Evaluating Chess-like Games Using Generated Natural Language Descriptions

Jakub Kowalski*, Łukasz Żarczyński, and Andrzej Kisielewicz**

[1] Institute of Computer Science, University of Wrocław, jko@cs.uni.wroc.pl
[2] Institute of Computer Science, University of Wrocław, luk.zarczynski@gmail.com
[3] Institute of Mathematics, University of Wrocław, kisiel@math.uni.wroc.pl

Abstract. We continue our study of the chess-like games defined as the class of Simplified Boardgames. We present an algorithm generating natural language descriptions of piece movements that can be used as a tool not only for explaining them to the human player, but also for the task of procedural game generation using an evolutionary approach. We test our algorithm on some existing human-made and procedurally generated chess-like games.

1 Introduction

The task of Procedural Content Generation (PCG) [1] is to create digital content using algorithmic methods. In particular, the domain of games is the area, where PCG is used for creating various elements including names, maps, textures, music, enemies, or even missions. The most sophisticated and complex goal is to create the complete rules of a game [2–4].

Designing a game generation algorithm requires restricting the set of possible games to some well defined domain. This places the task into the area of General Game Playing, i.e. the art of designing programs which can play any game from some fixed class of games. The use of PCG in General Game Playing begins with the Pell's *Metagame* system [5], describing the so-called Symmetric Chess-like Games. The evaluation of the quality of generated games was left entirely for the human expert.

One of the most prominent PCG examples is Browne's *Ludi* system [6], which used genetic programming combined with a simulation-based self-play evaluation function, based on the broad range of aesthetic measures, to generate combinatorial boardgames. It produced the first fully computer-invented games to be commercially published.

Experiments on evolving rules of the card games, using grammar-guided genetic programming combined with simulation-based fitness function and MCTS

* 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 2015/17/B/ST6/01893.

agents, has been described in [7]. In [8], the authors introduce a psychology-based way of "fun" measurement, and evolve game rules for the two-dimensional Pac-Man-like games using a hill-climbing approach. The goal of the ongoing project ANGELINA is to generate complete arcade-style 2D games, including rules, levels, and game characteristics [9]. An initial work towards the game rules generation for the General Video Game Playing [10] has been presented in [11].

As "fun" is something too difficult to measure, the evaluation of the game's quality is restricted usually to its strategic properties. The good game should be playable, fair, and complex enough to ensure a proper level of challenge. When considering games for the AI, e.g. for General Game Playing competitions [10, 12], these requirements are mostly sufficient. However, when designing a game for humans, we should also restrict the rules to be not too complex and, even more challenging, to be somehow intuitive and easy to learn. In this paper, we tackle the problem of measuring the complexity of a chess-like game description from the perspective of a human player.

The descriptions of non-standard chess-like pieces are usually based on the analogies to well known pieces or movement patterns [13]. We use the similar approach, first decomposing a given piece into the parts based on the chess movement classification [14], and then creating the piece's description in the natural language and evaluating its complexity. Thus, we call our method the NLD (Natural Language Description) evaluation.

The algorithm presented in this paper describes and evaluates fairy chess pieces (i.e belonging to the family of unorthodox chess variants) given as regular expressions in the Simplified Boardgames standard [15]. Yet, the method can be generalized to a much larger class of games with the whole variety of features represented by other chess variants. We use our algorithm to test the quality of the obtained results on a series of human-made and procedurally generated pieces. We also apply it to the sets of games evolved for their strategic properties in [16, 17], to reveal those among them that have an interesting gameplay and are easy to understand by a human player.

2 Simplified Boardgames

Simplified Boardgames is the class of fairy chess-like games introduced by Björnsson in [18]. The language describes turn-based, two player, zero-sum chess-like games on a rectangular board with piece movements described by regular languages and independent on the move history. It was slightly extended in [19], and further formalized in [15].

The language can describe many of the fairy chess variants in a concise way, including games with asymmetry and position-dependent moves (e.g. chess initial double pawn move). The usage of finite automata for describing pieces' rules, and thus for move generation, allows fast and efficient computation of all legal moves given a board setup. However, it causes some important limitations, e.g. it is impossible to express actions like castling, en-passant, or promotions.

