Efficient Reasoning in Regular Boardgames

Jakub Kowalski, Radosław Miernik, Maksymilian Mika, Wojciech Pawlik,
Jakub Sutowicz, Marek Szykuła, Andrzej Tkaczyk
Institute of Computer Science
University of Wrocław
Wrocław, Poland
jko@cs.uni.wroc.pl, radekmie@gmail.com, mika.maksymilian@gmail.com, pawlik.wj@gmail.com,
jakubsutowicz@gmail.com, msz@cs.uni.wroc.pl, andrzej.tkaczyk31@gmail.com

***Abstract*—We present the technical side of reasoning in Regular Boardgames (RBG) language – a universal General Game Playing (GGP) formalism for the class of finite deterministic games with perfect information, encoding rules in the form of regular expressions. RBG serves as a research tool that aims to aid in the development of generalized algorithms for knowledge inference, analysis, generation, learning, and playing games. In all these tasks, both generality and efficiency are important.**

In the first part, this paper describes optimizations used by the RBG compiler. The impact of these optimizations ranges from 1.7 to even 33-fold efficiency improvement when measuring the number of possible game playouts per second. Then, we perform an in-depth efficiency comparison with three other modern GGP systems (GDL, Ludii, Ai Ai). We also include our own highly optimized game-specific reasoners to provide a point of reference of the maximum speed. Our experiments show that RBG is currently the fastest among the abstract general game playing languages, and its efficiency can be competitive to common interface-based systems that rely on handcrafted game-specific implementations. Finally, we discuss some issues and methodology of computing benchmarks like this.

***Index Terms*—General Game Playing, Game Description Languages, Regular Boardgames, Optimization, Benchmarks**

I. Introduction

The idea of a generalized game playing (GGP) program, the one with the ability to successfully play any given game even such that it has not seen before, may be seen as a direct descendant of the famous General Problem Solver created by Simon, Shaw, and Newell in 1959 [1]. Although the first published formalism starting a new domain of GGP research is a work from 1968 by Pitrat [2] concerning a generalization of chess-like games, which was followed in the 90s by Pell and his Metagame approach [3], the real attention towards the idea started in 2005 with the publication of Stanford's Game Description Language (GDL) and the announcement of the annual International General Game Playing Competition (IGGPC) co-located with AAAI conference [4], [5]. Since that time, for almost a decade, Stanford's GGP had been the leading field for developing generalized AI solutions, and a source of numerous advancements in search [6], [7], knowledge representation [8], [9], and other fields [10], [11]. In 2016, the last (so far) IGGPC was held, given the number of GDL-related publications was steadily decreasing, as researchers started shifting their attention to other topics. Today, however, we apparently experience a General Game Playing renaissance. In just a few years, several alternative languages and platforms had been released – by multiple author groups, featuring a variety of game types, based on diverse methodologies, and with different purposes under consideration.

These new GGP platforms are made by hobbyists (Ai Ai [12]), researchers (Regular Boardgames [13], Ludii [14]), or even big companies like Google DeepMind (OpenSpiel [15]) and Facebook (Polygames [16]). They range from defining a limited number of boardgames (GBG [17]), any turn-based games: perfect information deterministic (Regular Boardgames) / nondeterministic with imperfect information (Ludii), to Atari-like real-time games (ALE [18], GVGAI [19]). Their methods for describing game rules vary from using regular expressions and automata (Regular Boardgames), a simple objective scripting language (GVGAI), high-level keywords (Ludii), or using underlying game-specific implementations in, e.g., Java (Ai Ai, GBG) / C++ (OpenSpiel, Polygames). Some are aiming for efficiency, self-containment, and generality under a uniform mechanism (Regular Boardgames), other for human-user game-playing experience (Ai Ai), a study on structure, history, and modeling of games (Ludii), or support for generalized reinforcement learning (OpenSpiel and Polygames).

In this work, we present the technical side of reasoning in Regular Boardgames (RBG) language – a universal GGP formalism for the class of finite deterministic games with perfect information, encoding rules in the form of regular expressions. RBG serves as a research tool that aims in development of general algorithms for games, which includes knowledge inference and game analysis, learning, and playing algorithms. In all these tasks, both generality and efficiency are important. Generality is necessary to avoid solutions designed only for specific game types, which have no chances to work on a new, and previously unpredicted problem instances. Computational efficiency makes every task more feasible, allowing e.g., a more detailed analysis of the search tree – which increases the playing strength of an AI agent.

RBG tries to achieve both goals. We explain how it reaches a high level of performance, competitive even with some manually implemented reasoners, while still describing games

This work was supported by the National Science Centre, Poland under project number 2017/25/B/ST6/01920.

978-1-7281-4533-4/20/$31.00 ©2020 IEEE

completely in a general abstract form. We present the insights of the RBG compiler and its optimizations.

Then, we perform an in-depth efficiency comparison with other popular and currently developed GGP systems. Additionally, we include in the comparison our own highly optimized game-specific reasoners of a few games under RBG interface. The results from the benchmarks can be used as a point of reference for both implementing a reasoner for a given game and developing new general game playing systems. In the former, one can compare the efficiency of a game implementation against various levels of optimization. As for the latter usage, any GGP approach to be practical requires some amount of fast reasoning. This efficiency survey shows where such a system fits regarding its computational capabilities, and on what types of games it behaves better or worse. It also points out the set of games that is good to implement when aiming to compare with other GGP systems. Finally, we discuss issues and methodology of producing such benchmarks, such as the impact of altering the formal game rules for different variants.

