Testing General Game Players Against a Simplified Boardgames Player Using Temporal-difference Learning

Jakub Kowalski
Institute of Computer Science
University of Wrocław, Poland
Email: kot@ii.uni.wroc.pl

Andrzej Kisielewicz
Institute of Mathematics
University of Wrocław, Poland
Email: Andrzej.Kisielewicz@math.uni.wroc.pl

***Abstract*—In this paper we assess the progress of General Game Playing by comparing some state-of-the-art GGP players with an exemplary program dedicated to playing games in a smaller class called Simplified Boardgames. Conclusions on further possible development are made. The paper is also the first step in creating a standard test class for measuring performance of GGP players.**

I. Introduction

The aim of *General Game Playing* (GGP) is to develop a system that can play variety of games with previously unknown rules. Unlike standard artificial game playing, where designing an agent requires special knowledge about the game, in GGP the key is to create a universal algorithm performing well in different situations and environments. As such, General Game Playing was identified as a new Grand Challenge of Artificial Intelligence and from 2005 the annual International General Game Playing Competition (IGGPC) is taking place to foster and monitor progress in this research area [1].

Games in GGP are described in a language called *Game Description Language* (GDL) [2]. It has enough power to describe all turn-based, finite and deterministic n-player games with full information. Playing a game given by such a description requires not only developing a move searching algorithm, but also implementing a reasoning approach to understand the game rules in the sense of computing legal moves, the state update function, and the goal function.

Through a decade of developing general problem-solving approaches, a great advancement has been achieved. New approaches, algorithms and data structures have been used to improve players' efficiency. Despite that fact, still GGP players are not a match for the specialized programs designed to master one game only. This, however, shows a lack of some "in-between" problem which can be used as comparison, being on the edge of GGP programs capabilities. This should be a mid-range class of games, much tighter than that described by GDL, but wide enough to be challenging on its own.

Such general classes of games, simpler than GDL, were created as first approaches to the general game playing idea by Pitrat in 1968 [3], and later by Pell in 1992 [4]. Both of these classes were describing chess-like boardgames, which ensures that some of standard approaches used in chess and checkers engines can be reused for creating efficient playing programs. For example Pell's Metagamer [5], used alpha-beta minmax search with a predefined set of evaluation functions.

A larger, but simpler to describe, class called *Simplified Boardgames* was proposed recently by Björnsson [6]. This class contains two player, rectangular board games, where piece moves are described by regular expressions, independent of piece position and move history. Simplified Boardgames research focused mainly on the learning of legal moves based on the observation of matches.

In this paper we are using Simplified Boardgames as a comparison class for GGP players. We designed a program that can play previously unknown boardgames in a manner defined by the GGP protocol. Our player uses the alpha-beta minmax algorithm and evaluates states by material and position analysis using piece tables and piece-square tables. Proper weight assignment for the evaluation function, which was stated as an open problem for Metagamer player ([5]) is solved here by applying temporal-difference learning [7]. This was one of the ideas mentioned by Pell, and it was already applied for the material analysis in some non standard chess variants [8], as well as for standard chess [9]. We provided several experiments opposing our simplified boardgames player against some of the top GGP players, and comparing efficiency for different styles of learning.

The paper is organized as follows. Sections II, III and IV, provides necessary background for General Game Playing, simplified boardgames and reinforcement learning respectively. Detailed setup of our experiments is presented in section V. Section VI contains overview of the results. We conclude and give perspective of future research in Section VIII.

II. General Game Playing

General Game Playing, in its modern form introduced in 2005 at Stanford University, focused on developing programs which can play, without human intervention, in any game described by Game Description Language. We assume that the reader is familiar with basics of this domain of AI. For a detailed, up-to-date survey of the discipline we recommend [10].

A. Game Description Language

Game Description Language [1], [2], is a strictly declarative language using logic programming-like syntax based on

the Knowledge Interchange Format. Every game description contains declarations of player roles, initial game state, legal moves, state transition function, terminating conditions and the goal function.

GDL does not provide any predefined functions: neither arithmetic expressions nor game-domain specific structures like board or card deck. Every function and declaration must be defined explicitly from scratch, and the only keywords used to define game are presented in Table I.

TABLE I. GDL KEYWORDS

`(role ?r)`	`?r` is a player
`(init ?f)`	fact `?f` is true in the initial state
`(true ?f)`	fact `?f` is true in the current state
`(legal ?r ?a)`	in the current state `?r` can perform action `?a`
`(does ?r ?a)`	`?r` performed action `?a` in the previous state
`(next ?f)`	`?f` will be true in the next state
`terminal`	current state is terminal
`(goal ?r ?n)`	player `?r` score is `?n`

The execution model works as follows. Starting from the initial state, in every state s every player r selects one legal action. Then, the joint move of all players is applied to the state update function to obtain a new state s'. If s' is terminal, then every player has given goal score and the game ends.

