Evolving Chess-like Games Using Relative Algorithm Performance Profiles

Jakub Kowalski[⋆] and Marek Szykuła[⋆⋆]

Institute of Computer Science, University of Wrocław, Poland
{jko,msz}@cs.uni.wroc.pl

Abstract. We deal with the problem of automatic generation of complete rules of an arbitrary game. This requires a generic and accurate evaluating function that is used to score games. Recently, the idea that game quality can be measured using differences in performance of various game-playing algorithms of different strengths has been proposed; this is called Relative Algorithm Performance Profiles.
We formalize this method into a generally application algorithm estimating game quality, according to some set of model games with properties that we want to reproduce. We applied our method to evolve chess-like boardgames. The results show that we can obtain playable and balanced games of high quality.

Keywords: Procedural Content Generation, Evolutionary Algorithms, Relative Algorithm Performance Profiles, Simplified Boardgames, General Game Playing

1 Introduction

The idea of Procedural Content Generation has been found widely applicable to creating various parts of computer games. It can be used for generating every single part of the game, from textures and items, through levels and music, to AI opponents [1]. However, the most challenging task here is to generate a complete game [2–4]. The core of that task concerns game rules, which will specify the environment, the player role, and the plot. So far, several such attempts were made, creating games belonging to some restricted classes of possible rules. The most notorious one is Browne's Ludi system [5], which created two commercially successful boardgames. Also it is worth mentioning that METAGAME, one of the first general game playing systems producing symmetric chess-like games, also was equipped with a generator of rules [6].

Of course attempts of game generation do not end on boardgames, and some results concerning different genres have been published. Let us just mention

[⋆] Supported in part by the National Science Centre, Poland under project number 2014/13/N/ST6/01817.

[⋆⋆] Supported in part by the National Science Centre, Poland under project number 2013/09/N/ST6/01194.

here card games [7], strategy games [8], and Pac-Man-like grid-world games [9]. Arcade-style video games become a domain of special interest. The ANGELINA is an ongoing project generating complete games including rules, levels, and game characteristics [10]. Alternatively, a Video Game Description Language (VGDL) [11] game generator has been recently proposed in [12].

Although not so little research in the direction of generating complete games was made, the main question – how to judge the quality of arbitrary games and how to distinguish the good ones from the bad ones – still remains a key problem. As it is usually not very difficult to generate syntactically valid game rules belonging to some of the existing general game description languages, there are numerous syntactically valid and playable games which are not interesting from the human point of view.

In this paper we investigate the method of evaluation based on the assumption that games should be primarily sensitive to the skill of players. We present an extension of the method called Relative Algorithm Performance Profiles (RAPP) [13], which makes use of score differences between strong and weak AI players to evaluate games. We applied our method in the domain of the simplified boardgames [14], a modern language to describe chess-like boardgames, which is more concise and less restricted than METAGAME.

Instead of computing the quality of a game basing just on the assumption that we should be interested in games for which good algorithms play significantly better than bad ones, we present more methodical and sophisticated approach. First, for a given set of player strategies, we train our evaluating function using a set of *model* games, i.e. games which have desired properties and should be considered interesting. Then, to reduce computational effort, we use a generic tactic to choose the subset of strategies that will best reflect relations observed for the model. This allows us to value the game according to the level of similarity between relations of the algorithms for the evaluated game and for the model.

1.1 General Game Playing

The aim of *General Game Playing* (GGP) is to develop a system that can play a variety of games with previously unknown rules and without any human intervention. Using games as a test-bet, the goal of GGP is to construct universal algorithms performing well in different situations and environments. As such, GGP 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 [15]. The General Description Language (GDL) [16] created for that purpose can describe any n-player, turn-based, finite, deterministic game with full information. The expressiveness of the language was even enhanced by some extensions: GDL-II [17] which adds nondeterminism and imperfect information, and rtGDL [18] which adds asynchronous real-time events.

A somewhat concurrent approach is represented by the General Video Game AI (GVG-AI) Competition [19]. This is a brand new, yet very rapidly growing area of GGP research. The special Video Game Description Language (VGDL) was designed to describe limited class of Atari-like 2D arcade video games [11]. However, the players are not provided with the game rules which they have to understand; instead, they have a simulation engine which they can use to learn how the game behaves.

Although widespread research in GGP domain was initiated and motivated by IGGPC, it should be pointed out that the idea is dated much earlier. The first approach, concerning a class of fairy-chess boardgames has been done by Pitrat in 1968 [20], and the second contribution, aforementioned METAGAME, contains several works of Pell from 1992 [6].

1.2 Relative Algorithm Performance Profiles

The concept of player performance profiles has been used as a method for improving the level of game playing programs by comparing them with various opponent's strategies [21, 22]. The idea behind the RAPP is to use a comparison of playing agents not to evaluate their strategies, but to evaluate a game. This is an indirect approach, which focuses not on the fact how the game is built, but rather how it behaves. That makes RAPP a promising method for application in the GGP domains, where game descriptions are complicated and very sensitive (what actually applies to all GGP languages including GDL and VGDL).