2.1 Language definition

Here we follow [15] to provide a shortened necessary introduction. The game is played between the two players, *black* and *white*, on a rectangular board of size $width \times height$. White player is always the first to move.

During a single turn, the player has to make a move using one of his pieces. Making a move is done by choosing the piece and changing its position according to the specified movement rule for this piece. At any time, at most one piece can occupy a square, so finishing the move on a square containing a piece (regardless of the owner) results in removing it (capturing). No piece addition is possible. After performing a move, the player gives control to the opponent.

For a given piece, the set of its legal moves is defined as 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. A positive Δy means forward for the moving player.

Consider a piece and a word $w \in \Sigma^*$ that belongs to the language described by the regular expression in the movement rule for this piece. Let $w = a_1 a_2 \dots a_k$, where each $a_i = (\Delta x_i, \Delta y_i, on_i)$, and suppose that the piece stands on a square $\langle x, y \rangle$. Then, w describes a move of the piece, which is applicable in the current board position 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 $\langle x + \sum_{j=1}^{i} \Delta x_j, y + \sum_{j=1}^{i} \Delta y_j \rangle$. The move of w changes the position of the piece piece from $\langle x, y \rangle$ to $\langle x + \sum_{i=1}^{k} \Delta x_i, y + \sum_{i=1}^{k} \Delta y_i \rangle$.

An example of how the move rules work is shown in Figure 1. A partial codification of simplified version of chess is presented in Figure 2.

2.2 Evolving game rules

In our work, we are using the data obtained from the two experiments concerning generation of the fairy chess games belonging to the Simplified Boardgames class.

The evolutionary system described in [16] uses an adaptive genetic algorithm with the fitness function based on the simulated playouts to generate playable and balanced fairy chess games. The generator is not too restrictive and allows e.g. asymmetric initial position and terminal conditions. A hand-made evaluation function analyzes the playout histories and checks for e.g. balance, game tree size, pieces importance, and, to some extent, complexity of the game rules.

Another approach, described in [17], uses Simplified Boardgames as an exemplary domain to extend and formalize the idea of Relative Algorithm Performance Profiles (RAPP) [20]. The games it generates are more constrained and chess-like: the system always produces fully symmetrical games with one royal piece, and an initial row of pawn-like pieces. The evaluation function uses a number of algorithms (player profiles) with various degree of intelligence. To assess the strategic properties of some generated game, it runs those algorithms against each other and compare the results with the results obtained on human-made

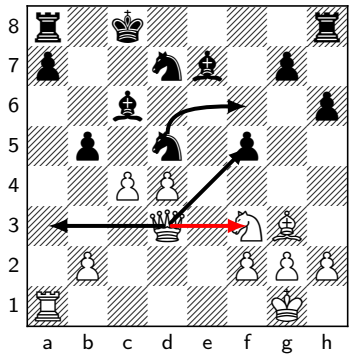

Fig. 1. A chess example. Two legal moves for the queen on $d3$ are shown. The capture to $f5$ is codified by the word $(1,1,e)(1,1,p)$, while move to $a3$ is encoded by $(-1,0,e)(-1,0,e)(-1,0,e)$. The move to $f3$ is illegal, as in the language of queen's moves no move can end on a square containing own's piece. The $d5-f6$ knight move is a direct jump codified by the one-letter word $(2,1,e)$.

```
<PIECES> // P - pawn, R - rook, N - knight
P (0,1,e) + (-1,1,p) + (1,1,p)  &
R (0,1,e)(0,1,e)^*+(0,1,e)^*(0,1,p)+(0,-1,e)(0,-1,e)^*+(0,-1,e)^*(0,-1,p)+
  (1,0,e)(1,0,e)^*+(1,0,e)^*(1,0,p)+(-1,0,e)(-1,0,e)^*+(-1,0,e)^*(-1,0,p)&
N (2,1,e) + (2,-1,e) + ... +(-1,-2,e) + ... + (-1,-2,p) &
```

Fig. 2. The part of the piece definition section of chess as a Simplified Boardgame. Moves are represented by the regular expressions (`^*` being the Kleene star operator).

chess-like games. Based on the RAPP assumption, we expect that all games that behave similarly to the high quality human-made games will also be good. The results indeed show that in this way we can obtain playable and balanced games of high quality, but not necessary with the rules which will be intuitive and easy to learn by human players.