To be considered as *valid* a GDL game specification must be *stratified* and *allowed*. This, and other syntactic restrictions ensures that the game have a unique standard model with only a finite number of true positive instances, so all deductions about the state and its successors are finite and decidable. For details we must refer to [2].

B. GGP Framework

Every GGP match is supervised by an application called *game controller*, which is responsible for supervising the course of the game including guarding timelimits and communicating between players using HTTP messages sent via TCP protocol. The detailed specification of the communication protocol can be found in [1].

A match begins with a message sending to players game rules in GDL, their roles, and two timelimits: *startclock* and *playclock*. The players have time to process the given rules and after *startclock* seconds send confirmation to the controller. Then, until a terminal state is reached, the players have to select their moves and send them to the controller within *playclock* seconds, while the controller sends back the played (joint) moves, necessary for players to calculate the next game state. If a player does not send legal move within the time limit, the controller will record a proper error and continue, substituting this player's move with random legal one.

C. Reasoning Engines

The *reasoning engine*, or simply *reasoner*, is an essential part of every GGP player allowing it to perform the state update. Given a state and moves of all players, the reasoner has to compute a new state including information about legal moves, terminality condition and player scores.

An efficient reasoner implementation is one of the most important parts of the general game player. Move searching algorithms, either simulation-based [11] or knowledge-based [12], require fast computation of parts of game trees, to improve their outcome. For this reason many of the players use distributed architectures to support parallel computations [13], combined with specialized solutions to speed-up GDL reasoning like compiling GDL into another language [14], [15], usage of propositional networks [16] or instantiating games to use binary decision diagrams [17].

III. SIMPLIFIED BOARDGAMES

The class of *Simplified Boardgames* introduced in [6] covers a set of games defined by the following rules:

- The game is played on a rectangular board consisting $n \times m$ squares. There are two players: black and white, each player controls an army of pieces, possibly of multiple types. Every square can be occupied by at most one piece. There is a fixed initial position.
- Players take actions in turns, with white starting. On its turn, a player moves one of its pieces from its current square to a different one, if such move is allowed by the game rules. If any piece, own or opponent's, happens to be on the destination square, it is removed from the board (*captured*).
- A *terminal position* arises when a piece of a certain type reaches a *goal* square, a current player has no legal moves, or there are less pieces of a certain type than some fixed constant (here we extend the simplified boardgames class; the latter condition was not the part of the original definition in [6]). The set of terminating rules is fixed, and can by different for each player.
- The game is a zero-sum game. When a player shifts the game into a terminal position, the winner is decided based on the fulfilled terminal conditions. If no winner is determined until a preset maximum game length is reached, the game ends in a *tie*.

A set of legal moves rules for each piece is the set of words described by a regular expression over an alphabet Σ containing triplets $(\Delta x, \Delta y, on)$ where Δx, Δy are relative column/row distances and $on \in \{e, p, w\}$ describes a content of a destination square: e indicates an empty square, p a square occupied by an opponent piece, and w a square occupied by an own piece.

Consider a rule $w \in \Sigma^*$, such that $w = a_1 a_2 \dots a_k$, each $a_i = (\Delta x_i, \Delta y_i, on_i)$, and suppose that a piece stands on a square $\langle x, y \rangle$. Then, the rule w is applicable if and only if, for every $i \leq k$, the content condition on_i is fulfilled by a content of the square $\langle x + \sum_{j=1}^{i} \Delta x_j, y + \sum_{j=1}^{i} \Delta y_j \rangle$. If a move rule w is applicable in a current game position, then the move transferring a piece from $\langle x, y \rangle$ to $\langle x + \sum_{i=1}^{k} \Delta x_k, y + \sum_{k=1}^{k} \Delta y_i \rangle$ is legal.

The language of piece's moves is impartial to the current absolute position of a piece. Rules where the same square is visited more then once are forbidden. An example of how move rules works is shown in Figure 1.

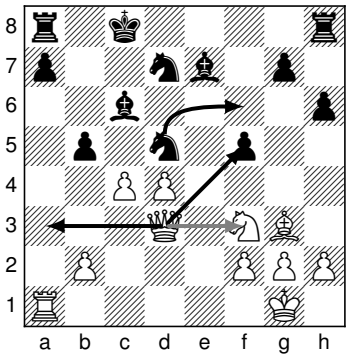

Fig. 1. A chess example. Two legal moves for the queen on $d4$ are shown. The capture to $f5$ is codified by a word $(1,1,e)(1,1,p)$, while move to $a3$ is encoded by $(-1,0,e)(-1,0,e)(-1,0,e)$. The move to $f3$ is illegal, as in the language of queen's moves, no move can end on a square containg own's piece. The $d5-f6$ knight move is a direct jump without necessity of checking content of in-between fields, so it is codified by a one-letter word $(2,1,e)$.

The main feature of the described class is simplicity in defining game rules, especially encoding moves as a regular language, excluding any "non regular" moves as promotion, castling, en passant, etc., (which were partially present e.g., in Pell's Symmetric chess-like games [18]). For that reason, along with the name *simplified boardgames*, we will use the term *regular boardgames*, especially when we want to emphasize extended terminal conditions.