The initial study concerning RAPP focuses on verifying if the concept is applicable in the domain of VGDL games [13]. The authors show that in human-designed games the differentiation between scores obtained by strong and weak algorithms is greater than in randomly generated or mutated games. This leads to the conclusion that good games should magnify the differences between the results of distinct algorithms.

More insightful research has been presented in [12], where RAPP has been used in a fitness function evaluating VGDL games. The function compares the scores of the following two algorithms: *DeepSearch* presented in details in the paper, and *DoNothing* which always returns the *null* action. A number of games were generated; many of them evaluated with near-perfect fitness, and some of them had interesting properties and features. Yet, for creating high quality games, comparable to human-designed ones, the necessity of refining the fitness function to identify more aspects of the game has been stated.

1.3 Simplified Boardgames

Simplified boardgames is a class of games introduced in [14] and slightly extended in [23] (see [24] for an alternative extension). The simplified boardgames language describes turn-based, two player, zero-sum games on a rectangular board with piece movements being a subset of a regular language. The player can win by moving a certain piece to a fixed set of squares, by capturing a fixed amount of the opponent's pieces of a certain type, or by bringing the opponent into the position where he has no legal moves. Every game has assigned a turnlimit, whose exceedance causes a draw.

The language can describe many of the Fairy Chess variants, including games with asymmetry and moves that can capture own pieces. However, the regularity of the description, besides being easily processable and concise, poses some important limitations. Actions like castling, en-passant, or promotions are impossible to express, as all the moves depending on the move history or (according to the statement within the original language description) moves depending on the absolute location of the piece. However, the latter restriction can be bypassed, so it is possible to describe e.g. chess pawn initial two-square advance.

The 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 and Δy are relative column/row distances, and $on \in \{e, p, w\}$ describes the content of the 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 such that $1 \leq i \leq k$, the content condition on_i is fulfilled by the content of the square $\langle x + \sum_{j=1}^{i} \Delta x_j, y + \sum_{j=1}^{i} \Delta y_j\rangle$. If move rule w is applicable in the 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.

2 Method

RAPP is an approach to evaluate the quality of games by measuring results of different controllers playing them.To use this method, one needs a set of algorithms serving as controllers, and a set of approved games that will be used as the model set. Then, a game is evaluated by running the algorithms on it, and comparing their results with those obtained by playing the model set of games.

2.1 Games in the Example Set

As a set of exemplary, well-founded games, we have chosen ten human-written variants of fairy-chess (including chess itself). Most of these games use the orthodox chess pieces, and have a similar starting state, e.g. the first line of pawns, one king, or pawns promotion.

Our set of example games contains: *Gardner*, *Action Man's Chess*, *Petty Chess*, *Half Chess*, *Demi-chess*, *Los Alamos Chess*, *Cannons and Crabs*, *Small-Deacon Chess*, *Shatranj*, and *Chess*. The detailed rules of each game can be seen in [25]. In some cases, we had to omit some of the special game rules (eg. castling, en-passant, promotion) to fit the game within the simplified boardgames framework. The concept of chess check is replaced by the goal of capturing the king. We also changed promotion of the weakest piece into the winning condition. To each game we assigned a turn limit, whose exceeding causes a draw.

2.2 Evaluating Algorithms

As the evaluating algorithms we have used a min-max search with a constant depth and different heuristic functions evaluating game states. In total, there are 16 distinct player profiles, which differ by strategic aspects they cover. Because such a big number of profiles is impractical to evaluate a large number of games, by analyzing their behavior on the example games, we narrowed this set to a subset of algorithms that produce most characteristic results.

All the heuristic functions are sums of weighted game features. The primary set of features contains material and positional features, which are general approaches to evaluate a state in a chess-like game [26]. For a given type of piece, the material feature is the difference between the numbers of pieces of the two players. The value of a positional feature for a given piece and square is 1 if such piece occupy the square, and 0 otherwise. Thus, the weight of this feature determines the willingness of the player to put a piece of that type on the square.

Implemented profiles use two strategies of assigning weights to material and positional features. The first is **Constant**, and it assigns to every material feature the square root of the number of squares on the board, and zero to all positional features. The alternative strategy is **Weighted**, which bases on the mobility of each piece as presented in [23]. For every available move of a piece, a probability that this move will be legal is estimated. The weight of the positional features is the sum of the probabilities of all moves that are legal from the given square divided by the number of squares on the board. The weight of the material features is the sum of all positional features for the given piece.

This primary set of algorithms can be improved by using the following more subtle heuristics:

Mobility: This computes for each square the square root of the number of legal moves ending on this square, and adds to the score of the game state the difference between the sum of these values between players. The aim of this heuristic is to promote expansion of pieces and covering a large area of the board.

Control: This is a similar, yet in some sense opposite, strategy to the previous one. For every square it computes which player wins a maximal sequence of captures, i.e. who has more capturing moves, assuming that the square is occupied by an opponent's piece. The score of the game state is modified by the difference between the numbers of squares controlled by the players. This strategy assists protection of own pieces and points out holes in opponent's defense, posing a threat to unprotected pieces.

