

The weights assigned to the components are defined to be equal to products of the score correlations and stabilities (i.e. $c \cdot s$). Note that, since stability value is always positive, the sign of the resulting weight will always be the same as that of the correlation.

Additionally, each evaluation function contains a special constant component equal to the average score of a given role in a game, whose value is calculated based on the simulations performed during the features analysis phase. Its effect is a shift in the expected value of the evaluation function. In the case of a 2-player competitive game, if all the other components evaluate to 0, thus they indicate no bias towards any of the players, the evaluation value will be equal to the average score expected for the player. Since the generated weights of the components are relatively low, the value of the evaluation function should rarely fall out of range of possible scores. Simple automatic scaling of the function would fix this problem, should the need arise, but it was not necessary in the case of any of the selected games.

4) Implementation: During the experiments described in this article we used an implementation without any time limit for the evaluator generation. Two out of the three main parts of the algorithm can easily be parallelized – especially, but not exclusively, in a shared-memory model employed by our solution. The time-consuming task of generating a pool of random game sequences can easily be divided into independent and parallel tasks of creating each of the sequences. Similarly, once this pool is prepared, statistic calculations for each of the features do not require almost any cooperation and can safely be parallelized. Finally, while it would be possible to at least partially parallelize the features' generation phase, we decided instead to run it in a single dedicated thread while other processors were utilized for a game sequences pool generation.

Overall, while obviously there is still place for optimization, we assess our solution's speed as more than satisfactory. On an Intel Core i7 machine employed in our experiments, the whole evaluation function generation phase would take from several seconds for a simple game (e.g. Connect4), through up to half a minute for a moderately sophisticated one (with Checkers being the slowest here with the results of 26-31s) and up to two minutes for Chess. In other words, we expect our approach could operate within tournament-level time limits even on a single machine. Further optimization and implementation in a distributed environment is also possible so as to reduce time requirements.

B. MTD(f) in GGP

In two-player turn-based adversarial games the most typical method of using a state evaluation function is employing it within the alpha-beta pruning algorithm or one of its modifications, or algorithms based on it (such as MTD(f) described in section III-B).

GGP framework, however, is not limited to 2-player games. While in this paper we restrict our experimentation to this category, we aim at designing a solution that would be applicable to any game that can be defined in GDL, including games with simultaneous moves and any finite number of players. While it is possible in such a case to perform a full fixed-depth game tree analysis, it would be far too slow to be practical.

Therefore, we decided to employ one of several possible variations of alpha-beta algorithm for multi-player simultaneous move games. Namely, we took a *paranoid* approach [25], which assumes that all other players form a coalition against us (trying to minimize our final score) and in each state decide on their actions after us (already knowing our not-yet-performed move). This way we effectively transform the game into a two-player one with twice the depth of the original one. While this transformation is clearly not optimal for all games, it is relatively simple and reasonable for most of the multi-player games encountered at GGP tournaments.

As mentioned earlier, with the game transformed as described above, we actually employ the time-constrained MTD(f) instead of plain alpha-beta pruning algorithm. While alpha-beta algorithm can be parallelized with, in some cases, significant gains in efficiency, at the moment we rely on its single-threaded implementation. Still, we find our solution speed to be acceptable. In entry and near-entry game positions we manage to perform a search with a depth ranging from 3-4 moves in Chess to 6 moves in Checkers and Connect4, within a 10s time limit.

C. Guided UCT

Arguably an important advantage of the UCT algorithm is that there is no need for an externally defined heuristic evaluation function. Still, in many cases (e.g. Go [7], Amazons [15], Lines of Action [29] and GGP itself [4], [26]) it seems that the UCT can benefit from the introduction of some sort of an additional, possibly game-specific, bias into the simulations. Below, we discuss several possibilities of combining the UCT with our automatically-inferred game-specific state evaluation function into what we call Guided UCT (GUCT) algorithm. While there are publications (e.g. aforementioned [29]) that propose alternative solutions, we concentrate here on universal, generally-effective approaches that seem especially well suited for the GGP research framework.

We believe the evaluation function can be useful both in the strict UCT phase (while traversing the graph of pre-generated and stored game states) and during the Monte-Carlo random rollouts. It is also worth noting that once a state evaluation function $F(s)$ is available, it is relatively easy to construct a state-action pair evaluation function $F(s,a)$. In our approach we define its value to be the average value of $F(s)$ across the set $D(s,a)$ of all states directly reachable from state s by performing action a (there may be more than one such state, because the state reached after performing a may also depend on actions performed by other players), i.e.:

$$F(s,a) = avg_{t \in D(s,a)}(F(t)) \tag{3}$$

1) Evaluation function as part of Q(s,a): Firstly, the heuristic evaluator can be employed by slightly modifying the state-action pair assessment function $Q(s,a)$ in (1), e.g. by defining it as a combination of $F(s,a)$ and the state-action value $Q(s,a)$ defined by the rules of plain UCT. One straightforward way to implement this idea, would be to preinitialize the action values whenever a new node is added to the in-memory UCT tree. All actions possible in such a new state would start with some preset value of $N(s,a)$ (proportional to the level of trust in the evaluation function) and $Q(s,a)$ values equal to $F(s,a)$. Afterwards, these values would be updated according to the standard UCT rules. This solution would make sure that some, potentially biased, assessment of the moves is available immediately and then gradually improved by averaging with, hopefully less biased, results of random rollouts. With time the influence of the heuristic evaluation function would gradually diminish.