IV. Reinforcement Learning in Boardgames

The key idea of *reinforcement learning* [7], [19] applied to games, is to learn the evaluation function by increasing the weights of the features which positively contributed to good choices of moves, and decreasing the weights of features which lead to bad moves. The question of which moves are good and which moves are bad is solved in a straightforward way by the assumption that all moves leading to a won game are good, while all moves leading to a lost game are bad. If a learning set is big enough, the statistics guarantees that favorable positions will occur more frequently in won games thus, the features which describes them will have higher weights.

A. Evaluation Function

The standard approach to evaluate a state of a game is to use some set of features $\mathcal{F}$ with a linear evaluation function of a form

$$E(p) = \sum_{f \in \mathcal{F}} w_f \cdot f(p), \tag{1}$$

where each individual feature f is a function assigning to each position p a real number, and w_f is a weight assigned to that feature. The task of learning the evaluation function for a given set of features is reduced to finding the weight values.

In our experiments we use two types of features which can be safely applied to all games in the simplified boardgames domain: material (piece values) and position (piece-square values). For every player, we create a *material feature* for each available type of piece, counting the number of pieces of that type for each side. Since in simplified boardgames the initial position does not have to be symmetric, we assume that the importance of the same piece for each player can differ, thus it should be evaluated separately. For both players, for each type of piece we create $n \cdot m$ *position features* (assuming the board size is $n \times m$). The feature value is 1 if the appropriate square is occupied by a piece of that type, and 0 otherwise.

To evaluate a given position from the perspective of the player who has to make a move, we sum weighted values of his pieces' features and subtract weighted values of the opponent's pieces' features. For example, if a white-to-move game position p contains two white rooks on squares (x_1, y_1) and (x_2, y_2), and a black rook on the square (x_3, y_3), then the position evaluation for white is:

$$E(p) = 2 \cdot w_{WR} - w_{BR} + w_{WR(x_1,y_1)} + w_{WR(x_2,y_2)} - w_{BR(x_3,y_3)},$$

where WR is a material feature containing the number of white rooks (analogously BR contains the number of black rooks), and $WR(x,y)$ is a position feature equal to one only if a white rook stands on the square (x,y) (analogously $BR(x,y)$).

B. Temporal-difference Learning

The reinforcement learning algorithm most frequently used in game playing is *temporal-difference learning* [20]. The most famous example of its successful usage is Tesauro's backgammon player which achieved a world champion level by learning only from self-play [21]. Faster convergence of TD(λ) algorithm is achieved by learning using differences between successive position estimations rather than the final score only. More precisely, for given feature f, its weight update at time t is computed by the following formula:

$$\Delta w_{t,f} = \alpha (P_{t+1} - P_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_{w_f} P_k, \tag{2}$$

where P_t is a position evaluation at time t, α is a global learning rate, $\nabla_{w_f} P_k$ is the partial derivative of P_k with respect to w_f, and λ is a parameter providing an exponentially decaying weight for more distant outcomes, minimizing their influence on the current position.

We want our predictions to reflect the probability of success, so we transform our raw evaluation E given by (1) into a sigmoid function

$$P(p) = \frac{1}{1 + e^{-\omega E(p)}}, \tag{3}$$

with parameter ω influencing the curve steepness. Then, the weight update $\nabla_{w_f} P_k$ for the feature f can be computed using the formula

$$\nabla_{w_f} P_k = f \cdot P_k \cdot (1 - P_k). \tag{4}$$

In our experiments we use the TDLeaf(λ) algorithm [9] which was designed to be effective on chess-like games where move

selection is based on deep searches. The key idea of this algorithm is to take the principal variation leaf node obtained from the search tree rooted in p rather then position p itself. We also use the Temporal Coherence algorithm [22] to dynamically adjust learning rate α for each feature separately.

V. Experimental setup

We have developed a program which can play any given, previously unknown, regular boardgame according to the General Game Playing framework. This allows us to compare our *Regular Boardgame Player* (*RBgPlayer*) with GGP players. We have chosen a few games and encoded them both in GDL and in regular boardgame format. Then we run matches using a standard GGP game controller. Players had some *startclock* time to prepare themselves to play and after that, they had to alternately perform moves, not exceeding *playclock* time. During one player turn the other one could perform some computations, but it had to sent the special `NOOP` move (which was the only legal move in GDL version) within the timelimit.

A. Games

Experiments have been performed using the following games, differentiated due to their size (depth and branching factor) and importance of material/position advantage:

Regular Chess is a standard game of chess cut to the regular boardgames framework. Thus there is no promotion nor check constraints; 50-move rules and draw by position repetition do not apply; special moves such as castling, en passant, two step pawn move are not allowed. To win the game, one has to capture the opponent's king or reach the opponent's backrank with a pawn. The game is considered as draw after 200 turns.