Goal: This modifies the values of pieces and squares that are crucial to win, according to the method presented in [23]. The weight of pieces occurring in the game's terminal conditions are increased depending on their numbers in the initial state. The larger it is, the less important is an individual piece. Moreover, for the pieces whose aim is to reach some squares, the weights of positional features are increased for the nearby squares (using the number of moves required to move from a square to another as the distance measure). This heuristic, in contrast to Mobility and Control, is computed at the beginning of the game and does not tune weights during the gameplay.

We have used Constant or Weighted material heuristics with all combinations of the three positional ones. From this point, we will use shortened names to identify these combinations; for example *CGM* stands for Constant+Goal+Mobility, while *WC* denotes Weighted+Control.

2.3 Results

We have tested the performance of all sixteen algorithms by playing against each other the games from the example set. For every game, each pair of the heuristics played 100 times using three-ply deep min-max search, which gives 12,000 plays to create a single game profile. The gathered data is the matrix of the average scores between every pair of the heuristics. A win of a play was counted as 1 point, a draw as 0.5, and a loss as 0.

To evaluate how good each heuristic is, we took the average of its results from playing against all other heuristics. We illustrate the performance of the algorithms taking its average score for the games from the example set. This is presented by the bars of *example games* in Figure 1.

From this point, we can see a clear tendency in the cases of some algorithms. Heuristic *CG* is undoubtedly the worst, with *C* being the second worst. All heuristics based on weighted material and position values performed above the average score of 0.5. *WGC* is counted as the best heuristic, yet the differences between the top three are very small (*WGC* – 0.66360, *WC* – 0.64927, *WCM* – 0.64780). However, the standard deviation shows that the set of example games is not consistent, and some games have very different profiles than the others.

3 Selection of the Model

The results from the previous experiment show us that some games from the example set behave substantially different in the set of example algorithms. For example, the performance of *CM* and *WGM* in *Petty chess* (0.653 and 0.745, respectively) is vitally better than in all other games, while the performance of *CGM* for that game is the lowest (0.12). Hence, our first task is to narrow the set of example games into a smaller set, which will contain games with similar behavior. Then it will be used as the point of reference – the model for generating game rules.

Due to the computational cost, we also needed to narrow the set of algorithms used as evaluation profiles. Choosing the set of representative heuristics for a given model set of games is the second task covered in this section.

3.1 Selecting Model Games

For two games $\mathcal{G}$ and $\mathcal{H}$, given their profile matrices $P_{\mathcal{G}}$, $P_{\mathcal{H}}$ obtained by running n heuristics, we can calculate their level of resemblance in a standard way as the pairwise distance between these matrices:

$$\mathrm{dist}(\mathcal{G},\mathcal{H}) = \frac{\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}(P_{\mathcal{G}}[i,j]-P_{\mathcal{H}}[i,j])^2}{n(n-1)/2}. \quad (1)$$

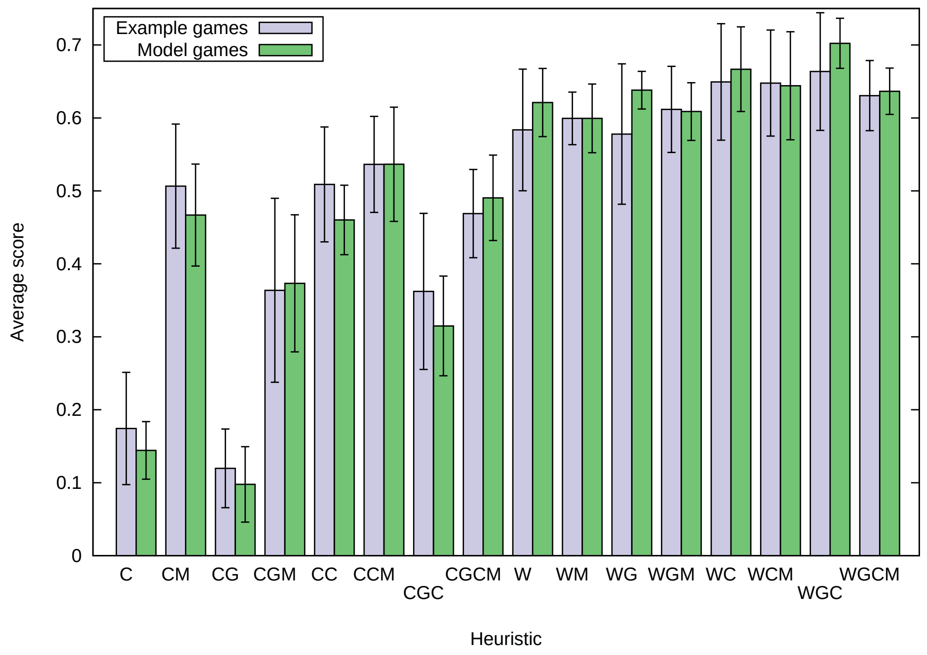

Fig. 1. The average score and the standard deviation of the sixteen algorithms on the sets of example and model games.