II. Regular Boardgames

We briefly describe the main concepts of Regular Boardgames. For the full formal definition, we refer to [13].

A game embedding in RBG consists of a *board*, *variables*, *player roles*, and *rules*. The *game state* contains a configuration of pieces on the board, values of the variables, the current player, the current position on the board, and the current index (position) in the rules. The *board* is a directed graph with labeled edges, called *directions*. The current player, in his turn, can perform a sequence of elementary *actions*, which, when applied sequentially, can modify the game state in a specific way. For an action to be *legal*, it must be both *valid* for the current game state when it is applied and also permitted by the rules. There are seven types of actions:

1) *Shift*, e.g., `left` or `up`, which changes the current position on the board following the specified direction. When there is no such edge, the action is invalid.
2) *On*, e.g., `{whiteQueen}`, which does not modify the game state but checks if the specified piece is on the board at the current position.
3) *Off*, e.g., `[whiteQueen]`, which puts the specified piece at the current position on the board. It is always valid.
4) *Comparison*, e.g., `{$ turn==100}`, which compares two arithmetic expressions involving variables.
5) *Assignment*, e.g., `[$ turn=turn+1]`, which assigns to a variable the value of an arithmetic expression.
6) *Switch*, e.g., `->white`, which changes the current player to the specified one. This action ends a move.
7) *Pattern*, e.g., `{? left up}`, which is valid only if there exists a legal sequence of actions under the specified rules; in the example, if from the current square there is a path with two edges labeled by `left` and `up`.

A sequence of actions ending with a switch defines a *move*.

Example 1: In Amazons, the following sequence of actions defines a move with a (white) queen moving two squares up and then shooting an arrow one square right.

```
{wQueen} [empty] up up [wQueen] right [arrow] ->black
```

Technically, a *move* is the subsequence of (indexed) actions that are offs, assignments, and switches, together with the positions in rules regular expression where they are applied. These are precisely the actions that modify the game state, except the board position and the rules index. Hence, the above example defines a move of length 4.

A playout ends when the current player has no legal moves. Then, each player's *score* is given in a dedicated variable, named the same as the player's role. The rules are given by a regular expression over the alphabet of the above actions. The language defined by this expression contains all potentially legal sequences of actions. For a concise encoding of the regular expression, a description in RBG is described through C-like macros that are instantiated for given parameters.

A. Example

A complete example of game Amazons is given in Fig. 1. Its underlying nondeterministic finite automaton, processed by the game compiler, is shown in Fig.2.

At the beginning of the description (Fig. 1, lines 1–14) we define the players (and their maximal achievable scores), pieces, variables (note that variables for players, containing their current scores, are created automatically), and the board graph with its initial state – in this case a rectangular board with four possible movement directions. Then we define some helpful, game-dependent macros. `anySquare` can change the current position to any square on board, by first jumping an arbitrary number of squares vertically, and then horizontally. `directedShift` allows movement in direction `dir` (given as a parameter) as long as the encountered squares are empty (they contain piece `e`), but at least one step has to be made. `queenShift` encodes all possible queen-like moves as a sum of directed shifts. Note that we can pass any sequence of tokens as a macro argument (in this case, two consecutive directions that allow us to encode diagonal movements).

The main game logic, the `turn` macro (lines 23–28), encodes a single turn for player `me`, whose queen pieces are encoded as `piece`. It starts by switching the player to ourselves, then switching the current square to any that contain our queen. We pick up the queen making this square empty, move it to the desired square, and put down. Then we find another square that will be the destination for an arrow. The `->>` gives control to the game manager (special role named *keeper*), as the player has no more decisions to make. The remaining steps put down the arrow (the `x` symbol) and set the winning score for the last player. If the playout ends because the current player has no legal moves, this stage will not be reached and the previous player will win the game. Finally, the overall rules of the game are encoded as a repetition of the sequence of the white and the black player turns (line 29).

III. Optimizations in RBG

The core of the RBG infrastructure is the compiler, which, given an RBG game description as the input, generates a C++ module implementing a reasoner for this game. As in every

```
#players = white(100), black(100)
#pieces = e, w, b, x
#variables = // no variables
#board = rectangle(up,down,left,right,
         [e, e, e, b, e, e, b, e, e, e]
         [e, e, e, e, e, e, e, e, e, e]
         [e, e, e, e, e, e, e, e, e, e]
         [b, e, e, e, e, e, e, e, e, b]
         [e, e, e, e, e, e, e, e, e, e]
         [e, e, e, e, e, e, e, e, e, e]
         [w, e, e, e, e, e, e, e, e, w]
         [e, e, e, e, e, e, e, e, e, e]
         [e, e, e, e, e, e, e, e, e, e]
         [e, e, e, w, e, e, w, e, e, e])
#anySquare = ((up* + down*)(left* + right*))
#directedShift(dir) = (dir {e} (dir {e})*)
#queenShift = (
    directedShift(up left) + directedShift(up) +
    directedShift(up right) + directedShift(left) +
    directedShift(right) + directedShift(down left) +
    directedShift(down) + directedShift(down right)
  )
#turn(piece; me; opp) = (
    ->me anySquare {piece} [e]
    queenShift [piece]
    queenShift
    ->> [x] [$ me=100, opp=0]
  )
#rules = (turn(w; white; black) turn(b; black; white))*
```

Fig. 1. RBG encoding of Amazons (orthodox version, non-splitted).