2) Evaluation function for move ordering: In the case of less trust in the quality of the F function, another, less invasive approach could be used. In a typical UCT implementation, whenever a new node is added to the tree, its actions are tested in the simulations performed in random order. When time runs out, there will obviously be game states left with some actions not played even once. It would, hence, be advantageous to start testing from the most promising actions and the evaluation function could help achieve this goal by means of ordering the moves according to their expected quality. An analogous move ordering scheme, albeit useful for different reasons, is often employed in alpha-beta algorithm implementations, e.g. by means of history heuristic [22]. Incidentally, this last concept has also proven useful in the context of UCT algorithm and GGP agent [4].

3) Evaluation function in random rollouts: Alternatively, the heuristic evaluation function can also be weaved into the Monte-Carlo rollouts phase of the UCT algorithm. Following the ideas presented, in a slightly different context, in [4], the random selection of actions to perform can be replaced by one in which each move's selection probability depends in some way on its evaluation by the F function. The actual selection would usually be implemented by means of either Gibbs distribution or ϵ-greedy policy.

4) Evaluation function as rollouts cutoff: While we performed some initial experiments with the approaches presented above, so far we have achieved the best results with yet another method of introducing an evaluation function into the Monte-Carlo phase of the UCT routine. Namely, in each node encountered in the fully random simulations we introduce a relatively minor (0.1 in the current experiments) probability of stopping the rollout there and using the heuristic evaluation function's value as the rollout's return value. In effect, the longer the rollout, the higher the chance it will be stopped and replaced by the value of the evaluation function in the last node reached. The potential advantages of this approach are two-fold. Firstly, there is a high chance we will cut short any long and time-consuming simulation, resulting in time saving that will allow us to perform more simulations than would be otherwise possible. Secondly, the evaluation function will typically be applied to states at least several moves down the game tree from the last in-memory game tree position. It is reasonable to assume that the closer to the end of the game the state is, the easier it is to estimate the final score and, hence, the more reliable the evaluation function's value is.

As a final note, it should be stressed that the above-mentioned four strategies can be safely mixed with one another. Still, after some initial testing, we decided to use only the last one in the experiments described herein.

V. Experiments

In order to verify the quality of both the mechanism for automatic generation of the evaluation function and our search algorithm, we organized several tournaments, comparing our GGP agents (the UCT-based and the MTD(f)-based) with the one based on a plain UCT algorithm, as well as with each other.

At this stage of research we decided to limit the experiments to various 2-player games. We relied on the definitions available on the Dresden GGP Server [9], selecting 13 games based on real-world ones for the ease of intuitive assessment of their complexity level and identification of their pertinent features. After performing a set of simple experiments, we chose 5 of those for further analysis, each for a reason:

1) **Chess** as the most sophisticated of all the games considered;
2) **Checkers** as a moderately sophisticated game in which our agent is very successful;
3) **Connect4** and **Connect5** as games in which our agent achieves mixed results;
4) **Othello** as the game in which our agent fares worst of all the 13 games tested.

Apart from minor changes forced by limitations of GDL (such as inability to efficiently define a draw by threefold repetition rule in Chess), those games closely resemble their original counterparts, with the exception of the Connect5 which, contrary to its name suggesting some modification of Connect4, is simply a small-board variant of Gomoku.

In most of the tournaments described below (in sections V-A-V-C), more than 1000 matches were played for each of the five aforementioned games. The actual number of matches depended on the stability of results obtained. In general we would not stop an experiment until a standard error of mean of at most 6 percentage points (i.e. 3% of the range of possible values) was achieved for each of the tested time limits. In effect, the average standard error of the presented results does not exceed 5pp for any of the games and typically it stays significantly below 4pp. Since our test regime caused the experiments to be extremely time consuming (they spanned several months of usage of 3-5 machines) and we are not, at this point, interested in analysis of minute scores differences, we feel that the data accuracy is satisfactory for our needs.

A. Guided UCT vs. Plain UCT

In our first test we tried to assess the level of play of GUCT agent in relation to a plain UCT-based solution (with no preparation phase before the first move). We, therefore, had the two contesting agents repeatedly play against each other with different time limits per move (from 1 to 60 seconds). The GUCT player regenerated its evaluation function from scratch before each play. Sides were swapped after each game and the winner was awarded 1 point, while the loser -1 point. Ties resulted in 0 points for both players.

Figure 4 presents the results for GUCT agent in each game for each time limit. It is apparent that Guided UCT agent decisively loses to UCT in one game (Othello), slightly loses in another one (Connect5) and outperforms plain UCT in the