We can extend this definition to sets of games in the way that the distance of a set is the sum of the distances between every pair of games from this set.

We computed distances of all subsets of the example games, and decided to divide the example set into two subsets. The larger set in the best partition is *Action Man's Chess*, *Cannons And Crabs*, *Chess*, *Los Alamos*, *Shatranj* and *Small-Deacon Chess*. This subset was evaluated as tenth among the sets containing six games. On the other hand, the set of the remaining games is one of the worst four element sets.

From the obtained results, we draw a conclusion that the larger subset represents a type of games which we are interested in. These games are significantly different, yet share enough similarities to be relatively close in terms of performance of the algorithms. The set also contains *Chess*, which is arguably one of most popular board games with very desired strategic properties. Thus, we decided to use this set as our *model set* of games.

The difference in performances of particular algorithms between playing the model set and the full example set of games is presented in Figure 1. There are significant variations in scores of some algorithms and, as expected, in most cases the standard deviation is smaller.

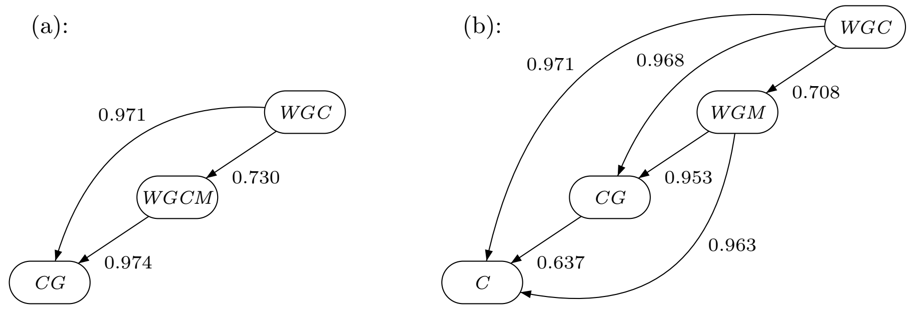

Fig. 2. Representation graphs of the relative performances in the models with (a) 3 algorithms and (b) 4 algorithms. The arrows are directed from the stronger toward the weaker algorithm, and their labels contain the average score.

3.2 Selecting Evaluation Algorithms

Having the model set, we can evaluate how similarly a given game behave to the model, by comparing the relative performances of the heuristics. The more heuristics we will use, the more accurate evaluation will be. The drawback is that the required number of tests grows quadratically in the number of heuristics. For this reason, we needed to find a method of selecting a small number of algorithms, which stand as the best representatives for the model games.

We have decided to selected the algorithms that are most distant to each other in terms of their relative performances. The reasoning behind this idea is twofold. First, the selected heuristics are substantially different, which allows further evaluation to incorporate more game features. Second, we take into account only the most essential differences in performance, which are easier to capture during evaluation using a smaller number of test plays. If the difference in the performance of two algorithms is small, it could be easily misjudged, depending on the number of test plays.

Let P_{model} be the matrix containing the average relative performances on the model games of some k algorithms. We can define its spread as follows:

$$\text{spread}(P_{\text{model}}) = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} |P_{\text{model}}[i,j] - 0.5|^2. \tag{2}$$

Then, for the given k, we can compute the matrices for all k-subsets of the example algorithms and calculate their spreads. We performed this computation for $k \in \{3, 4, 5\}$, and found the subsets with the largest spreads. For $k = 3$ this is *CG*, *WGC*, and *WGCM*; for $k = 4$ this is *C*, *CG*, *WGM*, and *WGC*; for $k = 5$ this is *C*, *CG*, *WGM*, *WGC*, and *WGCM*. In further experiments, we will use two models, using the best 3-subset and the best 4-subset, respectively. A visualization of the relative performance of the algorithms in these models is shown in Figure 2.

3.3 Fitness Function

According to our RAPP method assumptions, the fitness of a game should be based on the distance between the matrix the algorithm performances for that game and the average performance matrix of the model games. This, however, is not enough to ensure that desired game properties will be fulfilled. There are several features that may be important for a good boardgame, e.g. balance between players, complexity of the rules, branching factor, game length, and usage of the pieces [27]. Our goal was to keep the fitness formula as simple and general as possible, so we decided to choose two features we perceive as the most crucial in the case of the simplified boardgames: balance between players and game length being not too short.

Let $score_w$ be the percent of points scored by the white player during p test plays, and $score_b$ be the number of points scored by the black player. Then $B = |score_w - score_b|/p$ is the balance between the players. Let say that a play is *too short* if it ends in 10 turns. If during the test s games were qualified as *too short*, then $Q = \frac{s}{p}$ is the measure of the game's *quickness.* At last, let D be the modified formula (1) of distance between the game's matrix of the algorithm performances and the model matrix, where the absolute value of the difference is used instead of the power of two. Then, our fitness function is calculated as follows:

$$f = \begin{cases} (1-D)(1-B)(1-Q) & \text{if the game is } \textit{playable}, \\ -1 & \text{otherwise.} \end{cases} \tag{3}$$

The game is considered as *playable* if during the test all plays fit within the given timelimit. This function roughly matches with our intuition about that when a given game should be considered better than the others. We made some experiments using the unmodified formula (1), yet because it usually returns very small values, the overall fitness was high, and the differences between good games and slightly-less-good games were very small. For this reason we decided to stricter results to increase the impact of D in the formula, and make the differences in fitness more visible.