3 Evaluation Function

Given a set of games assessed for their strategic properties, we would like to reevaluate them taking into account learnability of their rules. By learnability we mean the ease to understand and remember how a given game works. As in Simplified Boardgames terminal conditions are deliberately kept simple, we will focus on evaluating piece rules. We use the theory of fairy chess movements [14] to construct a natural language description based on the analogies with the well-known chess pieces and other easy-to-explain chunks of rules. An additional benefit of such an approach is that the generated descriptions can be directly presented to the user in some kind of *How to Play* guide.

3.1 Fairy chess theory of movements

The term *fairy chess* is used to describe the unorthodox chess variants common in chess problems, or, more broadly, the generalization of the chess-like games. T.R. Dawson's *Theory of Movements* formed the basis for the fairy chess movement types, dividing them into: *leap*, *ride*, and *hop* [14]. The Metagame system is entirely restricted to generate pieces accordingly to this rules [21]. The Simplified Boardgames class is more general, nevertheless – to provide human-like descriptions – we will refer to this theory.

Let us describe these three types of movements in detail. The *leap* takes the piece from one square directly to the other (given the direction vector), with no regard for intervening squares. For example, the chess knight movement rules consist of $\langle 1,2\rangle$-leap, $\langle 2,1\rangle$-leap, $\langle -1,2\rangle$-leap, and so on. We say that a piece is $\langle m,n\rangle$-leaper if it leaps in all possible directions by vectors $\langle \pm m,\pm n\rangle$ and $\langle \pm n,\pm m\rangle$. So the knight is a $\langle 2,1\rangle$-leaper, and king is the $\langle 1,0\rangle$ and $\langle 1,1\rangle$-leaper. Implicitly, leaping to a square means capturing or moving to an empty square.

Next, the *rider* can move an unlimited distance in a given direction, provided there are no pieces in the way, and finish move on an empty square or capture an opponent piece. It can be seen as an extension of a leaper piece. Examples of this kind of pieces are rook ($\langle 1,0\rangle$-rider), bishop ($\langle 1,1\rangle$-rider) and nightrider ($\langle 1,2\rangle$-rider). Again, if no additional restriction are put, we assume riders to move in all possible directions.

Finally, the *hopper* is a piece that has to jump over some other piece. Usually it has to jump over the exactly one piece of any color. There are multiple subtypes of hoppers, so they have multiple more formal definitions. There are no hoppers in the standard chess, while in Chinese chess there is a cannon, which captures like the hopper over the rook lines, and moves like the rook when not capturing.

3.2 Generating descriptions

Our piece decomposition algorithm describes a given piece as a combination of simpler movement patterns. At the beginning, the regular expression determining a piece is unfolded into the sum of individual movements. The algorithm uses a greedy approach to describe all movements starting from the most promising partial descriptions.

The procedure distinguishes *classes* and *operators*. There are two classes: the *leaper* and the *rider* (the latter includes hoppers), each with an associated vector $\langle x,y\rangle$. Operators are used to describe proper subsets of moves. A description of a set of movements consists of a class paired with a sets of operators. For example, a $\langle 1,1\rangle$-*rider* may be restricted in this way by a single operator *forwards*.

The initialization phase divides every move into parts based on the common vector (so e.g., $(0,1,e)^*(0,1,p)(0,2,e)$ consists of two parts). Similar types of vectors are joined in descriptions; for example, $(1,1,e)^*(0,2,p)+(-1,1,e)^*(0,2,p)$ results with the description "rides diagonally forward and then captures outwards 2 vertically", which corresponds to the singly-bent riders nomenclature (see [13] for the details).

In general, for every undescribed yet move m (of the considered piece) , our algorithm tries to generate its description by using Algorithm 1. It also tries to include in the same description other moves not described yet, whenever they fit. Therefore the procedure Algorithm 1 takes two parameters: a move m, and the list of the undescribed moves M. If this fails (which happens for moves with more than three parts), a generic procedure is called. It always succeeds by describing any move as a list of preconditions followed by a destination step. After all piece moves have been described, the repairing run is launched. It joins descriptions made by the complementing operators and applies some grammar-fixing rules to make descriptions sound more naturally.