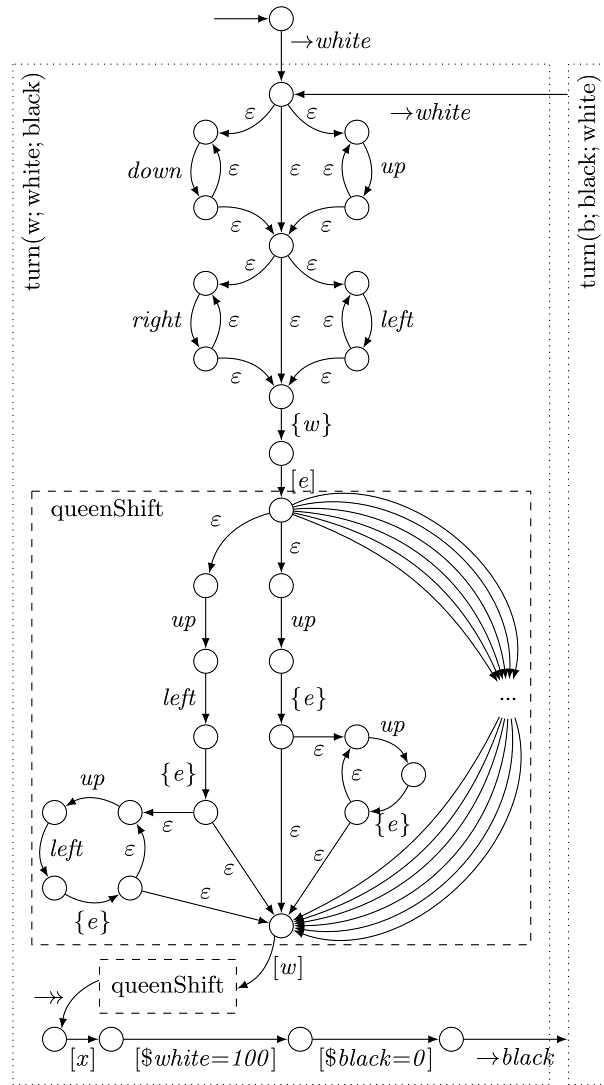

Fig. 2. NFA represented by the Amazons description from Fig.1.

GGP system, the reasoner satisfies a common interface, which, in the case of RBG, allows computing legal moves, reading parameters, accessing the board, etc.

A fundamental part of the reasoner is computing a list of all legal moves. This is done through a DFS-based algorithm [13, Theorem 9] on the automaton that is the NFA representing the game rules joint with the board graph. A straightforward implementation of the algorithm already provides a decent level of efficiency, but, through a prior analysis of the game rules, we were able to improve it significantly. We describe here a few of the most important optimizations, which are obtained by inferring knowledge from the game description. Table I shows the results of four techniques.

1) Shift tables: Very often, traversing the board consist of multiple shift actions, representing even complex behavior. For instance, on a rectangular board, `(left* + right*)(up* + down*)` allows changing the current square into any square, and `up* {! up}` changes the square to that in the top row but in the same column. Obviously, the number of possibilities from such sequences is limited. Each sequence of actions consisting only of shifts and possibly patterns with shifts can be represented by a map that, for each given square, stores a subset of allowed destination squares. Hence, we replace each such sequence with one elementary *shift table* action, which simply enumerates all the possibilities with non-deterministic transitions. Additionally, we can generate further simplifications if the shift table is deterministic or does not depend on the current square.

Shift tables as a whole are the most important optimization, which significantly affects every game.

2) Visited check skipping: The basic reasoning algorithm requires that we check whether a pair of the current square and the index in the rules has been already visited. Consider for instance `((NW + NE + E + SE + SW + W) {x})* {! NW}`, which checks whether from the current square there is a path on squares with `x` to the north-west line (example from Hex). Obviously, by applying actions we could return to the same square and the same position in the rules. However, in many typical cases, this is not possible. We can detect these cases by analyzing the transitions in the joint automaton, and omit checking visited pairs.

3) Bounding move length: Because of the *straightness* condition that RBG description must satisfy [13], guaranteeing that the game is finite and the number of legal moves if always finite, the moves have a bounded maximal length measured in the number of modifiers. For instance, in breakthrough, each move has length 2, which corresponds to the selection of the initial (picking up the pawn) and the destination squares. In chess, the maximum move length is 7. We can use the limit directly to define the move type structure having exactly this

optimal size, also avoiding any dynamic memory allocations.

We can easily calculate this limit if the joint automaton does not contain cycles containing a modifier and not containing a switch. For such games, we have a usually small upper bound on the length of every move. In other case (an example is the draughts family of games), such cycles could potentially generate infinite moves, thus the straightness condition must be satisfied non-trivially, and calculating the limit in general is a PSPACE-hard problem (cf. [13, Theorem 10]).