Regular Gardner (recently weakly solved [23]) is one of the minichess variants, played on a 5×5 board with a 100 turn limit. The starting position looks as in the regular chess with removed columns f, g, h, and rows 3, 4, 5. The same rules as in regular chess apply.

Regular Asymmetric Los Alamos is a slight modification of the standard Los Alamos 6×6 minichess variant. White pieces are the same as in the standard version (backrank from left: rook, knight, queen, king, knight, rook), but the black player has two bishops instead of two knights. Winning conditions and other rules are the same as for the Regular Gardner, with the exception that the turnlimit is set to 120.

Regular Escort Breakthrough is a modified version of the standard 8×8 breakthrough. Every player has one king on his backrank, and a row of pawns before it. A pawn may move one space straight or diagonally forward if the target square is empty, and only diagonally forward if it is occupied by enemy's piece. A king may move as the standard chess king, except that it cannot move backwards. To win, one has to reach opponent's backrank with the king. If no one wins within 60 turns, the game ends in a draw.

B. Search Engine

The basics of the search algorithm used in our RBgPlayer is negamax, which is a variant of standard alpha-beta that relies on the zero-sum property. We used the parallel iterative deepening algorithm complemented with a quiescence search and killer-move heuristic. We use non-persistent transposition tables, removing unattainable positions after captures.

During the RBgPlayer's turn, we start the iterative deepening search from depth 3, with parallelization occurring on the highest level only. When the time limit is reached, the embedded timer terminates searching and the result from the last fully computed depth is returned.

C. Learning Features Weights

The learning phase occurs during the opponent's turn and in preparation time before the match begins. We perform a series of constant depth playouts, without quiescence search, updating the weights after every playout. We learn weight updates separately for the white and black player, and we use the average as the final update.

We do not use non-determinism or randomness during the move searching phase to prevent game repetition, which is often an important issue during self-play learning. The main reason is that this would significantly slow down searching in situation where the time is crucial. Also, our learning function is self adapting during the game's course, depending on the opponent's choice of moves, which makes the self-play part safer for over-fitting errors.

If the time and game tree allow, we increase the playout depth (starting from 2), remembering times of longest playouts. This allows us to run more accurate simulations with assurance there will be enough of them (our algorithm tries to have at least 20 simulations per turn).

The real danger during the learning process, making playout repetition most probable, is too fast minimization of the learning value α, set by the Temporal Coherence algorithm. To prevent this, we update α values not after every simulation, but after every tenth simulation. According to our observations, this is enough to ensure that decreasing α is a result of a real weight convergence, not only coincidental cases.

D. Computing Initial Weights

Normally, computing proper weights takes a very long time. For example, according to experiments conducted in [8], piece values stabilized after 250-500 full matches, depending on the game. Because during a GGP match we do not have time for such an exhaustive learning process, we have developed a heuristic algorithm to compute initial weights based mainly on piece mobility and winning conditions.

As the first step, we identify how aggressive the game is, by counting ratio between displacement moves, captures and self captures; and then predict the content of the board during a middle game. The result of this prediction is given in a form of probability that a square will be empty, occupied by white or occupied by black. Then, for a given move rule we calculate its score, equal to the probability that this move will be valid. This is done by multiplying probabilities for all steps except the last one. If the final step is capture, we multiply the move score by 2, assuming that the possibility of capturing enemy pieces can always give advantage.

The initial weight for a position feature of a given piece is the sum of the scores of the moves that can be performed

by the piece from the given square, divided by the size of the board ($n \times m$). The material feature weight of a piece is just the sum of all its position features weights.

The weights computed so far, are now modified taking into account winning conditions. Let s be the constant describing the winning score in min-max nodes evaluations. If some player wins by capturing k pieces of a given type, then this piece's value is increased by $\sqrt[k]{s}$. This results in better protection of pieces which losing has more severe consequences. If some player wins by reaching a goal square by a piece of certain type, and he has j such pieces in the initial position, then the value of this piece is increased by $\sqrt[j]{s}$. In this case, we also modify the piece-square values. The values of goal squares are increased by s. The value of each square from which a goal square can be reached in $l \leq \frac{\max(n,m)}{2}$ moves, for each move, is increased by $\frac{\sqrt[l]{s}}{nm}$ multiplied by the probability of the move. This rewards advancing such pieces. However, because bonuses are not too large and added only in a limited board area, it prevents the blind march, which usually ends in capturing lonely pieces.

E. Reference GGP Players

We compare our min-max based RBgPlayer with two GGP programs, both using variants of the Monte Carlo Tree Search algorithm which, since its first success in IGGPC 2007, is the most popular algorithm for GGP players.

Dumalion is one of the top 8 players in 2014 International General Game Playing Competition, during which it won against previous year's champion TurboTurtle. Dumalion is based on a GDL compilation mechanism [24] and parallelization of the UCT algorithm over a cluster of computers. The main program is written in Java and running on a computer with Intel(R) Core(TM) 2 Quad CPU Q9300 2.50GHz 4 cores and 4GB of RAM. GDL reasoners are compiled into the C++ language and during this test run on 25 computers with Intel(R) Core(TM) i7-2600 CPU 3.40GHz 4 cores and 16GB of RAM.