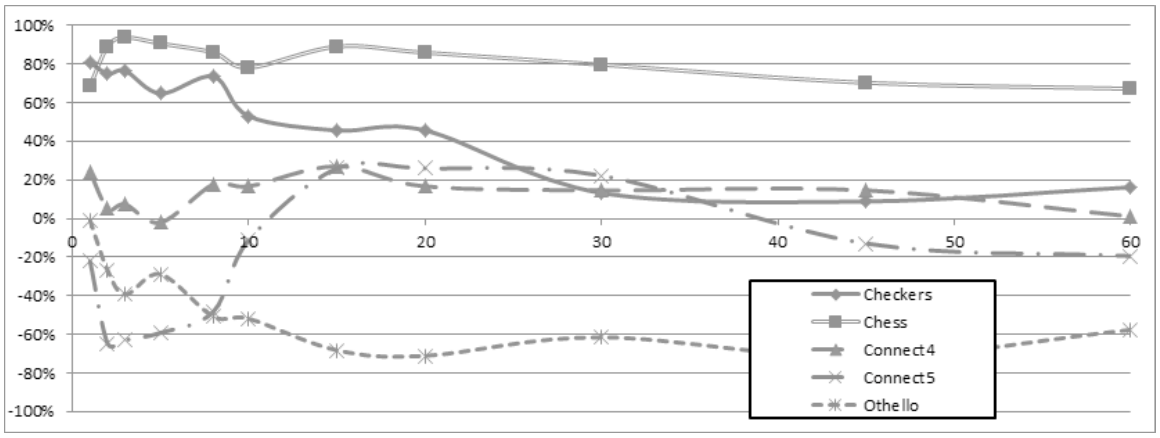

Fig. 4. GUCT vs. plain UCT tournament results (in percent). The X axis denotes time limits per move (in seconds).

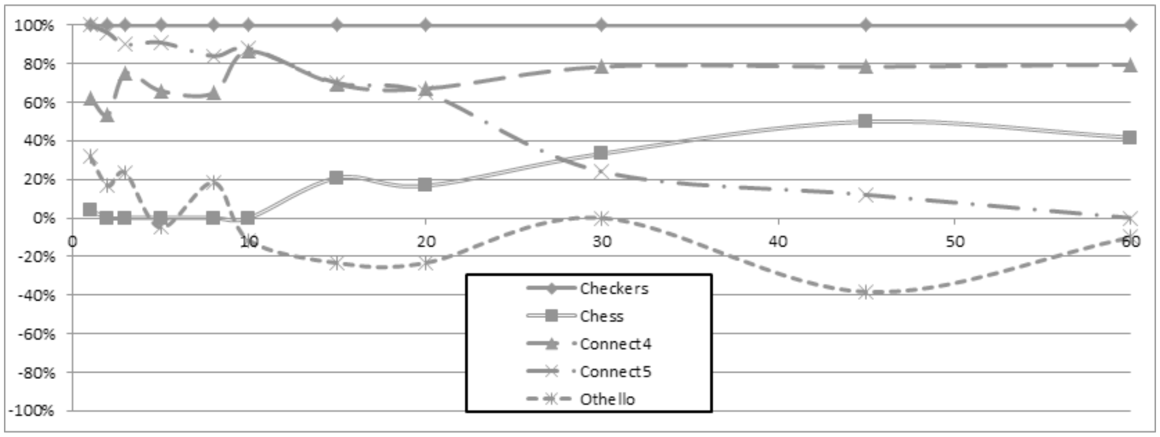

Fig. 5. MTD(f) vs. single-threaded plain UCT tournament results (in percent). The X axis denotes time limits per move (in seconds).

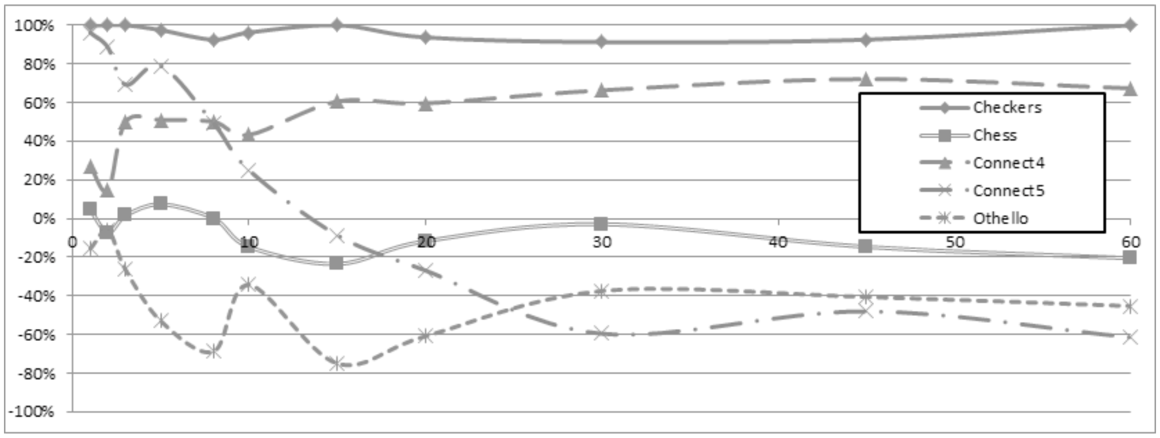

Fig. 6. MTD(f) vs. multi-threaded plain UCT tournament results (in percent). The X axis denotes time limits per move (in seconds).

remaining three – most notably in Chess, being the game with the highest branching factor among all tested and difficult to sample conclusively by the plain UCT algorithm. As far as Chess is concerned, one can also easily observe a relatively lower score for the time limit of 1 second per move. This stems from the fact that it is hard for any of the players to achieve a reasonable level of play with such a strict time regime and, therefore, more games actually end in draws.

The results for Checkers (another game for which an effective evaluation function must have been generated), besides showing a high level of play of the GUCT agent, also make an interesting dependency visible. It seems that with the increase in thinking time for the players, the advantage of GUCT over UCT diminishes.

