Efficient Reasoning in Regular Boardgames

Jakub Kowalski, Radosław Miernik,
Maksymilian Mika, Wojciech Pawlik,
Jakub Sutowicz, Marek Szykuła, Andrzej Tkaczyk

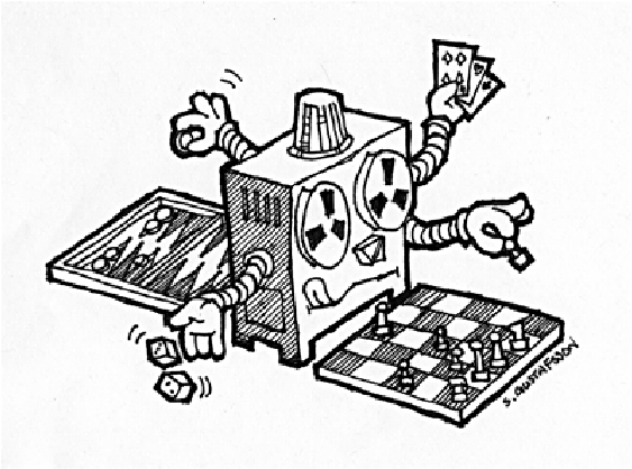
University of Wrocław, Poland

COG 2020

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

General Game Playing

General Game Playing (GGP)



General Game Playing renaissance

The old days

- Stanford's GGP leading the field since 2005
- Many valuable contributions (MCTS, knowledge representation, ...)
- Last Annual International General Game Playing Competition was organized in 2016

Recent years

- Several alternative languages and platforms had been released:
  - multiple author groups,
  - variety of game types,
  - diverse methodologies,
  - different purposes under consideration.

General Game Playing renaissance

- Authorship:
  - hobbyists (Ai Ai),
  - researchers (Regular Boardgames, Ludii, GBG),
  - big companies (OpenSpiel, Polygames).
- Expressiveness:
  - limited number of boardgames (GBG),
  - any turn-based games (Regular Boardgames, Ludii),
  - Atari-like real-time games (ALE, GVGAI).
- Methodology:
  - regular expressions and automata (Regular Boardgames),
  - objective scripting language (GVGAI),
  - high-level keywords (Ludii),
  - underlying game-specific implementations (Ai Ai, GBG, OpenSpiel, Polygames).
- Purpose:
  - efficiency and generality under a uniform mechanism (Regular Boardgames),
  - human-user game-playing experience (Ai Ai),
  - study on structure, history, and modeling of games (Ludii),
  - support for generalized reinforcement learning (OpenSpiel, Polygames).

This paper

Overview

- We present the technical side of reasoning in Regular Boardgames (RBG) language:
  - universal GGP formalism for the class of finite deterministic games with perfect information,
  - encoding rules in the form of regular expressions,
  - research tool that aims in development of general algorithms for games,
  - aiming for both both generality and efficiency at the same time.

Contributions

- Present the insights of the RBG compiler and its optimizations.
- Perform an in-depth efficiency comparison with other popular and currently developed GGP systems and highly optimized game-specific reasoners using RBG interface.
- Discuss issues and methodology of producing such benchmarks, so they can be used as a valid point of reference.

Regular Boardgames

Brief introduction to RBG

Basic notation

- 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.

Brief introduction to RBG

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`.

Brief introduction to RBG

Moves and playouts

- A sequence of actions ending with a switch defines a *move*.
  - `{wQueen} [empty] up up [wQueen] right [arrow] ->black`
- 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.
- A playout ends when the current player has no legal moves.
- Each player's *score* is given in a dedicated variable.

Game description

- 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.
- C-like macros for a concise encoding.

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

```
#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))*
```

Optimizations in RBG

RBG Infrastructure

RBG compiler

- Takes RBG game description as input.
- Generates a C++ module implementing a reasoner for this game.
- Reasoner satisfies a common interface for computing legal moves, reading parameters, accessing the board, etc.

Computing legal moves

- A fundamental part of the reasoner.
- DFS-based algorithm on the automaton that is the NFA representing the game rules joint with the board graph.
- Its straightforward implementation already provides a decent level of efficiency.

Optimizations

- Yet, through a prior analysis of the game rules, we were able to improve it significantly.
- In this work, we describe a few of the most important optimizations.

Shift tables

Complex shift-only behaviours

- Changing the current square into any square:
  `(left* + right*)(up* + down*)`
- Changes the square in the column to the top row:
  `up* {! up}`

Optimization

- Number of possibilities from such sequences is limited.
- They can be represented by maps that, for each given square, stores a subset of valid destinations.
- We replace those sequence with one elementary *shift table* action, which simply enumerates all the possibilities with non-deterministic transitions.
- Further simplifications if the shift table is deterministic or independent on the current square.
- Significantly affects every game.

Visited check skipping

Example (from Hex)

```
((NW + NE + E + SE + SW + W) {x})* {! NW}
```

checks whether from the current square there is a path on squares with `x` to the north-west line

Optimization

- Normally we need to check whether a pair of the current square and the index in the rules has been already visited.
- Because by applying actions we could return to the same square and position in the rules.
- But in many typical cases, this is not possible.
- Which can be detected (by analyzing the transitions in the joint automaton) and omitted.