Sancho by Steve Draper and Andrew Rose is the winner of the 2014 International General Game Playing Competition. The program uses the UCT algorithm and propositional networks [16] as a reasoning engine. It is written in Java, and operates on Intel(R) Core(TM) i7-4770K 3.50GHz CPU 4 cores computer with 32GB of RAM.

Unfortunately, no publications are available revealing more details about this player and describing its method. Therefore, we provide a short description. The Sancho player has implemented a piece detection method, and after estimating piece values, it uses this information to help Monte Carlo searching. The algorithm detects predicates that may represent a piece and correlates the number of such pieces with the goal score. If a correlation is above certain threshold, the player treats the predicate as a piece. A number of random playouts are run to establish the pieces values. The obtained piece value depends on its mobility and correlation between a number of pieces and playouts results.

A heuristic evaluation function using pieces values is used as the initial score for a new node added to UCT tree. The node's visit counter is initialized according to some weight function. After new node creation process, all simulations and score's updates behave like in the standard UCT algorithm. This makes that the initial distribution of playouts heavily depends on heuristic knowledge. If the initial heuristic reflects the real importance of the pieces, it helps to guide search into promising parts of the tree and not to waste simulations. If not, the values of the scores contradictory to the observations are gradually revised by successive simulations.

To complement the technical information above, we also describe our RBgPlayer specification. The program is written in C# using .NET Framework 4.5, and run on single machine with Intel(R) Core(TM) i5-2410M CPU 2.30GHz 4 cores and 12GB of RAM.

VI. RESULTS

We ran two sets of experiments, one of them using precomputed initial weights, while the other used only pure learning to determine proper weights. In the latter case, all material features weights were initially set to 1, while all positional features weights were set to 0. We tested our RBgPlayer against Dumalion, Sancho without piece detection, and Sancho with piece detection enabled. We used two time setups: one similar to those which are used during GGP competitions for large games, with 3 minutes startclock and 1 minute playclock; and another one with shorter times (60 seconds startclock, 20 seconds playclock) used only for experiments with precomputed weights. To test every setup we ran 20 matches (10 with RBgPlayer as white, and 10 as black).

Table II shows the results of experiments with RBgPlayer using precomputed weights, while Table III contains the results of with pure learning experiments. A single result consists of three numbers: the first is the percent of RBgPlayer wins, the second the percent of draws, and the third is the percent of RBgPlayer loses. The last column contains the results against Sancho with piece detection on, while the column before, with piece detection off.

TABLE II. PRECOMPUTED INITIAL WEIGHTS (WINS:DRAWS:LOSES PERCENT)

Game	startclock, playclock	Dumalion	Sancho	
			det. off	det. on
Chess	180, 60	100 : 0 : 0	100 : 0 : 0	15 : 5 : 80
	60, 20	100 : 0 : 0	95 : 5 : 0	25 : 5 : 70
Asymmetric Los Alamos	180, 60	100 : 0 : 0	100 : 0 : 0	50 : 0 : 50
	60, 20	100 : 0 : 0	100 : 0 : 0	75 : 0 : 25
Gardner	180, 60	100 : 0 : 0	40 : 5 : 55	20 : 15 : 65
	60, 20	100 : 0 : 0	45 : 10 : 45	25 : 5 : 70
Escort Breakthrough	180, 60	55 : 20 : 25	75 : 10 : 15	35 : 30 : 35
	60, 20	35 : 15 : 50	85 : 10 : 5	35 : 5 : 60

TABLE III. PURE LEARNING (WINS:DRAWS:LOSES PERCENT)

Game	startclock, playclock	Dumalion	Sancho	
			det. off	det. on
Chess	180, 60	100 : 0 : 0	95 : 0 : 5	0 : 0 : 100
Asymmetric Los Alamos	180, 60	100 : 0 : 0	80 : 0 : 20	5 : 0 : 95
Gardner	180, 60	95 : 0 : 5	10 : 0 : 90	15 : 10 : 75
Escort Breakthrough	180, 60	15 : 55 : 30	0 : 40 : 60	5 : 20 : 75

At first, one may observe that the table reflects precisely the playing level of GGP programs, which shows that Regular

Boardgames form a good base for a testing class for measuring the performance of GGP players. An expected trend observed in the table is that the GGP agents are playing worse as the game tree size grows. Escort Breakthrough has a larger branching factor than Gardner at the beginning, but after that, both values are similar. However the tree depth in Breakthrough is much smaller, which means random MCTS playouts are nearly two times faster achieving faster convergence and leading to definitely better results. The differences between Gardner, Los Alamos and Chess lie both in branching factors and game tree depths. These differences are reflected in our results. We can also observe that smaller timeouts, combined with game tree depth, favor RBgPlayer. It seems that in such games with complex rules short lookahead based on heuristic position evaluation leads to better results than small number of simulations (especially in an endgame phase, which is often crucial).