4) Monotonic classes of moves: Sometimes, especially in simple games, the bottleneck is the general interface itself. In the case of RBG compiler, all legal moves must be generated every time from scratch. Besides advantages like preserving minimal game state representation, informative move content (which contains a sequence of actions, in contrast to, e.g., a pure move index), fixed ordering, and modifiable moves list, it comes with an efficiency drawback in certain situations. In the case of simple games with many moves, the cost of generating them can be prevailing and thus get behind systems that admit, e.g., only modifying the list of legal moves.

We develop the general concept of *monotonicity classes*, which can deal with the above problem in several game types. We will split the game states into classes such that they share their legal moves. Let $\mathbb{S}$ be the set of all reachable game states from the initial state, and for $S \in \mathbb{S}$, let $\mathrm{moves}(S)$ be the set of all legal moves. Let $\mathbb{M} = \bigcup_{S \in \mathbb{S}} \mathrm{moves}(S)$ (the set of all possible moves). Now we define that a function $c \colon \mathbb{S} \to \mathbb{N}$ is *monotonic classifier* if for every state $S \in \mathbb{S}$, we have $\mathrm{moves}(S') \subseteq \mathrm{moves}(S)$ for every state $S' \in \mathbb{S}$ that is a successor of S in the game tree with $c(S) = c(S')$. There always exists a trivial monotonic classifier that assigns a different class number to each state. However, the smaller number of assigned classes is better. A game description is k-move-monotonic if there exists a monotonic classifier assigning at most k class numbers. Obviously, the existence of monotonic classifiers depends on the particular move representation.

Returning to RBG, a natural candidate to classify moves are switches. For each switch, we need to check whether for a game state with the rules index at this switch, the legal moves are a superset of the legal moves of every successor game state. We use several conditions for that, for example, if a move is related to the specific content of a square (e.g., empty) and, in the rules, this content is never added, then there will be no new moves in the successor game states. The monotonicity optimization also requires shift tables to detect if moves do not depend on the current square. Using that, we can determine that the game descriptions of, e.g., Connect4, Gomoku, and Hex (without the pie rule) are 2-monotonic. Also, in Pentago (split), we assign one monotonic class for the moves related to rotation (the eight rotation moves are invariant), despite that the placement moves cannot be assigned to the same class.

A. Efficiency Gain

Table I shows the impact of the described optimizations. We show their importance in the final version by the efficiency drop if an optimization is skipped. The effects of the optimizations strongly correlate. Monotonic classes optimization requires shift tables, thus there is no result for a variant with only shift tables skipped. Also, as described before, some optimizations (monotonic classes, bounding move length) provide a boost only for specific types of games, and are neutral for all the remaining ones; these cases are represented by 0%.

The most significant and universal optimization is shift tables. The second one, not much behind it, is visited checks optimization. They both affect every game. Bounding move length is a decent optimization and affects only games where moves have a bounded length by the rules and independently on the board, but this is actually a large class of games. Monotonic classes affect only simple games where moves are very straightforward, but these are usually the cases where listing all moves adds a significant computational cost.

Optimizations positively affect the computation time because they mostly reduce the amount of the generated C++ code, whose compilation is by far dominant. For example, Chess with optimizations is compiled in 7.2s and without them in 10.43s. In general, all the first three optimizations reduce the compilation time, and monotonic classes leave it unaffected. Except for monotonic classes, which require to store moves along with the game state, the optimizations do not have any real drawbacks, thus they should always be used.

IV. Comparison of Different Systems

A. Other GGP Approaches

Here we present in slightly more detail GGP systems that will be used in our experiments. We can describe those systems as belonging to three types of GGP approaches: "true" general game playing, where the description language is "closed" (e.g., GDL, Toss [20], Regular Boardgames); "hybrid", that describe games using an extendable set of generalized keywords (Metagame [21], Ludi, VGDL [22], Ludii); and one that just make use of a common interface for game-specific implementations (Ai Ai, GBG, OpenSpiel, Polygames). These categories are informal. Closed languages try to provide a uniform and minimal mechanism so that each new game can be effectively implemented purely in the proposed language. Hybrid languages try to provide high-level concepts that cover parts of game rules. As such, implementing a new game that requires a new rule type usually needs an extension of the language. The last type of systems just requires games to be manually implemented in a usual programming language and satisfying some interface. They also often provide parameterization of the rules. In theory, more game-specific code allows more optimization, thus should result in higher efficiency.

The GGP systems we have chosen for the comparison are the ones that are possibly very recent, currently developed, and containing enough games to find a common test set, with the exception of GDL, which is a classical example. Besides Regular Boardgames, there has been no recent approach to create a closed language for describing a large class of games.

We also performed experiments with OpenSpiel [15], however, given that this system during the playouts also computes

TABLE I
THE IMPACT OF SPECIFIC OPTIMIZATIONS OF THE RBG COMPILER (FLAT MC PLAYOUTS/SEC.).