In order to verify that this is not a statistical flux, we have performed a limited number of additional tests with higher time limits. The net result is that with higher time limits, both players begin to play more and more similarly. This effect is to be expected, since a greater number of simulations naturally improves the UCT method's efficiency and weakens the total influence of the evaluation function in GUCT. The same effect may not be so visible for other games (with the exception of Othello) because it can manifest itself at different move time limits and/or can be dampened by other effects.

In the case of Connect4, our agent still clearly outperforms plain UCT, however, with a much lower margin than in Chess or Checkers. At the same time, Connect5 turns out to be a somewhat more difficult game for GUCT. Additionally, in this case, a very interesting dependence of the scores values on time limits can be observed: GUCT agent's score is very low for short times and then raises significantly only to oscillate above and below the zero point depending on the time limit. At this point, we are not able to provide a full explanation of this behavior. We speculate that it is caused by an intricate interplay between the effects of random simulations and a less-than-ideal but useful evaluation function, both assessing similar states differently and urging the agent to employ different playing strategies. This may lead to confused decisions depending on the strength of influence of each of the two components.

Finally, after some inconclusive results with extremely low time limits (probably due to the plain UCT's inability to come up with any reasonable strategy with very low simulation counts), our GUCT agent decisively loses to plain UCT in Othello with higher thinking times per move. It is obvious that our method fails to properly identify the key positional factors in Othello and generate at least a reasonable positional strategy. Please remember, however, that Othello has been included in our experiment precisely because of how difficult it was for our agent.

B. Alpha-Beta Pruning vs. Plain UCT

In the second test, we analogically assessed the efficiency of our MTD(f)-based player by pitting it against the plain UCT agent. This time, however, another factor had to be considered. While our UCT implementation was multithreaded and capable of making good use of all 8 (virtual) cores of our test machine, MTD(f) was implemented in its basic single-threaded form. Therefore, we decided to actually perform 2 separate experiments. In the first one, the MTD(f) player would face a single-threaded version of the UCT agent. This comparison would be fair in that both players would have exactly the same processing power at their disposal. In the second one we would use a typical, multi-threaded implementation of UCT conceptually similar to the one presented in [3]. Considering the fact that UCT can be parallelized much more easily and possibly with much higher gains than alpha-beta pruning algorithms, we expected the first experiment to slightly favor the MTD(f) method and the second one – the UCT algorithm.

The results of the tournaments are presented in figures 5 and 6, which make several interesting observations possible. Firstly, for games with moderately sophisticated rules, limited branching factor and reasonable automatically generated evaluation functions (as proven by the previous experiment), MTD(f) seems to fare much better than GUCT did in the previous tests, even facing a multi-threaded implementation of UCT.

On the other hand, the MTD(f)-based agent fares surprisingly badly in Chess, actually losing many of the games to the multi-threaded UCT. It seems that a relatively high branching factor and high cost of generating successive game states are the two major factors here. Additionally, in figure 5 it can be observed that, with low time limits per move (up to 10 seconds), all Chess games repeatedly end in draws, again with none of the players being able to devise a winning strategy within such a limited time. It seems that while the inferred evaluation function is strong enough to significantly improve the playing level with reasonable-depth searches, it is not sufficient to guide the player on its own (with dramatically limited search time). Such an effect could be expected even with simpler, manually-defined Chess position evaluators. Our agent simply requires a play clock of more than 10 seconds to play chess-level game reasonably. Fortunately, one can safely expect that any game of such complexity will be played with a reasonably high time limit per move.

While Othello scores have improved compared to the first tournament, the problem with generating efficient evaluation function for this game is still visible.

As previously, the dependence between time limits and scores in Connect5 remains the most specific one. With a strict time regime the UCT level of play is clearly low enough for it to be easily defeated by an alpha-beta player even with a non-optimal evaluation function. As time limits grow, however, the UCT algorithm quickly gains more than MTD(f) (which still heavily relies on the evaluation function).

C. Alpha-Beta Pruning vs. Guided UCT

Finally, we decided to directly compare our two agents. The results, presented in figures 7 and 8, are in line with the previous experiments' results. Alpha-beta pruning is a clear winner in the case of Checkers and Connect4 - moderately complicated games with, apparently, successful evaluation functions. Chess, on the other hand, is clearly better suited

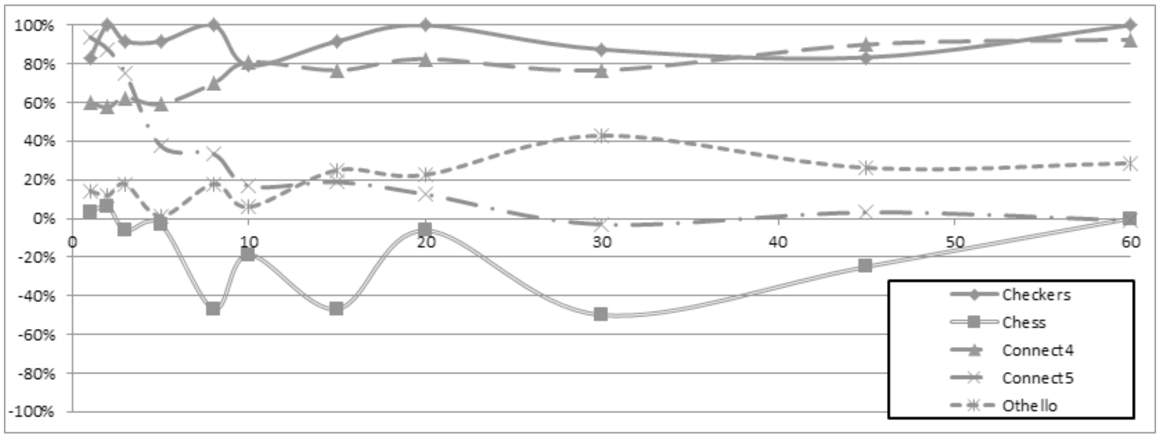

Fig. 7. MTD(f) vs. single-threaded GUCT tournament results (in percent). The X axis denotes time limits per move (in seconds).