4 Evolving New Games

Our goal is to generate games which have similar properties, in the sense of efficiency of strategies, to the games in the model. We use RAPP to achieve that, and evolutionary search over a constrained subset of the simplified boardgames.

4.1 Generating Games

For the sake of efficiency and reduction of the search space, we restricted our generating mechanism to produce only chess-like games. By that, we mean the games fulfilling the following additional conditions:

- The initial position is symmetric and contains two rows of pieces for both players. The front row contains only the pieces called *pawns* or empty squares. The back row contains, among other pieces, one piece of the *king* type.

- The terminal conditions are symmetric. A player wins by capturing the enemy king, or by reaching the opponent's back row with a pawn.

To generate a new game, we start by selecting randomly parameters *width* and *height* of the board, the number of *non-winning* types of figures (in addition to always present *king* and *pawn*). For the purpose of our experiments, *width* and *height* are taken from $\{6, 7, 8\}$, and the number of figures from $\{3, 4, 5\}$. Parameter *turnlimit* is computed by the formula $3 \times \mathit{width} \times \mathit{height} + r$, where r is a uniformly random number taken from $\{0, \ldots, 19\}$.

Generating the initial position is straightforward: In the front row every square can be either empty with probability 0.1 or occupied by a pawn. The king is placed uniformly at random in a column of the back row, and the remaining squares are either empty with probability 0.1 or filled by a non-winning piece chosen uniformly at random.

The remaining part, generating regular expressions for descriptions of piece rules, is more challenging. First, we generate a lot of *raw* move patterns; they will be used as building blocks for piece rules. A single raw move pattern is a list of tuples $(\Delta x, \Delta y, \mathit{on}, \mathit{star})$, where Δx and Δy are signed integers that define the horizontal and vertical shifts of the moves, *on* is from $\{e, p, w, ep, ew, pw\}$ and defines allowed contents on the destination square (e – empty, p – enemy's piece, w – own piece), and *star* is a logical value indicating whether this tuple can be repeated with Kleene star. There is also a set of modifiers extending the pattern: *FB* mirrors the pattern across the horizontal axis, and *ROT* rotates the pattern through 90°, 180°, and 270° about the relative origin $(0, 0)$. For example, the orthodox rook can be described as $(0, 1, e, \mathit{true})(0, 1, ep, \mathit{false})[\mathit{ROT}])$.

Generating raw move patterns is based on a number of parameters. The process begins with picking at random a list of $(\Delta x, \Delta y)$ pairs. Exemplary probability values for each possible pair are listed in Table 1(a). New elements are added to the list as long as another probability test is passed. Next, we extend the list by adding *on* values. The exemplary probability table used for that purpose is shown in Table 1(b). We use a separate set of probabilities to determine the content of the destination square in the last part of a pattern. In particular, we do not allow self capturing, while it is possible to step over own figures during the movement. The last step is to add modifiers. As long as the probability test of the *modifier_prob* parameter (e.g. 0.1) is passed, one modifier is added to the pattern (see Table 1(c) for exemplary values). When *STAR* modifier is drawn, it is applied to a random tuple in the list by changing the *star* value to true. Also, if $\Delta x > 0$ occurs in the pattern, we mirror it across the vertical axis to avoid side asymmetry. This completes generating raw move patterns.

Next, we generate each piece by choosing raw patterns, whose sum will be used as the movement language. This process is guided by the following two parameters: the chance of adding a new raw pattern to the set, and the maximal mobility of the piece. Because the values of these parameters are different for king and pawn, and non-wining pieces, we can enforce limited mobility for kings and pawns, to make resulting games more chess-like.

Table 1. Exemplary probability parameters used in generating games for picking at random: (a) relative coordinates, (b) square content, (c) pattern modifiers.

$\Delta y \backslash \Delta x$	0	+1	+2	+3
+3	0.6%	0.6%	0.6%	0.6%
+2	7%	7%	3.5%	0.6%
+1	20%	14%	5%	0.6%
0	0%	16%	7%	0.6%
-1	7%	7%	1.4%	0.6%

(a)

on	*e*	*p*	*w*	*ep*	*ew*	*pw*
normal	42%	14%	14%	7%	7%	14%
last	33%	33%	0%	33%	0%	0%

(b)

modifier	*FB*	*ROT*	*STAR*
probability	50%	25%	25%

(c)

A weak point of this approach is that it heavily relies on human designed values. We have used four different settings, including the one presented here. The main differences in the settings we used were in probabilities of long-range jumps, probabilities of backward moves (also with additional possibility of $\Delta y = -2$), chances of applying modifiers, and probability of enlarge a raw move pattern.