Let us focus on the main part of the procedure presented in Algorithm 1. Given the move m and the list of all undescribed moves M, as the candidates to a common description we leave only those moves in M that consists of the same number of parts as m (lines 1–4).

We iterate over the piece classes based on the i-th part vector of the move m, and the predefined lists containing combination of at most 4 operators (lines 8–33). Lists of operators are sorted based on their heuristic score, so checking is performed in a greedy way. Thus the obtained solution can be suboptimal, but the speed gain is significant comparing to the usage of e.g. the set cover algorithm. For the performance purposes, the operators lists are filtered to exclude contradicting cases (e.g. *forwards* cannot be combined with *backwards*).

For a chosen class and operators, we compute the set of moves this pair appoints and its natural language description (line 10). Then, we try to build a new prefix map H which extends $movesByPrefix$ on the i-th part. This may happen only if the subset of moves we try to describe is fully included for every prefix computed so far (lines 11–19).

Then we have to check if H truly describes our particular move m (lines 20–22). If there are more parts to describe, we recursively call function for the next part. If this succeeds we can merge our descriptions (lines 23–30).

3.3 Evaluating descriptions

Our NLD evaluation function f scores a piece given its partition into a sum of classes and operators. Let us start defining f with the formula for scoring a single translating vector $\langle x, y\rangle$. The unintuitiveness of a vector is measured as the sum of the maximum of the absolute values of the coordinates and the distance to the closest orthogonal or diagonal line. (Thus, $\langle 2, 0\rangle$ is easier to see and perform on board than $\langle 3, 0\rangle$ or $\langle 2, 1\rangle$).

$$f(\langle x, y\rangle) = 1 + \max(|x|, |y|) + \min(dist_{+}(x, y), dist_{\times}(x, y)). \tag{1}$$

Further, we define f for each of the possible operators o. The values for the operators have been tested manually. Some of the operators are presented in Table 1.

Each part of a piece is a class combined with a list of operators. The cost of that component depends on the vector $\langle x, y\rangle$ associated with the class and the costs of the operators:

Algorithm 1 Generating the description of a move m, given a list of all undescribed moves M.

```
1: function findDescription(M : [Σ*], m : Σ*)
2:     moves ← Moves from M with the same number of parts as m
3:     movesByPrefix ← {(ϵ, moves)} ▷ Map from prefixes (parts) to lists of moves
4:     return findDescription(m, movesByPrefix, 0)
5: end function
6:
7: function findDescription(m : Σ*, movesByPrefix : Σ* → [Σ*], i : ℕ)
8:     for operators in SortedOperatorSets do
9:         for class in PieceClasses do
10:            (moves, description) ← getMoves(class, operators)
11:            H ← {}                                   ▷ Creates empty map
12:            for (prefix, M) in movesByPrefix do
13:                if moves ⊆ getParts(M, i) then
14:                    G ←Moves from M whose i-th element is in moves
15:                    H.updateWith(G)
16:                else                    ▷ Wrong choice of operators and class
17:                    break and continue in line 8
18:                end if
19:            end for
20:            if m is not in some values of H then
21:                continue
22:            end if
23:            if i <numberOfParts(m) then
24:                r ← findDescription(m, H, i+1) ▷ Recursive call for the next part
25:                if r is Fail then
26:                    continue
27:                else
28:                    return description "and then" r  ▷ Combining part descriptions
29:                end if
30:            end if
31:            return description
32:        end for
33:    end for
34:    return Fail
35: end function
```

Table 1. Evaluation costs of some operators.

$f(o)$	o
0	none
1	backwards, forwards
2	sideways, only capture, without capture, outwards max times, min times, not horizontal
3	exactly times, over own piece instead
5	only odd, only even

$$f(\langle x, y\rangle, [o]) = f(\langle x, y\rangle) \cdot (1 + \sum_i f(o_i)) \tag{2}$$

If some moves of a piece contain k parts, their score is the product of scores computed via equation 2. In the case that no class-based formula can be found, the generic procedure describes the move as a list of its transition vectors, and scores it as a product of individual vectors' scores using equation 1. When all piece movements are resolved and evaluated, the score of a piece is the sum of the function f applied to all components it consists of.