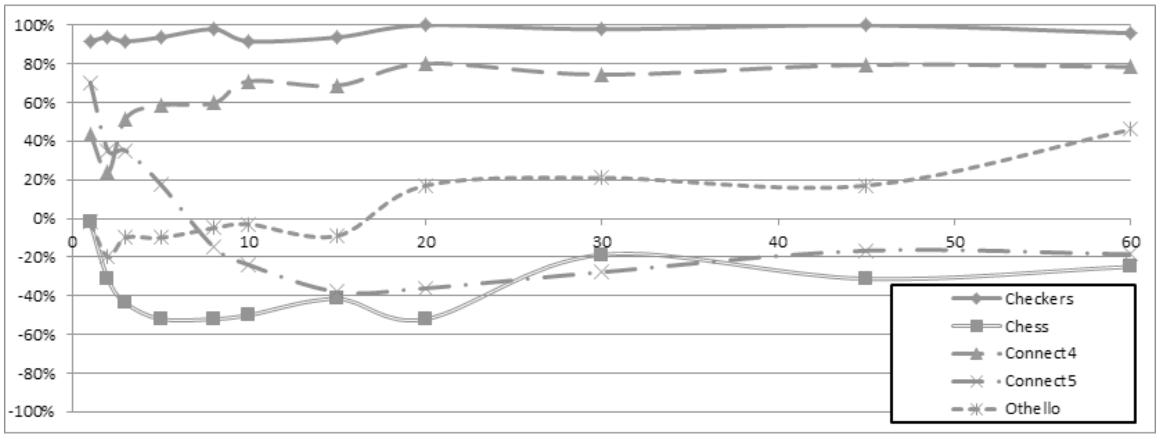

Fig. 8. MTD(f) vs. multi-threaded GUCT tournament results (in percent). The X axis denotes time limits per move (in seconds).

TABLE I
GUCT VS GUCT WITH NULL HEURISTIC TOURNAMENT RESULTS (WITH 95% CONFIDENCE BOUNDS)

	5s	10s	30s
Chess	44% (± 7%)	44% (± 8%)	54% (± 12%)
Checkers	85% (± 3%)	94% (± 4%)	85% (± 5%)
Othello	25% (± 9%)	14% (± 9%)	-11% (± 11%)
Connect4	56% (± 8%)	39% (± 12%)	36% (± 10%)
Connect5	25% (± 9%)	30% (± 9%)	43% (± 10%)

for the use of GUCT rather than MTD(f) (at least in the case of the time limits tested, and high cost of state transition due to GDL-based encoding of rules). In the case of Othello, a slightly better score of alpha-beta based player should not be treated as significant due to a generally low level of play. Connect5 scores, traditionally, depend significantly on the time limits. Strict time regime leads to easy victories by the MTD(f) method, while with higher time limits the chances are more even.

D. Guided UCT vs. Null Heuristic

In some games, it may be worthwhile to cut long and uninformative simulations short even without a reasonable evaluation function - by simply assuming that they end in a draw. This has been proven in the context of GGP in [5]. While the aforementioned approach included initial analysis to intelligently decide which rollouts to stop early, it might be possible that a similar effect could also be achieved for some games by our probabilistic cut-off algorithm. If this effect was dominant, the actual value of the evaluation function would not matter as long as it remained close to a draw value.

In order to verify if this is the case we compared our evaluation functions with a null one which returns a draw for every game state. The results of such a comparison for several typical time limits are presented in table I. Each value is based on at least 50 matches and the rules were identical to those in other tournaments: sides were swapped after each game, 1 point was awarded for a victory, -1 for a loss and 0 for a draw. To compensate for differences in match counts, results in the table are presented as percentages (with 100% representing GUCT victory in every game, and -100% its loss in all matches). Based on these values, it can safely be stated that, with a single possible exception of the Othello evaluator, our evaluation functions are actually useful and the results obtained in the previous experiments are not the effects of early cut-offs only.

E. Results summary

We think that the results achieved so far are promising, doubly so considering the simplified way of choosing the evaluation function components and assigning weights to them. Our method is obviously not equally successful in all games, but that is only to be expected considering the broad range of games that can be defined in GDL.

There are games, such as Othello, for which the usage of an automatically generated evaluation function may in some cases actually prove detrimental, but, please, bear in mind that Othello was selected for our experiments specifically because of this fact. Additionally, it is possible that it is not the game itself that is problematic for our solution but the specific way in which it was defined (the same game can be defined by different GDL descriptions).

The quality of the proposed players depends heavily on the amount of time available for each move. In the experiments we performed tests with a range of limit values typical for GGP tournaments held so far, thus proving the feasibility of employing the proposed solution in a tournament environment.