4.2 Genetic Operators

Let n be an even population size. Given fitness values, we select $n/2$ pairs of parents using the roulette-wheel method. Although it never happened in practice, in this method if all games from the population have fitness less than or equal to zero, the uniform selection is used instead. Every pair of parents produce two offspring using the uniform crossover, which independently swaps some squares of the initial position (except the squares containing kings) and some piece rules.

There are two types of mutation, both independently modifying an offspring with some small probability. The *piece mutation* regenerates the rule of a random piece. The *position mutation* changes the content of a random square. If the square belongs to the first row, a pawn is replaced by the empty square and vice versa. If the square is occupied by the king, it is swapped with some other second row square. In the remaining cases, the square content is replaced by a random non-winning piece or the square is left empty (with uniform probability).

The next generation is created by choosing the best n games from the population of parents and children. The evolution process stops when the maximum number of generations is done.

5 Experiments

We tested our method using the extracted set of six model games and the two variants using 3 and 4 algorithms. We generated 200 games (using four sets of parameters), and evaluated each of them by 3 or 4 algorithms. In all our tests, we have used 50 min-max plays (25 per side) with depth 3 to compare every pair of algorithms and obtain the values to the profile matrix of the game. In the same way, we also evaluated all the example games.

Table 2. Comparison of game evaluation between the sets of example games, randomly generated games, and evolved games. The last column contains the results of the test with increased population size and mutation rate.

Variant	3 algs.			4 algs.			4 algs., population 16		
	Max.	Avg.	Promising	Max.	Avg.	Promising	Max.	Avg.	Promising
Evolved	0.971	0.911	100%	0.959	0.916	100%	0.978	0.922	100%
Generated	0.907	0.671	29%	0.942	0.704	34%			
Example	0.858	0.811	90%	0.959	0.859	100%			

The last test used evolution. For both variants with 3 and 4 algorithms we made 12 runs of evolution with populations of size 10, and both mutation rates equal to 5%. Additionally, we made 12 runs for the variant with 4 algorithms, and increased population size to 16 and mutation rates to 10%. In all these cases, the number of generations was 20.

5.1 Overview

We say that a game is *promising* if its balance factor B of the fitness function is less than 0.2 and its Q factor is less than 0.05. Such games may not have a great score taking into account the set of model games, but their properties assure that they should be at least human playable.

For all sets of games (example, generated, and evolved), we used two measures of comparison: the maximal fitness within the set and the average of the fitness of *promising* games. We are convinced that the last one is very important, as it shows the potential of the set. It may happen that the top game is not the best one from the human point of view; then the other candidates should be as good as possible in terms of accordance with the model. At last, we show the percent of *promising* games within the sets. These results are presented in Table 2.

As we can see, in all variants the best individual score was obtained by the evolutionary approach. Moreover, the average score of all final populations is also the highest. On the other hand, a pure generation can create a game even better than the best one from the example set (as in the variant with 3 algorithms), yet it is not so likely. The average score is low, as it is the fraction of promising games. Thus, the pure generation is too general and too random to be seen as a proper method just by itself, and there is a necessity of combining it with some score-improving method as evolution process. The score of the example games is not so high as one might think; this is due to the fact that the model reflects the average relations observed between the games, so every individual is likely to be distinct to that average. However, the games from the model set were usually rated better than the other example games that are outside the model.

There are two important observations. One is that, a larger number of algorithms indeed yields a better ordering of games and evaluations closer to expected, as the score is obtained using a more detailed model. The second is that, the evaluation method is very play-sensitive, i.e. the scores of one game obtained by consecutive tests can vary. Although the general tendency, whether a game is good, average, or poor, usually remains clear, the detailed results may be significantly different. This can influence the ordering of games and so the evolution process. However, to achieve better stability, it should be sufficient to increase the number of plays used for game evaluation.

5.2 Evolution results

Figure 3 presents the course of the evolution in the variant with 4 algorithms and population of size 16. This look similar in the other variants, but in the variant with 3 algorithms, the density of final scores distribution is visibly lower.

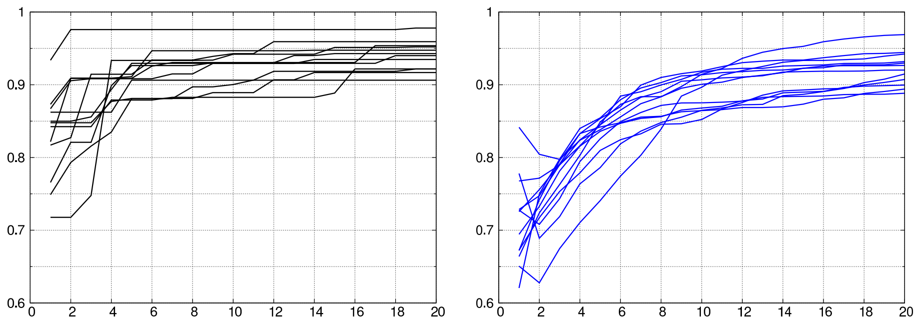

Fig. 3. Visualization of our evolutionary runs in the variant with 4 algorithms and population of size 16. The left and right charts show respectively the maximum and average fitness values of playable games in every iteration.