If, as our main reference point, we take GGP champion Sancho in the mode with piece detection off, we see that Gardner is the game with probability of winning close to equal for this player and the dedicated regular boardgames player. In slightly larger Los Alamos the GGP player chances to win plummet to zero. Gardner is still a complex, but relatively small game in human terms. A tempting conclusion, that any larger game leads to dramatic drop in GGP players efficiency, is not encouraging in terms of domain success. However, the fact that GGP players behave with every game like they are seeing it for the first time should be taken into account. Comparing the efficiency of Sancho against RBgPlayer without precomputed weights we can see a significant improvement in GGP agent scores. Without game-specific knowledge, RBgPlayer has too little time to gather reliable information about piece behavior to construct an accurate evaluation function.

The use of heuristic knowledge about piece mobility makes Sancho player with piece detection behave more like a dedicated player. Note that Sancho possesses only very limited knowledge. It does not detect board or other concepts but pieces. Moreover, there may be games where pieces are not detected properly because of the fixed threshold in the predicate-score correlation test. Also, weight values may be challenged as based only on random simulations. In spite of all these, this limited knowledge combined with engineering effort to make player computationally efficient, turns out to be more then enough to have clear advantage over RBgPlayer in most experiments. Initial heuristic guidance over the game tree to explore its most promising areas without costly computations, put Monte Carlo ahead of min-max with simple material and positional evaluation.

It should be also pointed out that the influence of heuristic evaluation strongly depends on the game tree size. In small games, sampling quickly dominates the initial heuristic values, while in large ones with computationally expensive simulations, heuristic evaluation remains dominant. Since UCT needs help exactly for large games this is very welcome behavior.

There is however no assurance that the method used by Sancho is accurate and safe from producing wrong results. What we have learned from our experiments, the weight assignment algorithm seems to have a tendency for overestimating the values of some types of pieces. It seems also to have problems with inequitable initial positions. For Asymmetric Los Alamos the heuristic does not work so well as in other cases.

A. Feature weights

Table IV shows the comparison between the values of pieces (material features weights) calculated by RBgPlayer's initial weight computation algorithm and Sancho's simulation-based algorithm. For every game, the obtained values have been normalized by the assumption that the pawn's value is 1. The values calculated by RBgPlayer are sums of the scores based on mobility and goal conditions. The goal condition part has nonzero value only for pawns and kings, and in such cases this value is put in the braces (thus for chess the pawn score 1.0 is obtained by the goal condition evaluated for 0.38 and the mobility evaluated for 0.62). The Sancho player has made different evaluations for the Asymmetric Los Alamos rook depending on the side: for white the weight is 1.5, while for black it is 1.2.

TABLE IV. PIECES VALUES

Piece	Chess		Assymetric Los Alamos		Gardner	
	RBg	Sancho	RBg	Sancho	RBg	Sancho
Pawn	1 (0.38)	1.00	1 (0.47)	1.00	1 (0.54)	1.00
Knight	2.50	3.69	1.97	2.80	1.57	4.34
Bishop	3.00	2.69	2.01	1.40	1.51	1.24
Rook	4.18	2.59	2.88	1.5/1.2	2.20	1.25
Queen	7.16	5.73	4.90	3.30	3.70	2.16
King	161.60 (158.48)	2.99	150.46 (147.71)	2.00	138.59 (136.24)	1.12

Although weights assigned to pieces differ in both approaches, they mostly coincide with the human judgment of which piece type is better. Scores relative to the pawn are usually smaller then given by humans (e.g. in chess), but this is a result of changes in the rules of the game. The fact, that pawn reaching the opponent's backrank causes an instant win increases the pawn's weight. The Sancho player has a tendency to overrate knights, and in Asymmetric Los Alamos this even influences rook score, as the simulations probably discover knight+rook as a complementary combination of pieces.

In Escort Breakthrough, the precomputed heuristic grants a very high value to the king, as it is the only piece allowing victory. This prevents RBgPlayer from losing its king to a much greater extent than in the pure learning case. Also, because without a king, a match normally goes on until turnlimit is reached, the small search horizon is enough to compete against UCT, which requires full length playouts.

B. Learning

The main difference between the process of learning weights during our experiments and that presented in [8], [9] is that in the GGP scenario in which we work, the weights values are updated from turn to turn, and a new game always begins without knowledge about previous experience. Thus the setup is more difficult, but there are also advantages. For example, our player has possibilities to adapt to the current situation.

It is well known that, e.g. in chess, weights of some pieces are different depending on the game phase (beginning, middle, endgame) or other pieces on board (e.g. two bishop bonus

which is usually a part of chess evaluation function). In the learning setup basing on material and positional evaluation, it is impossible to embed such nuances. However, as learning proceeds after every turn, the weights can automatically adapt to better suit to the current situation. During our experiments, we have observed that material evaluation for pieces was being changed between turns to reflect their power in the current state. A practical problem is time for such learning to be sufficient. In chess experiments, during the startclock, RBgPlayer was able to perform 10-30 learning simulations, and as the vast majority of them ended with a draw, no really significant weight changes were done.