Game	**No optimizations**	**No shift tables, no mon. classes**	**No visited check skipping**	**No bounding move length**	**No monotonic classes**	**All opt.**
Amazons	1,642 (-41%)	2,500 (-10%)	2,144 (-23%)	2,078 (-25%)	(0%)	2,781
Amazons (split2)	9,340 (-48%)	12,264 (-32%)	14,818 (-18%)	15,682 (-13%)	(0%)	18,084
Arimaa (split)	79 (-91%)	112 (-88%)	751 (-16%)	(0%)	(0%)	898
Breakthrough (8x8)	16,330 (-63%)	21,022 (-52%)	29,136 (-33%)	40,269 (-8%)	(0%)	43,575
Canadian Draughts	442 (-70%)	449 (-69%)	1,294 (-12%)	(0%)	(0%)	1,465
Chess (50-move rule)	249 (-73%)	370 (-60%)	656 (-30%)	854 (-9%)	(0%)	935
Connect4	271,240 (-66%)	351,767 (-56%)	604,614 (-25%)	765,700 (-5%)	485,451 (-40%)	804,326
English Draughts	14,052 (-75%)	14,327 (-75%)	30,593 (-46%)	(0%)	(0%)	56,269
Gomoku (standard)	3,455 (-97%)	5,377 (-95%)	81,801 (-28%)	95,561 (-16%)	7,101 (-94%)	113,718
Knightthrough	27,193 (-59%)	35,254 (-46%)	47,637 (-28%)	52,469 (-21%)	(0%)	65,823
Pentago (split)	16,854 (-63%)	20,782 (-54%)	43,207 (-5%)	44,993 (-1%)	42,942 (-6%)	45,445
Tic-tac-toe	767,315 (-57%)	962,030 (-46%)	1,550,360 (-13%)	1,575,951 (-11%)	1,374,291 (-23%)	1,777,036

observation tensors makes the comparison unfair. Thus, we decided not to include the results in this paper.

Apart from the other GGP systems that we described above, for some games we had developed game-specific reasoners (in C++) that implement the common RBG interface (currently, the part of it necessary for computing moves and states). This is an attempt to show possibly maximal reasoning efficiency. The implementations are highly optimized with a lot of low-level tricks designed for a single specific game.

1) Stanford's GDL: GDL [5], used in IGGPC, is the most well-known and deeply-researched game description language. It can describe any turn-based, simultaneous-moves, finite, and deterministic n-player game with perfect information. It is a high-level, strictly declarative language based on Datalog.

GDL does not provide any predefined functions, meaning that every predicate encoding the game structure must be defined explicitly from scratch. As a result, the game descriptions are usually long and hard to understand. Because their processing requires logic resolution, it is also very computationally expensive. In fact, many games expressible in GDL could not be played by any program at a decent level. Some games, due to computational cost, are not playable at all. For example, features like longest ride in checkers or capturing in go are difficult and inefficient to implement. In such cases, only simplified rules are encoded (yet often they are available in repositories under the standard name of the full game). GDL has a number of independent reasoner implementations, among which propnets [23] are considered the fastest.

2) Ludii: Although the Ludii system [14] (the successor of Ludi, used to generate first market-selling AI-authored boardgame [24]) has been designed primarily to chart the historical development of games and explore their role in human culture, its latest versions came out with additional tools for agent implementations, game visualizations and human playing [25]. The language is based on a large number of ludemes, conceptual units of game-related information, whose behavior is encoded in the underlying Java implementation. This makes the resulting games usually concise and well suited for tasks such as procedural content generation, but hard to understand without documentation of each ludeme, which is already very long and constantly growing. Another drawback is that a large but limited set of currently implemented ludemes greatly hampers natural expressiveness and efficiency of more complex and non-standard games that do not have dedicated highly specialized building-blocks. On the positive sides, Ludii comes with a large number of implemented games. The language allows generation of generalized game-related content such as human-playable GUI and various game/algorithm analyzing tools. High-level ludemes are also an easy source of heuristics, which Ludii agents can benefit, without the need to detect game features in a knowledge-free manner.

The efficiency is similar to that of a GDL propnet, sometimes overcoming the latter. Ludii is closed-source with one reference implementation provided, and due to the complication level, it is impractical to develop an independent branch.

3) Ai Ai: Stephen Tavener's Ai Ai [12] is a closed source program that allows playing abstract games versus both AI and human opponents, with user-friendly visualization, multiple options to customize, AI settings, and game analysis tools. It is an advanced platform containing many games, and more are being added all the time. Games can be hand-coded in Java (for efficiency), or assembled from large blocks using the MGL (Modular Game Language) – a scripting language based on JSON. In practice, almost all of the games are programmed directly in Java, so the resulting game engine is as fast as its underlying implementation is optimized. Thus, although it is considered as a general game playing approach, the reasoners are game-specific with a common interface.

B. Technical Setup

All experiments were performed on a single core of Intel(R) Core(TM) i7-4790 @3.60GHz of a computer with 16GB RAM. The GCC version was `gcc (Ubuntu 7.5.0-3ubuntu1 18.04) 7.5.0` with `boost 1.65.1.0`. The Java version was `Java(TM) SE Runtime Environment (build 13.0.2+8)`.

TABLE II
COMPARISON OF THE REASONING EFFICIENCY OF DIFFERENT GGP SYSTEMS. THE PERCENTAGE VALUES ARE RATIOS TO THE RBG COMPILER (FLAT MC PLAYOUTS/SEC.).