It can be seen that the process of evolution can increase the quality of the population to a decent level even when the initial population is poor. What is to be expected, the improvement of the best individual takes place stepwise, as it is not so easy to obtain a better game in every generation. On the other hand, the improvement of the average population is more smooth. Also, usually, except the first few generations, all games remaining in the populations are *promising*, so dysfunctional individuals are quickly discarded.

An example of an evolved game is presented in Figure 4. This game appeared in the 16-th generation and obtained score 0.9538. Its rules are not too complicated and human-readable, and it requires a non-trivial strategy since the beginning. Note that an opening using ♘ blocks the pawn advance of the opponent. Thus, to be able to move out the other pieces, and simultaneously prevent opponent's expansion, the player has to move his own pawns in a clever way.

Although pawns seem to be powerful and advance rapidly, it is impossible to move them to the opponent's backrank, so capturing the king remains the only reachable goal. It is common between best-scored games that pawn usefulness is very limited, e.g. they cannot advance but only move sideways. It seems that

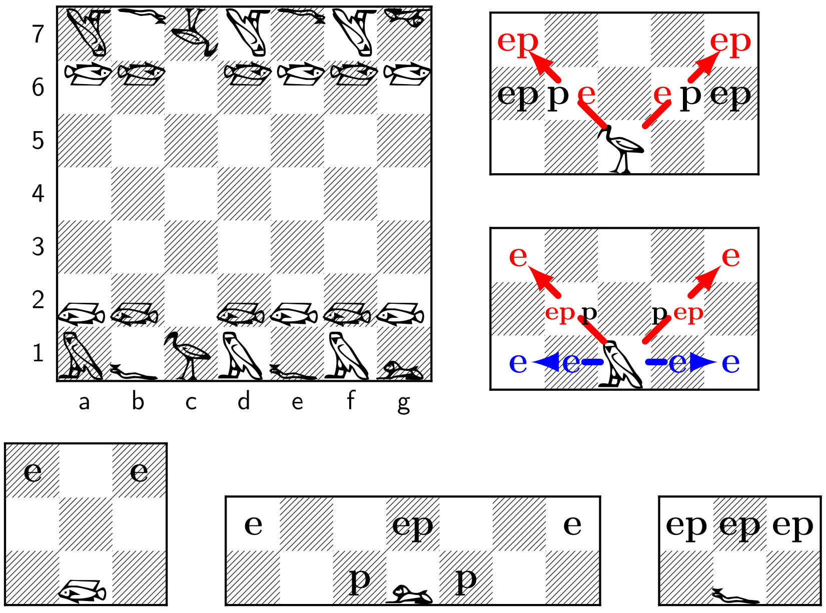

Fig. 4. Initial position and piece rules for the evolved game. Destination square content (from $\{e, p, w\}$) designates legal moves. Moves with consecutive requirements are distinguished by colors and arrows. The pawn symbol is , and the king symbol is . The turnlimit is 160.

pawn movements are so crucial and hard to control that it is easier to gain better game flow stability by reducing their impact on the game than by finding a set of moves that is not too strong yet still substantially useful.

Also we have found that the rules which seem pretty understandable in their regular expression form are often very hard to be translated on the board and remembered during a play. It seems that by generating we can easily produce movement rules which are reasonable, yet over-complicated from the human point of view. This observation addresses the problem of differences between generating game rules for humans and for AI players (e.g. to be used during GGP competitions).

6 Conclusion

We proposed an extension of the RAPP method evaluating the quality of a game using the performances of game-playing algorithms. The original approach was based on the assumption that in well-designed games, strong algorithms should significantly outperform weak ones. In this paper, we presented a more sophisticated solution, which uses more detailed information concerning the mutual relations of the given algorithms, and thus, it is able to catch and evaluate more subtle aspects of games. However, to make this working, one needs a set of model games, which have desired strategic properties, and use it to actually train the evaluation function.

The previous application of RAPP concerns VGDL games [12], which are one-player puzzles focusing on maximizing the player's score. We have shown its application in the domain of two-player, zero-sum games belonging to the class of simplified boardgames. We also tested it with different numbers of algorithms used for performance comparison in the evaluation function. In contrast with the previous study using only 2 algorithms, we provided an evaluation function in a more general form, which can be applied with any number of strategies, and allowed us to perform tests with 3 and 4 algorithms.

Finally, we applied our method to an evolutionary system, which was able to produce in some cases even "more model" games than these actually used in the model set. Not all of the top rated games could be evaluated as well-designed by human players. The reason for that is mostly abnormality and large complexity of movement rules. However, this issue can be solved in two ways: by improving move generation techniques and their parameterization, or by extending the fitness function to take into account features like complexity of the rules or pieces usefulness.