This is not a problem in the precomputed initial weight values scenario, but in the learning only scenario, piece values in chess began usually to differentiate around the 8th-12th turn. Earlier, all piece types were evaluated as nearly equal in strength. During the middle game only a few chess simulations can be done during the learning phase. The number of middle phase simulations in Garder was about 15-20. This is not enough to obtain the best suited values, but enough to slightly push the weights in the right direction, and this works well in the scenario with precomputed initial weights. An interesting issue concerns endgame situations with some unused pieces. In this case, such pieces tend to have negative weights, which is often reasonable, because they block mobility of other pieces (although still contradictory with human evaluation of such situations). This occurs only when weights depend on pure learning. With precomputed weights, the values are larger and all material features are more stable.

C. Other Issues

An interesting aspect of a dynamically learned evaluation function is the usage of transposition tables. Although transposition tables speed up computations, the retrieved values are often outdated due to reevaluation of weights. However, the value stored for a state has a search depth level assigned, and it is used only if the current searching depth is not greater then that. This ensures that, during the move searching phase, the state evaluations will be recomputed using the current heuristic, and they will override values stored during shallow learning playouts. Experimentally, we did not observe disadvantages of using transposition tables, while using them usually results in reaching a deeper level in iterative deepening alpha-beta as used by RBgPlayer.

Another important issue is GDL code efficiency. In GDL, there are many ways to encode a single game, and the way it is done, has an essential impact on the number of computations needed to calculate a state, and so, on the performance of GGP players. During our experiments, we have been using the optimized style of codifying chess-like games, presented by Alex Landau the author of the Alloy GGP Player [25]. At the beginning we had been using a less efficient coding style, which resulted in larger number of losses by Sancho (in some cases even twice).

VII. Conclusions and Future Work

In this paper, we assess the progress of General Game Playing by comparing some state-of-the-art GGP players with an exemplary program dedicated to play in a smaller class of games called Simplified Boardgames. We have developed a min-max based player and confront it with two top GGP players on a number of games. Our player takes advantage of the fact that the game played is always a regular boardgame and evaluates game positions using material and positional evaluation functions, with weights learned during the play.

The obtained results lead to the conclusion that GGP programs play at an acceptable level on smaller games. Due to the careful, efficiency-oriented design, they are able to produce millions of simulations which pay-off when using MCTS methods. However, this approach disappoints when a game tree becomes bigger, in which case a relatively simple knowledge-based approach achieves better results. The experiment shows, where a limit of GGP players capabilities lies.

The outstanding results of Sancho with implemented boardgame detection demonstrate the benefit of having knowledge about essential game properties. This contradicts conclusions formulated in [26]. Based on this observation, we think that a key to GGP players improvement is a proper game type detection. It is worth noting that from the very beginning of GGP the natural division of games into single player and multiplayer is applied, and nearly all GGP players use special strategies for single player games (DFS, reduction to ASP [27]), different from those for multiplayer ones.

It may appear somewhat against the idea of generality, but defining GDL class as a union of specialized subclasses seems to be a promising research direction for improving GGP players efficiency. Such an approach shifts the burden into the following tasks: defining the proper set of classes, detecting game class membership, developing algorithms for particular classes. A proper class of games should be narrow enough, to allow players to use specific game type knowledge, but wide enough to cover a large part of the GDL class. It should be also possible to detect whether a game belongs to this class based on the properties of GDL predicates.

It may even be argued that the general approach based on straightforward UCT heuristics (like MAST, PAST, etc. [11]) and software engineering improvements will reach its limits soon. Starting to classify games using a multilevel hierarchy, and introducing a mechanism for detecting game classes seems unavoidable for getting further progress. It is rather obvious that, like in Problem Solving, the more general the method the weaker the performance: a general purpose algorithm will always be worse than a good algorithm designed specially for a given task. Of course, detecting game classes may be suitably combined with general knowledge-based approaches (like that in [12]).

A. Future Work

In accordance with the long term research goals stated above, we would like to improve the boardgames detection and playing algorithms in MCTS GDL-based players. In particular, we plan to make more efficient use of the knowledge about pieces in MCTS search.

Another goal is to extend further the regular boardgames class to cover also exceptional moves used in classical boardgames such as promotions, castling, en passant, etc. We plan to do this in a systematic way, without losing the property that move rules are determined generally by a regular language. For example, changing piece type after a move, moving more than one piece simultaneously, or history dependent moves – all these may be achieved by extending the alphabet.

We also would like to establish the regular boardgames class as a standard test class for GGP. To enable easy testing, we are going to provide an automatic translation system from regular boardgames into GDL (in a manner similar to that recently proposed for Card Game Description Language [15]) producing efficient GDL code. The present paper may be considered as the first step in this direction.