Game	GDL propnet	Ludii 0.9.3	Ai Ai 4.0.2.0	RBG compiler 1.2	RBG game-specific 1.2
Amazons	4 (0.1%)	–	–	2,781	–
Amazons (split2)	365 (2%)	2,634 (15%)	13,724 (76%)	18,084	–
Arimaa (split)	–	22 (2%)*	4,507 (501%)*	898	–
Breakthrough (8x8)	2,711 (6%)	2,344 (5%)	29,247 (67%)	43,575	157,333 (361%)
Canadian Draughts	–	156 (11%)*	–	1,465	–
Chess (50-move rule)	43 (5%)	88 (9%)*	248 (27%)*	935	–
Connect4	46,896 (6%)	38,544 (5%)	1,315,457 (164%)†	804,326†	2,139,403 (266%)†
Connect6 (split)	–	1,192 (3%)	21,725 (55%)	39,330	–
English Draughts	–	–	–	56,269	188,143 (334%)
English Draughts (split)	3,429 (6%)	2,830 (5%)*	84,751 (143%)*	59,335	231,252 (390%)
Gomoku (standard)	1,147 (1%)	4,091 (4%)	47,332 (42%)	113,718	–
Hex (9x9)	476 (0.8%)	9,259 (16%)	95,113 (165%)	57,508	–
Knightthrough (8x8)	4,913 (7%)	2,987 (5%)	68,250 (104%)	65,822	–
Pentago (split)	6,408 (14%)	–	–	45,445	–
Reversi	370 (3%)	757 (5%)*	53,866 (387%)*	13,910	182,228 (1,310%)
Skirmish (100 turns)	239 (3%)	848 (11%)*	–	7,715	–
Yavalath	–	49,060 (8%)	204,484 (32%)	636,032	–

* – the implemented rules are different from the version in RBG (explained in Subsection V-A).
† – see the issues described in Subsection V-B.

Each test (one game) was a run of the flat MC algorithm, yielding statistics of the average score of uniformly random playouts for each legal move from the initial state of the game, or just a run of random playouts for a given time, depending on the system. The preprocessing time was not counted in any case. The GGP system versions were the available ones up-to-date on 4th June, 2020.

Each GDL propnet test constitutes of the average time of 10 runs lasting 10 minutes, not counting the preprocessing (averaging is a proper practice here, because of non-deterministic propnet construction). The used propnet implementation is by C. Sironi based on `ggp-base`, currently not available online, but some results were reported independently [23].

Each Ludii 0.9.3 test was performed via the command-line option `--time-playouts` with default settings and 1 minute measuring time.

Each Ai Ai 4.0.2.0 test was performed through the dedicated menu option `MC Playouts/Second (This Game)`, which measures over 100 seconds.

Each RBG 1.2 test lasted 1 minute and was performed via the shared benchmark script (`rbg2cpp/run_benchmark.sh`). A test of an RBG game-specific reasoner was also 1 minute long, and it was compiled with the same overlaying benchmark procedure used for RBG; the package is included in RBG 1.2 release. This version contains all optimizations described in Section III.

C. The Results

Table II shows the results of the main experiment. There is a large gap between systems with abstract languages (GDL, Ludii) and systems with game-specific implementations (Ai Ai). RBG achieves similar performance to the latter, although the values strongly vary depending on the game, which could be explained by various levels of effort put in optimizing a game-specific implementation. Our game-specific implementations are faster than almost everything else, showing that automatically generated RBG reasoners still have optimization potential, as the RBG interface is currently not a barrier.

V. IMPACT OF METHODOLOGY

In [26] we pointed out several issues concerning the methodology of the benchmarks in GGP; here, we discuss two technical ones that particularly affected our experiment.

A. Influence of the Rules Implementation

An important issue is game matching, which needs special care among different systems. By the *same games* we understand those that have isomorphic game trees. This includes win/draw/loss distinction in terminal states. We made an effort to match the games in RBG with the existing implementations in GDL. In the other systems, some games have encoded a different variation of the rules. Up to our knowledge, we marked all these cases in Table II with a star. These differences are relatively minor to provide a meaningful comparison, basing on our subjective opinion and some internal tests with game variations. A possible exception is Chess and Arimaa in Ai Ai; they implement, among others, the threefold repetition rule, which is costly. Also, Canadian Draughts in Ludii is a split version. Nevertheless, the results could differ slightly under an exact match. Example: English Draughts (split) in RBG and GDL ends in a draw after 20 moves without moving a man nor a capture. However, in Ludii, instead of that, there is an internal hard turn limit set independently on the rules. This is a minor difference, as ending a random playout in this

way is rare. As it is sadly not a standard for GGP systems to provide exact specification of the rules or reliable game statistics, most of such disparities are very hard to spot, thus they influence the fairness of the published benchmarks. Here, we would like to show some more detailed examples of how heavily the reasoning efficiency can be altered by modifying the game rules without changing their commonsense meaning.

Let us continue our example of Amazons. The orthodox version under the standard interpretation is that the player's single turn consists of moving a queen and shooting an arrow. Thus, the first player has 2176 possible moves, and the average branching factor is 374 for the first player and 299 for the second [27]. However, some implementations modify the rules so the player turn is split in two: firstly a queen movement is selected, and then an arrow shot from this queen. This interpretation operates on the game tree that is not isomorphic with the orthodox version, but it considerably reduces the branching factor thus computation time [28]. The rules in RBG encoding the described variant are shown in Fig. 3. Compared to the orthodox version, in RBG, this *split2* approach allows more than 6 times faster simulations (see Table III).