A nice property of the RAPP method is its generality. The presented methods of extracting the set of model games and model algorithms are domain independent, and can be used for any types of games. Unfortunately, the task of finding sets of example games and algorithms requires human expert knowledge on the subject. Making these tasks knowledge-free is a potential field of further study. In the case of example algorithms, the simplest (yet probably of low quality) approach is to use an MCTS algorithm, where the strength of a player is determined by its timelimit.

We confirm the conclusions from [12], that the best use for a RAPP-based game generation is being a fine sieve, with a human intervention at the last stage, where a man can choose the best game from the given results. Also, we confirm that the game evaluation, as it requires a significant number of simulations, is very time-consuming. However, given very promising results, it is a choice of preferring quality over quantity. The expected practical usage of such generated games, concerning e.g. providing interesting and novel games for the GVG-AI competition [19], requires obtaining just over a dozen games per year.

References

1. Shaker, N., Togelius, J., Nelson, M.J.: Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Springer (2015)
2. Nelson, M.J., Mateas, M.: Towards Automated Game Design. In: AI* IA 2007: Artificial Intelligence and Human-Oriented Computing. (2007) 626–637
3. Togelius, J., Nelson, M.J., Liapis, A.: Characteristics of Generatable Games. In: Proceedings of the FDG Workshop on Procedural Content Generation. (2014)
4. Zook, A., Riedl, M.O.: Automatic Game Design via Mechanic Generation. In: AAAI. (2014) 530–537

5. Browne, C., Maire, F.: Evolutionary game design. Computational Intelligence and AI in Games, IEEE Transactions on **2**(1) (2010) 1–16
6. Pell, B.: METAGAME in Symmetric Chess-Like Games. In: Programming in Artificial Intelligence: The Third Computer Olympiad. (1992)
7. Font, J.M., Mahlmann, T., Manrique, D., Togelius, J.: A Card Game Description Language. In: Applications of Evolutionary Computation. Volume 7835 of LNCS. Springer (2013) 254–263
8. Mahlmann, T., Togelius, J., Yannakakis, G.: Modelling and evaluation of complex scenarios with the Strategy Game Description Language. In: CIG. (2011) 174–181
9. Togelius, J., Schmidhuber, J.: An experiment in automatic game design. In: CIG. (2008) 111–118
10. Cook, M., Colton, S.: Multi-faceted evolution of simple arcade games. In: CIG. (2011) 289–296
11. Schaul, T.: A video game description language for model-based or interactive learning. In: CIG. (2013) 1–8
12. Nielsen, T., Barros, G., Togelius, J., Nelson, M.: Towards generating arcade game rules with VGDL. In: CIG. (2015) 185–192
13. Nielsen, T., Barros, G., Togelius, J., Nelson, M.: General Video Game Evaluation Using Relative Algorithm Performance Profiles. In: Applications of Evolutionary Computation. Volume 9028 of LNCS. (2015) 369–380
14. Björnsson, Y.: Learning Rules of Simplified Boardgames by Observing. In: ECAI. Volume 242 of FAIA. IOS Press (2012) 175–180
15. Genesereth, M., Love, N., Pell, B.: General game playing: Overview of the AAAI competition. AI Magazine **26** (2005) 62–72
16. Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General Game Playing: Game Description Language Specification. Technical report, Stanford Logic Group (2008)
17. Thielscher, M.: A General Game Description Language for Incomplete Information Games. In: AAAI. (2010) 994–999
18. Kowalski, J., Kisielewicz, A.: Game Description Language for Real-time Games. In: GIGA. (2015) 23–30
19. Perez, D., Samothrakis, S., Togelius, J., Schaul, T., Lucas, S., Couëtoux, A., Lee, J., Lim, C., Thompson, T.: The 2014 General Video Game Playing Competition. CIG (2015)
20. Pitrat, J.: Realization of a general game-playing program. In: IFIP Congress. (1968) 1570–1574
21. Jaśkowski, W., Liskowski, P., Szubert, M., Krawiec, K.: Improving coevolution by random sampling. In: GECCO. (2013) 1141–1148
22. Szubert, M., Jaśkowski, W., Liskowski, P., Krawiec, K.: The role of behavioral diversity and difficulty of opponents in coevolving game-playing agents. In: EvoApplications 2015. Volume 9028 of LNCS., Springer (2015) 394–405
23. Kowalski, J., Kisielewicz, A.: Testing General Game Players Against a Simplified Boardgames Player Using Temporal-difference Learning. In: CEC, IEEE (2015) 1466–1473
24. Gregory, P., Björnsson, Y., Schiffel, S.: The GRL System: Learning Board Game Rules With Piece-Move Interactions. In: GIGA. (2015) 55–62
25. : The Chess Variant Pages. http://www.chessvariants.org/ (September 2015)
26. Droste, S., Fürnkranz, J.: Learning the piece values for three chess variants. International Computer Games Association Journal (2008) 209–233
27. Hom, V., Marks, J.: Automatic design of balanced board games. In: AIIDE. (2007) 25–30