Finally, to score the entire game, we require the list $[p]$ of pieces, and the function # counting occurrences of a piece in the initial position. Let k be the number of defined types of pieces. Then, the NLD game score is calculated as:

$$f([p]) = (\sum_{i=1}^{k} \#p_i \cdot f(p_i))/(\sum_{i=1}^{k} \#p_i) \cdot (10 + k). \tag{3}$$

The purpose of constant values in the above formulas is to smooth the differences between various possibilities. For example, the evaluation should benefit 4-pieces games in comparison to 5-pieces games as generally simpler, but the 4:5 relation has been found too severe. The values of the constants have been tested by analyzing the behavior of evaluation function over the predefined set of testing pieces and games.

4 Experiments and Results

In this section we describe some results of our experiments. First, we present descriptions and scores our system proposes for the existing fairy chess pieces of varying complexity. We also apply our NLD evaluation to the procedurally generated games from [17] and [16], and analyze the relationship between the strategic properties and the rules simplicity. In particular, we present an example of the high-quality, easy-to-learn RAPP-generated game.

4.1 Evaluating pieces

First, we would like to present the descriptions generated by our system for a few examples of the fairy chess pieces taken from *Piececlopedia* subpage of [22].

Piece complexities computed by our algorithm are put in parentheses.

- queen *(6)*: Rides in every direction
- short rook *(12)*: Rides max 4 times horizontally or vertically
- lance *(4)*: Rides vertically forward
- centaur *(14)*: Leaps 1 in every direction or over vector (1,2)/(2,1)
- griffon *(78)*: Moves 1 diagonally or moves 1 diagonally and then rides outwards vertically or horizontally
- hippogriff *(120)*: Moves 1 diagonally and then rides minimum 2 times outwards horizontally or vertically

– duke *(60)*: Rides without capturing diagonally and then leaps outwards 1 vertically or horizontally
– moa *(36)*: Moves 1 diagonally and then leaps outwards 1 vertically or horizontally
– ferz then wazir *(12)*: Moves 1 diagonally and then leaps 1 vertically or horizontally
– buffalo *(28)*: Leaps over vector (3,2)/(2,3) or (3,1)/(1,3) or (1,2)/(2,1)

We would like to point out that procedurally generated pieces are usually not so well-formed. They may contain various rules for movement, capture, and even self-capture, combined with artificial conditions both in the sense of the square content and the relative position. For example we may have a piece that "captures own forward over vector (5,2)/(2,5) or rides capturing but riding only over own pieces horizontally or vertically" (complexity 136).

4.2 Evaluating games

Our main goal is to evaluate entire games, especially procedurally generated ones. We tested the NLD measure on the sets of games produced by the RAPP evolver [17] and simple simulation-based (SIMB) evolver [16]. Also, we applied our evaluation to some human-made chess variants. As it is difficult to assign to the NLD values some standalone interpretation, their meaning for our purpose remains purely comparative.

First, let us present graphically the correlation between the complexity of rules and the strategic properties of the generated games, depending on the generator used. Figure 3 highlights the games belonging to the Pareto front given two evaluations as parameters. The RAPP chart is trimmed to NLB value of 3000, and shows 71% of all 2581 games. The Pareto front contains 8 games, including one game with the RAPP score 0.978 and the NLD score 78557 not shown on the chart. The SIMB chart is trimmed to NLB value of 2000, and shows 54% of all 8002 games. The Pareto front contains 17 games with the worst NLD score of 1983. Although the general plot shape is similar, this comparison reveals some differences in the simplicity of games generated by both approaches.

For comparison, Table 2 presents the NLD scores obtained by existing chess variants and previously published procedurally generated simplified boardgames. All game rules had been, if necessary, simplified to fit within the Simplified Boardgames framework.

An example of a generated and evaluated game is presented in Figure 4. It is the second highest NLD scored game in the Pareto front from the RAPP-generated set. It is also the third best game given RAPP evaluation.

Confirming conclusions from [17], strategic evaluation supports setups with multiple immobile pieces (especially pawns) and only a few mobile pieces with a real offensive power. This makes the games very positional and, as most of the moves cannot be reversed, requiring careful ahead planning.

Table 2. NLD Evaluation of example chess-like games.