The relative quality of the two game tree search algorithms utilized in our application seems, additionally, to depend heavily on the specifics of the game played and, to some extent, the quality of the evaluation function. The alpha-beta pruning based agent is generally better suited for games of moderate complexity (and, hence, a relatively lower cost of state expansions) and higher quality, more reliable evaluation functions. GUCT algorithm has some potential to deal with more biased evaluation functions and definitely copes better with sophisticated games with higher branching factors and more expensive rules processing.

Evaluation functions differ significantly from match to match, since they are regenerated each time in a process based on a randomly selected set of sample game positions. Additionally, the same value can often be calculated in multiple ways, e.g. the piece count difference can be represented by a single compound feature, separate features for each player with opposite weights or a number of features calculating the piece count for each row of the chessboard.

F. Sample evaluation functions

Sample evaluation functions for Checkers and Othello are presented in the appendices. On a general note, we observe that the majority of the components of the evaluation function have proper (i.e. consistent with human intuition) signs, which means that advantageous features contribute positively whereas disadvantageous ones have a negative contribution to the overall position assessment.

In the case of Checkers, please notice that in the notation used in this game definition, h1 is a playable position while a1 is not. It is clearly visible (and was discernible across all generated instances of the evaluation function) that the evaluators, first of all, contain a general material-based features: their first two components calculate the differences in pieces and kings counts of the players. This rule would hold true for almost all evaluation function instances (in more than 80% of the analyzed cases). In rare exceptions these components would typically be replaced by a functionally equivalent notation, e.g. counting pieces and kings separately for each player, summing pieces counts separately for each row, etc.

Further components form at least some attempt of positional assessment. For instance, the evaluators (especially the first one) apparently stress control of the h column, probably because pieces in these positions cannot be captured. At the same time, there is also a lot of noise in the formula. For instance, some of the compound expressions contain references to cells a5 and h2 which will always remain empty.

Building an evaluation function for Othello proved to be a much more difficult task. These functions contain quite a lot of noise and apparently random components. Still, in most of them at least some reasonable components are identifiable. Simple piece counts are often present but are known to be not very useful as part of an evaluator. The sample evaluation function presented herein clearly shows that our algorithm managed, at least to some extent, to identify the importance of board borders and corners - unfortunately, in this case only for one side of the board.

The last component of each function represents the bias (autonomously developed by the method) whose role is to shift the average expected value of the evaluation function to the average expected value of the game.

VI. Conclusion and future research

We have described in detail our attempt at creating a General Game Playing agent relying on an automatically inferred heuristic state evaluation function. We also offered two methods of employing this function in the context of GGP based on two popular game tree search algorithms.

As stated earlier, we assess this attempt as successful, especially considering that, at the moment, only very simple statistical methods for building evaluation function from components generated by human-cognition-inspired processes of *generalization* and *specialization* of the game rules are employed. Further improvements in this part of our application are one of our current research goals. One of the solutions we are considering at the moment is to employ CI techniques for this task, in particular the Layered Learning method [19], [27]. We also plan to further tweak and improve the evaluation function during the game (i.e. in the playing phase of the match).

Additionally, our experiments show that the relative efficacy of the game-tree analysis algorithms tested (MTD(f) and GUCT) depends heavily on the characteristics of the game played. Devising a method for the automatic selection of one of the two approaches before the game begins (or perhaps even switching them while playing) is therefore another interesting research topic.

Finally, further analysis and a possible identification of the class of games for which our approach is not successful would probably allow methods to be designed for circumventing such problems and gaining a more even level of play across all games. Specifically, it can be clearly seen in the results presented that Othello proved to be one such game and its deeper analysis should provide important insight into this issue.

Appendix A

Sample automatically inferred evaluation functions for, respectively, white and black Checkers players. *wp, wk, bp, bk* denote white pawn, black pawn, white king, black king, respectively. *cell(x,y,z)* denotes the fact that a piece *z* occupies square (x, y). General interpretation of components used by evaluation functions can be found in section IV.

0,33*(cell(?, ?, wp)-cell(?, ?, bp)) + 0,11*(cell(?, ?, wk)-cell(?, ?, bk)) + -0,12*(cell(h, 3, bp)-cell(h, 5, bp)) + 0,12*(cell(h, ?, wp)-cell(h, ?, bk)) + 0,14*(cell(h, 3, wp)-cell(h, 3, bp)) + 0,12*cell(h, ?, wp) + 0,11*(cell(h, ?, wp)-cell(h, ?, bp)) + -0,065*cell(h, 5, b) + 0,065*(cell(a, 5, b)-cell(h, 5, b)) + 0,065*(cell(c, 5, b)-cell(h, 5, b)) + 0,065*(cell(e, 5, b)-cell(h, 5, b)) + 0,065*(cell(g, 5, b)-cell(h, 5, b)) + 0,065*(cell(h, 4, b)-cell(h, 5, b)) + -0,065*(cell(h, 5, b)-cell(h, 2, b)) + -0,065*(cell(h, 5, b)-cell(h, 6, b)) + -0,065*(cell(h, 5, b)-cell(h, 8, b)) + 0,1*(cell(?, 3, wp)-cell(?, 3, bp)) + 0,061*(cell(h, 3, b)-cell(h, 5, b)) + -0,053*(cell(h, 5, b)-cell(h, 5, bk)) + -0,13*cell(h, 3, bp) + 0,13*(cell(a, 3, bp)-cell(h, 3, bp)) + 0,13*(cell(c, 3, bp)-cell(h, 3, bp)) + 0,13*(cell(e, 3, bp)-cell(h, 3, bp)) + 0,13*(cell(g, 3, bp)-cell(h, 3, bp)) + 0,13*(cell(h, 1, bp)-cell(h, 3, bp)) + -0,13*(cell(h, 3, bp)-cell(h, 3, wk)) + -0,13*(cell(h, 3, bp)-cell(h, 4, bp)) + -0,13*(cell(h, 3, bp)-cell(h, 2, bp)) + -0,13*(cell(h, 3, bp)-cell(h, 6, bp)) + -0,13*(cell(h, 3, bp)-cell(h, 8, bp)) + 48*1