Finally, there are several interesting issues about improving regular boardgames player, including generation of opening/endgame libraries, and using more sophisticated evaluation function (including e.g. threats counting).

ACKNOWLEDGMENT

The authors would like to thank Steve Draper, co-author of Sancho player, for very useful comments, opinions and great support during the experiments.

This work was supported by Polish National Science Centre grants No 2014/13/N/ST6/01817 and No 2012/07/B/ST1/03318.

REFERENCES

[1] M. Genesereth, N. Love, and B. Pell, "General game playing: Overview of the AAAI competition," *AI Magazine*, vol. 26, pp. 62–72, 2005.

[2] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth, "General Game Playing: Game Description Language Specification," Stanford Logic Group, Tech. Rep., 2008.

[3] J. Pitrat, "Realization of a general game-playing program," in *IFIP Congress*, 1968, pp. 1570–1574.

[4] B. Pell, "Metagame: A new challenge for games and learning," in *Programming in Artificial Intellegence: The Third Computer Olympiad. Ellis Horwood*, 1992, pp. 237–251.

[5] ——, "A strategic metagame player for general chesslike games," in *AAAI*, 1994, pp. 1378–1385.

[6] Y. Björnsson, "Learning Rules of Simplified Boardgames by Observing," in *European Conference on Artificial Intelligence (ECAI)*, ser. FAIA. IOS Press, 2012, vol. 242, pp. 175–180.

[7] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Mach. Learn.*, vol. 3, no. 1, pp. 9–44, 1988.

[8] S. Droste and J. Frnkranz, "Learning the piece values for three chess variants," *International Computer Games Association Journal*, pp. 209–233, 2008.

[9] J. Baxter, A. Tridgell, and L. Weaver, "Learning to play chess using temporal differences," *Mach. Learn.*, vol. 40, no. 3, pp. 243–263, 2000.

[10] M. Genesereth and M. Thielscher, *General Game Playing*. Morgan & Claypool, 2014.

[11] H. Finnsson and Y. Björnsson, "Learning simulation control in general game-playing agents," in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[12] S. Haufe, D. Michulke, S. Schiffel, and M. Thielscher, "Knowledge-Based General Game Playing," *KI*, vol. 25, no. 1, pp. 25–33, 2011.

[13] J. Mhat and T. Cazenave, "A Parallel General Game Player," *KI*, vol. 25, no. 1, pp. 43–47, 2011.

[14] M. Schofield and A. Saffidine, "High Speed Forward Chaining for General Game Playing," in *Proceedings of the IJCAI-13 Workshop on General Game Playing (GIGA'13)*, 2013, pp. 31–38.

[15] J. Kowalski, "Embedding a card game language into a general game playing language," in *STAIRS 2014 - Proceedings of the 7th European Starting AI Researcher Symposium*, 2014, pp. 161–170.

[16] E. Cox, E. Schkufza, R. Madsen, and M. Genesereth, "Factoring General Games using Propositional Automata," in *Proceedings of the IJCAI Workshop on General Game Playing (GIGA'09)*, 2009.

[17] P. Kissmann and S. Edelkamp, "Instantiating General Games Using Prolog or Dependency Graphs," in *German Conference on Artificial Intelligence*, 2010, pp. 255–262.

[18] B. Pell, *Metagame in symmetric chess-like games*. University of Cambridge Computer Laboratory, 1992.

[19] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*. MIT Press, 1998.

[20] G. Tesauro, "Practical issues in temporal difference learning," *Mach. Learn.*, vol. 8, no. 3-4, pp. 257–277, 1992.

[21] ——, "Td-gammon, a self-teaching backgammon program, achieves master-level play," *Neural Comput.*, vol. 6, no. 2, pp. 215–219, 1994.

[22] D. F. Beal and M. C. Smith, "Temporal coherence and prediction decay in td learning," in *Proceedings of the 16th International Joint Conference on Artifical Intelligence - Volume 1*, ser. IJCAI'99, 1999, pp. 564–569.

[23] M. Mhalla and F. Prost, "Gardner's minichess variant is solved," *CoRR*, vol. abs/1307.7118, 2013.

[24] J. Kowalski and M. Szykuła, "Game Description Language Compiler Construction," in *AI 2013: Advances in Artificial Intelligence*, ser. LNCS. Springer, 2013, vol. 8272, pp. 234–245.

[25] A. Landau. (2014, may) Rewriting chess (again). Alloy GGP blog. [Online]. Available: http://alloyggp.blogspot.com/2014/05/rewriting-chess-again.html

[26] M. Swiechowski and J. Mandziuk, "Specialized vs. Multi-game Approaches to AI in Games," in *Proc. IEEE International Conference on Intelligent Systems (IS'14)*, 2014, pp. 243–254.

[27] M. Thielscher, "Answer Set Programming for Single-Player Games in General Game Playing," in *Logic Programming*, ser. LNCS, 2009, vol. 5649, pp. 327–341.