Although *split2* is, thanks to its straightforwardness, the most popular unorthodox variant, there are many other possible reinterpretations of the rules. Another example based on splitting player's move into two parts (*split2a* in Tab. III) chooses a queen and its movement direction in its first part, and shifts all the remaining operations to the second part. This version reaches similar efficiency as its predecessor. However, it is possible to create other variants that will be significantly faster. A variant named *split5* (see Fig. 3 for its rules) starts a new turn after nearly every atomic choice that guarantees the correctness of the remaining playout. This variant is over two times faster than *split2*. Table III shows the results for even more variants based on the same orthodox encoding of Amazons, visualizing the possible impact of the split-based trick we described. All mentioned Amazon variants are available in the RBG repository; they are obtained by a minor modification of the encoding of the orthodox version.

The split2 variant of amazons (difference code):

```
23  #turn(piece; me; opp) = (
24        ->me anySquare {piece} [e]
25       queenShift [piece]
26        ->me queenShift
27       ->> [x] [$ me=100, opp=0]
28    )
```

The split5 variant of amazons (difference code):

```
16  #directedShift(dir; me) = (dir {e} ->me (dir {e})*)
23  #turn(piece; me; opp) = (
24        ->me anySquare {piece} {? anyNeighbor {e}} ->> [e]
25        ->me queenShift(me) ->> [piece]
26        ->me queenShift(me) ->> [x] [$ me=100, opp=0]
27    )
```

Fig. 3. Two unorthodox variants of Amazons in RBG.

TABLE III
COMPARISON OF EFFICIENCY OF DIFFERENT VARIANTS OF AMAZONS (FLAT MC PLAYOUTS/SEC.).

Game	RBG 1.2	speedup
Amazons (orthodox)	2,781	100%
Amazons (split2)	18,084	650%
Amazons (split2a)	18,108	651%
Amazons (split3)	34,934	1,256%
Amazons (split5)	38,694	1,391%
Amazons (split5+)	38,004	1,367%

B. Random Generators and Benchmark Procedures

When doing many simulations, the overlaying interface becomes meaningful. We show this on a particular part that is the random generator used to draw moves uniformly in flat MC. This issue has never been raised before, but it is quite noticeable when the number of turns per second is large enough. In Table IV, we demonstrate the possible impact of the generator, which includes the random generator itself and an unbiased method for drawing an integer from a range. There is the standard method combining `std::uniform_int_distribution` with `std::mt19937` (used in the tests for Tables I–III), a reimplemented Java method from `java.util.Random`, and a modern unbiased drawing algorithm by Lemire [29] combined with a fast Mersenne Twister `boost::random::mt11213b`.

In our experiments, for RBG, we have used the default method, which is usually the slowest of the three but probably of the highest quality (based on the traditional measurement of the period). The propnet, Ai Ai, and most likely also Ludii, use the standard Java generator. Of course, there is a trade-off between the quality and the speed, and different systems use different methods. From our experience, the choice of reasonable generator does not influence the quality of agent nor change the statistics, but it impacts the cost of computing. The impact becomes higher when the reasoning itself is faster. In extreme cases, as Connect4, the cost of random move selection can be dominating.

The issue does not concern only random generators, but the whole benchmark procedure (with time measurements, gathering statistics, etc.). For instance, the flat MC algorithm in Ai Ai for Connect4 performs a much larger number of iterations per second (we got even 3,428,427) than the benchmark report, while other, more costly games reveal no noticeable difference.

Concluding, when reaching such a performance level, it is difficult to provide a reliable benchmark. Nevertheless, in such cases, we can expect that the cost of reasoning would be negligible compared to any accompanying computation, and then, the efficiency of reasoning loses its importance.

VI. CONCLUSION

Regular Boardgames is a modern general game playing system aiming for efficiency and describing games via an ab-

TABLE IV
THE IMPACT OF THE USED RANDOM GENERATOR (FLAT MC PLAYOUTS/SEC.).

Game	**RBG compiler**			**RBG game-specific**		
	Default method	Java method	Lemire's method	Default method	Java method	Lemire's method
Breakthrough	43,575	42,738	34,917	157,332	182,547	144,175
Connect4	804,326	1,052,897	1,075,988	2,139,403	4,230,855	4,965,320
English Draughts	56,269	58,615	59,044	188,143	249,169	251,900
English Draughts (split)	59,335	62,506	65,182	231,252	295,987	296,895
Reversi	13,910	13,961	14,140	182,228	213,313	219,012

stract, concise, and well-defined formal language. The shared environment currently consists, in particular, of the game compiler to C++, a network-based game manager, and a high-level API allowing writing AI in Python. In this paper, we have described a few optimizations of the RBG compiler, as one of the sources of its efficiency.

We performed extensive experiments comparing the efficiency of five modern general game playing systems. We conclude that RBG significantly outperforms systems based on other abstract languages and has comparable (with a high variation) performance to game-specific reasoners of other systems as Ai Ai. By comparing with our hand-made game-specific reasoners under the same interface, we demonstrated that there is still potential for optimization. This leads to the following research question: given game rules, how to automatically produce an optimal reasoner? Our implemented optimizations so far are just an infantile play around it.

The final issues discussed, so far overlooked, should help in developing standardized benchmark methods concerning reasoners, which would allow fair, reproducible, and transparent comparisons.

ACKNOWLEDGMENTS

We thank Stephen Tavener for helping and explaining how to perform the benchmark of his Ai Ai system. We also thank Chiara F. Sironi for sharing the GDL propnet code and for helping with using it. Finally, we thank anonymous reviewers for their valuable comments and important insights.