-0,23*(cell(?, ?, wp)-cell(?, ?, bp)) + -0,12*(cell(?, ?, wk)-cell(?, ?, bk)) + -0,12*(cell(h, 5, bp)-cell(h, 7, bp)) + 0,13*(cell(b, 5, bp)-cell(h, 5, bp)) + 0,14*(cell(h, 3, bp)-cell(h, 5, bp)) + 0,067*(cell(c, 2, wp)-cell(c, 2, bk)) + 0,084*cell(c, 2, wp) + -0,084*(cell(b, 2, wp)-cell(c, 2, wp)) + -0,084*(cell(c, 1, wp)-cell(c, 2, wp)) + -0,084*(cell(c, 3, wp)-cell(c, 2, wp)) + -0,084*(cell(c, 5, wp)-cell(c, 2, wp)) + -0,084*(cell(c, 7, wp)-cell(c, 2, wp)) + 0,084*(cell(c, 2, wp)-cell(c, 2, wk)) + 0,084*(cell(c, 2, wp)-cell(c, 8, wp)) + 0,084*(cell(c, 2, wp)-cell(d, 2, wp)) + 0,084*(cell(c, 2, wp)-cell(f, 2, wp)) + 0,084*(cell(c, 2, wp)-cell(h, 2, wp)) + -0,16*(cell(a, 6, wp)-cell(a, 6, bp)) + -0,079*(cell(a, ?, wp)-cell(a, ?, bp)) + -0,094*(cell(h, 7, wp)-cell(h, 7, bp)) + -0,097*(cell(?, 6, wp)-cell(?, 6, bp)) + -0,06*(cell(f, 1, b)-cell(f, 1, bk)) + -0,072*(cell(a, 2, wp)-cell(c, 2, wp)) + 0,13*(cell(a, 6, bp)-cell(a, 6, wk)) + 0,074*(cell(h, 5, b)-cell(h, 7, b)) + -0,057*(cell(g, ?, wp)-cell(g, ?, bp)) + 0,073*(cell(g, ?, bp)-cell(g, ?, wk)) + -0,052*(cell(a, ?, wk)-cell(h, ?, wk)) + 0,053*(cell(d, ?, bk)-cell(g, ?, bk)) + 0,08*(cell(b, ?, bp)-cell(b, ?, wk)) + 59*1

Appendix B

Sample automatically inferred evaluation function for black Othello player. *cell(x,y,z)* denotes the fact that a *z*-colored piece occupies square (x, y).

-0,19*cell(?, ?, red) + -0,22*cell(8, ?, red) + -0,22*cell(?, ?, ?) + -0,08*cell(8, 8, red) + -0,076*cell(?, 8, red) + -0,19*cell(8, ?, ?) + -0,17*cell(?, 8, ?) + -0,067*cell(8, 8, ?) + 0,14*(cell(8, ?, black)-cell(8, ?, red)) + -0,052*cell(4, ?, red) + 0,067*(cell(?, ?, black)-cell(?, ?, red)) + -0,095*cell(8, 4, red) + -0,042*cell(5, ?, red) + -0,099*cell(8, 5, red) + -0,11*cell(4, ?, ?) + -0,14*cell(?, 4, ?) + -0,052*cell(?, 4, red) + -0,13*cell(?, 5, ?) + -0,039*(cell(5, 8, black)-cell(5, 4, black)) + -0,097*(cell(8, ?, ?)-cell(5, ?, ?)) + -0,047*(cell(8, ?, red)-cell(5, ?, red)) + -0,087*(cell(5, 8, ?)-cell(5, 4, ?)) + -0,087*(cell(5, 8, ?)-cell(5, 5, ?)) + -0,087*cell(5, 8, ?) + -0,035*(cell(8, 5, red)-cell(5, 5, red)) + -0,11*cell(5, ?, ?) + -0,085*(cell(8, ?, ?)-cell(4, ?, ?)) + -0,036*cell(?, 5, red) + 0,028*(cell(?, 8, black)-cell(?, 8, red)) + -0,072*(cell(?, 8, ?)-cell(?, 5, ?)) + 49*1

Acknowledgment

The research was financed by the National Science Centre in Poland, based on the decision DEC-2012/07/B/ST6/01527.