Game	NLD evaluation
shatranj	166
chess	168
CWDA Colorbound Clobberers [23]	196
CWDA Remarkable Rookies [23]	222
CWDA Nutty Knights [23]	248
SIMB 30-P4 [16]	253
tamerlane chess [22]	376
RAPP The Legacy of Ibis [17]	387

5 Conclusion

We have presented a method for generating natural language descriptions for arbitrary fairy chess pieces given in the Simplified Boardgames language. The obtained descriptions explain the pieces in a manner similar to the descriptions provided by the domain experts [13], by referring to the human knowledge, intuition, and based on the theory of chess movements – which helps to quickly learn the rules of the game. The goal of creating these descriptions is to explain the rules of the procedurally generated chess-like games to the human players. The method can be generalized to include all features represented by other chess variants, which is the subject of further development.

The presented algorithm can be also used as the evaluation function for the game generation mechanism, complementing existing approaches that can produce games preserving high-quality strategic properties, but cannot guarantee that they will be intuitive and easy to learn.

We have tested our approach on some existing human-made chess-like games (including standard chess) and two sets of procedurally generated games. We have discussed correlation between our NLD evaluation and strategic properties for the generated games, and presented the example of a high-quality and easy-to-learn procedurally generated game.

References

1. Shaker, N., Togelius, J., Nelson, M.: Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Springer (2016)
2. Nelson, M.J., Mateas, M.: Towards Automated Game Design. In: AI* IA: Artificial Intelligence and Human-Oriented Computing. (2007) 626–637
3. Togelius, J., Nelson, M.J., Liapis, A.: Characteristics of Generatable Games. In: FDG Workshop on Procedural Content Generation. (2014)
4. Zook, A., Riedl, M.O.: Automatic Game Design via Mechanic Generation. In: AAAI Conference on Artificial Intelligence. (2014) 530–537

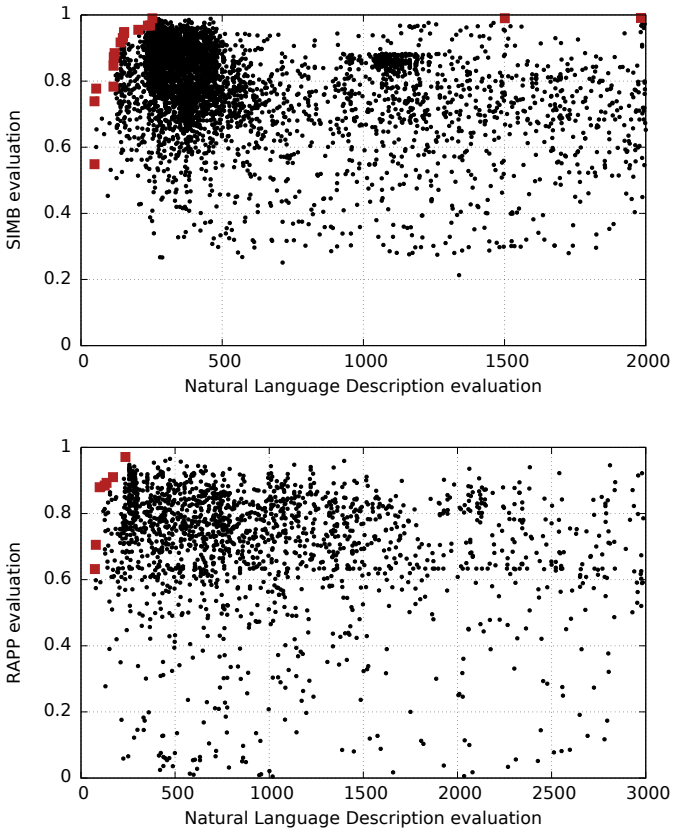

Fig. 3. Correlation between the NLD evaluation and the strategy properties evaluation: simple simulation-based [16] on the top, RAPP-based [17] on the bottom. Red nodes belong to the Pareto front. (The goal is to maximize the strategic evaluation and minimize the NLD evaluation.)