REFERENCES

[1] A. Newell, J. C. Shaw, and H. A. Simon, “Report on a general problem solving program,” in *IFIP congress*, vol. 256, 1959, p. 64.
[2] J. Pitrat, “Realization of a general game-playing program,” in *IFIP Congress*, 1968, pp. 1570–1574.
[3] B. Pell, “METAGAME: A New Challenge for Games and Learning,” in *Heuristic Programming in Artificial Intelligence: The Third Computer Olympiad.*, 1992.
[4] M. Genesereth, N. Love, and B. Pell, “General Game Playing: Overview of the AAAI Competition,” *AI Magazine*, vol. 26, pp. 62–72, 2005.
[5] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth, “General Game Playing: Game Description Language Specification,” Stanford Logic Group, Tech. Rep., 2006.
[6] H. Finnsson and Y. Björnsson, “Simulation-based Approach to General Game Playing,” in *AAAI*, vol. 8, 2008, pp. 259–264.
[7] ——, “Learning Simulation Control in General Game Playing Agents,” in *AAAI*, 2010, pp. 954–959.
[8] M. Thielscher, “A General Game Description Language for Incomplete Information Games,” in *AAAI*, 2010, pp. 994–999.
[9] J. Romero, A. Saffidine, and M. Thielscher, “Solving the Inferential Frame Problem in the General Game Description Language,” in *AAAI*, 2014, pp. 515–521.
[10] M. Thielscher, “General Game Playing in AI Research and Education,” in *KI 2011: Advances in Artificial Intelligence*, ser. LNCS, 2011, vol. 7006, pp. 26–37.
[11] M. Świechowski, H. Park, J. Mańdziuk, and K. Kim, “Recent Advances in General Game Playing,” *The Scientific World Journal*, 2015.
[12] S. Tavener. (2020) Ai Ai. [Online]. Available: http://mrraow.com/index.php/aiai-home/
[13] J. Kowalski, M. Mika, J. Sutowicz, and M. Szykuła, “Regular Boardgames,” in *AAAI*, 2019, pp. 1699–1706, Full version at https://arxiv.org/abs/1706.02462, Source code at https://github.com/marekesz/rbg/.
[14] E. Piette, D. J. Soemers, M. Stephenson, C. F. Sironi, M. H. M. Winands, and C. Browne, “Ludii – the ludemic general game system,” in *ECAI*, 2020, (to appear).
[15] M. Lanctot, E. Lockhart, J.-B. Lespiau, V. Zambaldi, S. Upadhyay, J. Pérolat, S. Srinivasan, F. Timbers, K. Tuyls, S. Omidshafiei, D. Hennes, D. Morrill, P. Muller, T. Ewalds, R. Faulkner, J. Kramár, B. D. Vylder, B. Saeta, J. Bradbury, D. Ding, S. Borgeaud, M. Lai, J. Schrittwieser, T. Anthony, E. Hughes, I. Danihelka, and J. Ryan-Davis, “OpenSpiel: A framework for reinforcement learning in games,” *CoRR*, vol. abs/1908.09453, 2019.
[16] Facebook AI Research. (2020) Polygames. [Online]. Available: https://github.com/facebookincubator/Polygames
[17] W. Konen, “General Board Game Playing for Education and Research in Generic AI Game Learning,” in *IEEE CCOG*, 2019, pp. 1–8.
[18] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, “The Arcade Learning Environment: An Evaluation Platform for General Agents,” *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.
[19] D. Perez-Liebana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, and S. M. Lucas, “General Video Game AI: A Multitrack Framework for Evaluating Agents, Games, and Content Generation Algorithms,” *IEEE Transactions on Games*, vol. 11, no. 3, pp. 195–214, 2019.
[20] L. Kaiser and L. Stafiniak, “First-Order Logic with Counting for General Game Playing,” in *AAAI*, 2011, pp. 791–796.
[21] B. Pell, “METAGAME in Symmetric Chess-Like Games,” in *Heuristic Programming in Artificial Intelligence: The Third Computer Olympiad.*, 1992.
[22] T. Schaul, “A video game description language for model-based or interactive learning,” in *IEEE CIG*, 2013, pp. 1–8.
[23] C. F. Sironi and M. H. M. Winands, “Optimizing Propositional Networks,” in *Computer Games*. Springer, 2017, pp. 133–151.
[24] C. Browne and F. Maire, “Evolutionary game design,” *IEEE TCIAIG*, vol. 2, no. 1, pp. 1–16, 2010.
[25] M. Stephenson, E. Piette, D. J. Soemers, and C. Browne, “Ludii as a competition platform,” in *2019 IEEE COG*, 2019, pp. 1–8.
[26] J. Kowalski and M. Szykuła, “Experimental Studies in General Game Playing: An Experience Report ,” in *AAAI 2020 Workshop on Reproducible AI*, 2020, https://arxiv.org/abs/2003.03410.
[27] P. Hensgens, “A Knowledge-based Approach of the Game of Amazons,” 2001, Master Thesis, Maastricht University.
[28] J. Kloetzer, H. Iida, and B. Bouzy, “The monte-carlo approach in amazons,” in *Computer Games Workshop*, pp. 185–192.
[29] D. Lemire, “Fast random integer generation in an interval,” *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 29, no. 1, pp. 1–12, 2019.