References

[1] P. Auer, N. Cesa-Bianchi, P. Fischer: Finite-time analysis of the multi-armed bandit problem. Machine Learning 47(2/3), pages 235–256, 2002
[2] J. Clune: Heuristic evaluation functions for General Game Playing. In Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI- 07), pages 1134–1139, Vancouver, BC, Canada, 2007. AAAI Press.
[3] M. Enzenberger, M. Müller: A lock-free multithreaded Monte-Carlo tree search algorithm. Lecture Notes in Computer Science (LNCS), vol. 6048, pages 14–20 (12th International Conference on Advances in Computer Games), Springer-Verlag, 2010
[4] H. Finnsson, Y. Björnsson: Simulation-based approach to General Game Playing. In Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-08), pages 259–264, Chicago, IL, 2008. AAAI Press.
[5] H. Finnsson: Generalized Monte-Carlo Tree Search Extensions for General Game Playing. In Twenty-Sixth AAAI Conference on Artificial Intelligence. 2012.
[6] S. Gelly, Y. Wang: Exploration exploitation in Go: UCT for Monte-Carlo Go. In Neural Information Processing Systems 2006 Workshop on On-line trading of exploration and exploitation, 2006
[7] S. Gelly, Y. Wang, R. Munos, O. Teytaud: Modification of UCT with patterns on Monte Carlo Go. Technical Report 6062, INRIA, 2006
[8] General Game Playing website by Stanford University. http://games.stanford.edu/.
[9] General Game Playing website by Dresden University of Technology. http://www.general-game-playing.de/.
[10] M. Genesereth, N. Love: General Game Playing: Overview of the AAAI Competition. http://games.stanford.edu/competition/misc/aaai.pdf, 2005.
[11] D. Kaiser: The Design and Implementation of a Successful General Game Playing Agent. In Proceedings of FLAIRS Conference, pages 110–115, 2007
[12] D. E. Knuth, R. W. Moore: An analysis of alpha-beta algorithm, Artificial Intelligence vol. 6, issue 4, 293–326, 1975
[13] L. Kocsis, C. Szepesvári: Bandit Based Monte-Carlo Planning. In Proceedings of the 17th European conference on Machine Learning (ECML'06), 282–293, Berlin, Heidelberg, Springer-Verlag, 2006
[14] G. Kuhlmann, K. Dresner, and P. Stone: Automatic heuristic construction in a complete General Game Player. In Proceedings of the Twenty-First AAAI Conference on Artificial Intelligence (AAAI-06), pages 1457–1462, Boston, MA, 2006. AAAI Press.
[15] R. Lorentz, Amazons discover Monte-Carlo, in Computers and Games (CG 2008), ser. Lecture Notes in Computer Science (LNCS), H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands, Eds., vol. 5131, 2008, pages 13–24.
[16] N. Love, T. Hinrichs, D. Haley, E. Schkufza, M. Genesereth: General Game Playing: Game Description Language Specification. http://games.stanford.edu/language/spec/gdl_spec_2008_03.pdf, 2008.
[17] J. Mańdziuk, *Knowledge-Free and Learning-Based Methods in Intelligent Game Playing*, ser. Studies in Computational Intelligence. Berlin, Heidelberg: Springer-Verlag, 2010, vol. 276.
[18] J. Mańdziuk, M. Świechowski: Generic Heuristic Approach to General Game Playing. Lecture Notes in Computer Science, vol. 7147 (SOFSEM'12), pages 649–660, 2012, Springer-Verlag.
[19] J. Mańdziuk, M. Kusiak, K. Walędzik: Evolutionary-based heuristic generators for checkers and give-away checkers. Expert Systems, vol. 24(4): 189–211, Blackwell-Publishing, 2007.

[20] J. Méhat, T. Cazenave: Combining UCT and Nested Monte Carlo Search for Single-Player General Game Playing, IEEE Transactions on Computational Intelligence and AI in Games, Vol. 2(4), 271–277, 2010.
[21] A. Plaat, J. Schaeffer, W. Pijls, A. de Bruin: Best-First Fixed-Depth Minimax Algorithms, Artificial Intelligence, vol. 87(1–2), 255–293, 1996.
[22] J. Schaeffer, The history heuristic, International Computer Chess Association Journal, vol. 6(3), 16–19, 1983.
[23] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, vol. 317(5844), 1518–1522, 2007.
[24] S. Schiffel, M.Thielscher: Automatic Construction of a Heuristic Search Function for General Game Playing. In Seventh IJCAI International Workshop on Non-monotonic Reasoning, Action and Change (NRAC07).
[25] N. Sturtevant, R. E. Korf: On Pruning Techniques for Multi-Player Games. In Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence, 201–207, AAAI Press, 2000.
[26] M. Świechowski, J. Mańdziuk: Self-Adaptation of Playing Strategies in General Game Playing. IEEE Transactions on Computational Intelligence and AI in Games, IEEE Press, (*in print*).
[27] K. Walędzik, J. Mańdziuk: The *Layered Learning* method and its application to generation of evaluation functions for the game of Checkers, Lecture Notes in Computer Science, vol. 6239 (PPSN XI, Part II), pages 543–552, 2010, Springer-Verlag.
[28] K. Walędzik, J. Mańdziuk: Multigame playing by means of UCT enhanced with automatically generated evaluation functions, Lecture Notes in Artificial Intelligence, vol. 6830 (AGI'11), pages 327–332, 2011, Springer-Verlag.
[29] M.H.M. Winands, Y. Björnsson, and J-T. Saito (2010). Monte Carlo Tree Search in Lines of Action. IEEE Transactions on Computational Intelligence and AI in Games, vol. 2, no. 4, pages 239–250.
[30] YAP Prolog. http://www.dcc.fc.up.pt/ vsc/Yap/.