5. Pell, B.: METAGAME: A New Challenge for Games and Learning. In: Heuristic Programming in Artificial Intelligence: The Third Computer Olympiad. (1992)
6. Browne, C., Maire, F.: Evolutionary game design. IEEE Transactions on Computational Intelligence and AI in Games **2**(1) (2010) 1–16
7. Font, J.M., Mahlmann, T., Manrique, D., Togelius, J.: Towards the automatic generation of card games through grammar-guided genetic programming. In: International Conference on the Foundations of Digital Games. (2013) 360–363
8. Togelius, J., Schmidhuber, J.: An experiment in automatic game design. In: IEEE Symposium on Computational Intelligence and Games. (2008) 111–118
9. Cook, M., Colton, S.: Multi-faceted evolution of simple arcade games. In: IEEE Conference on Computational Intelligence and Games. (2011) 289–296
10. Perez, D., Samothrakis, S., Togelius, J., Schaul, T., Lucas, S.M.: General Video Game AI: Competition, Challenges and Opportunities. In: AAAI Conference on Artificial Intelligence. (2016) 4335–4337

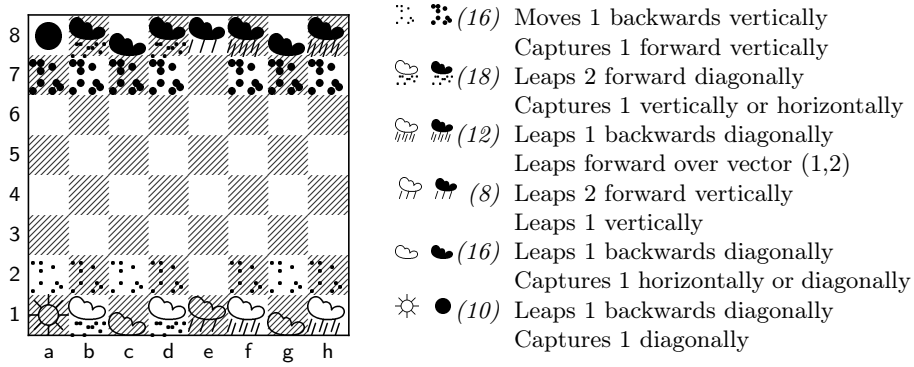

Fig. 4. Initial position and piece rules for the evolved game with RAPP score 0.971 and NLD score 236 (the numbers in parentheses are the estimated piece complexities). Winning conditions are to capturing the king (☼), reach the opponent's backrank with a pawn (⁛), or make the opponent unable to move. Turnlimit is 209.

11. Nielsen, T.S., Barros, G.A.B., Togelius, J., Nelson, M.J.: Towards generating arcade game rules with VGDL. In: IEEE Conference on Computational Intelligence and Games. (2015) 185–192
12. Genesereth, M., Thielscher, M.: General Game Playing. Morgan & Claypool (2014)
13. Wikipedia: Fairy chess piece — Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/wiki/Fairy_chess_piece` (Jan. 2017)
14. Dickins, A.: A Guide to Fairy Chess. Dover (1971)
15. Kowalski, J., Sutowicz, J., Szykuła, M.: Simplified Boardgames. arXiv:1606.02645 [cs.AI] (2016)
16. Kowalski, J., Szykuła, M.: Procedural Content Generation for GDL Descriptions of Simplified Boardgames. arXiv:1108.1494 [cs.AI] (2015)
17. Kowalski, J., Szykuła, M.: Evolving Chesslike Games Using Relative Algorithm Performance Profiles. In: Applications of Evolutionary Computation. Volume 9597 of LNCS. (2016) 574–589
18. Björnsson, Y.: Learning Rules of Simplified Boardgames by Observing. In: European Conference on Artificial Intelligence. Volume 242 of FAIA. (2012) 175–180
19. Kowalski, J., Kisielewicz, A.: Testing General Game Players Against a Simplified Boardgames Player Using Temporal-difference Learning. In: IEEE Congress on Evolutionary Computation. (2015) 1466–1473
20. Nielsen, T.S., Barros, G.A.B., Togelius, J., Nelson, M.J.: General Video Game Evaluation Using Relative Algorithm Performance Profiles. In: Applications of Evolutionary Computation. Volume 9028 of LNCS. (2015) 369–380
21. Pell, B.: METAGAME in Symmetric Chess-Like Games. In: Heuristic Programming in Artificial Intelligence: The Third Computer Olympiad. (1992)
22. Duniho, F.: The Chess Variant Pages. http://www.chessvariants.org/ (2016)
23. Wikipedia: Chess with different armies — Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/wiki/Chess_with_different_armies` (January 2017)