Bounding move length

Straightness

- Valid RBG games have to be finite.
- The moves have a bounded maximal length measured in the number of modifiers.
- (Some theoretical analysis and algorithms in the original RBG paper)

- In breakthrough, each move has length 2.
- In chess, the maximum move length is 7.
- For such cases we may use data structure with this optimal size, also avoiding any dynamic memory allocations.

Calculating bounding move length

- Easy if the joint automaton does not contain cycles containing a modifier and not containing a switch.
- In other case (draughts family of games), they could potentially generate infinite moves.
- The limit exist but calculating it in general is a PSPACE-hard problem.

Monotonic classes of moves

- 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.
- For $S \in \mathbb{S}$, let $\mathrm{moves}(S)$ be the set of all legal moves.
- Let $\mathbb{M} = \bigcup_{S \in \mathbb{S}} \mathrm{moves}(S)$.

Monotonic classifier

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 $c(S) = c(S')$.

Monotonicity class

- A trivial monotonic classifier assigns a different class number to each state.
- The smaller number of classes is better.
- A game description is k-move-monotonic if there exists a monotonic classifier assigning at most k class numbers.

Monotonic classes of moves

Generating legal moves

- Especially in simple games, the bottleneck is the general interface itself.
- In RBG compiler, all legal moves must be generated every time from scratch.
- It provides many advantages, but sometimes causes an efficiency drawback.
- For simple games with many moves, generating them is costly.
- Modifying the list of legal moves would be faster.

Application

- In RBG, a natural candidate to classify moves are switches.
- For each switch-related game state we check if the legal moves are a superset of the legal moves of every successor game state.
- We use several conditions to compute this property.
- Requires shift tables to detect if moves do not depend on the current square.
- We can determine that e.g., Connect4, Gomoku, and Hex are 2-monotonic.
- In Pentago (split), we can assign one monotonic class for the moves related to rotation.

Efficiency Gain

General behavior

- The effects of the optimizations strongly correlate.
- Monotonic classes requires shift tables.
- Some optimizations (monotonic classes, bounding move length) provide a boost only for specific types of games.

Importance of optimizations

- The most significant and universal optimization is shift tables.
- The second on, is visited checks optimization.
- Bounding move length is a decent optimization actually affecting a large class of games.
- Monotonic classes affect only simple games – but in these cases listing all moves is computationally costly.

The impact of specific optimizations of the RBG compiler

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

Comparison of Different Systems

Other GGP Approaches: Informal classification

"true" GGP with "closed" description language

- Trying to provide a uniform and minimal formalism.
- So that each new game can be effectively implemented purely in the proposed language.
- *GDL, Toss, Regular Boardgames*

"hybrid"

- Try to provide high-level concepts that cover parts of game rules.
- Describe games using an extendable set of generalized keywords.
- A new game that requires a new rule type usually needs an extension of the language.
- *Metagame, Ludi, VGDL, Ludii*

Game-specific with interface

- Just make use of a common interface for game-specific implementations.
- Every game must be manually implemented in a usual programming language.
- *Ai Ai, GBG, OpenSpiel, Polygames, Zillion of Games*

Stanford's GDL

Language

- International General Game Playing Competition stared in 2005.
- The most well-known and deeply-researched GDL.
- Can describe any turn-based, simultaneous-moves, finite, and deterministic n-player game with perfect information.
- High-level, strictly declarative language based on Datalog.
- Does not provide any predefined functions.
- Game descriptions are usually long and hard to understand.

Reasoning

- Very computationally expensive – processing requires logic resolution.
- Many games expressible in GDL could not be played by any program at a decent level.
- Some games are not playable at all, sometimes simplified version are implemented.
- Fastest reasoners are based on propositional networks (propnets).

Ludii

Language

- Successor of Ludi, famous for its market-selling procedurally generated boardgames.
- Designed primarily to chart the historical development of games and explore their role in human culture.
- Additional tools for agent implementations, game visualizations and human playing.
- Based on a large number of ludemes, conceptual units of game-related information, encoded in the underlying Java implementation.
- Well suited for procedural content generation.
- Games are usually concise, but hard to understand without documentation of each ludeme.
- Large number of implemented games.
- Advanced GUI, human-playable and with game/algorithm analyzing tools.

Reasoning

- The efficiency is similar to the one of a GDL propnet.
- Closed-source with one reference implementation provided.

Ai Ai

Language

- Games hand-coded in Java (for efficiency), or assembled from large blocks using the Modular Game Language – a scripting language based on JSON.
- Still considered as a general game playing approach.
- User-friendly GUI, customizable AI settings and game analysis.

Reasoning

- Almost all of the games are programmed directly in Java.
- The reasoners are game-specific with a common interface.
- Very fast, using low-level game-specific optimizations.
- Closed-source.

Comparison of the reasoning efficiency of different GGP systems

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	–

For some games we had developed game-specific reasoners (in C++) that implement the common RBG interface

The Results

- 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 game-specific Ai Ai implementations.
- But results of Ai Ai vary depending on the game depending on an effort put in optimizing a game-specific implementation.
- Our game-specific implementations are faster than almost everything else.
- The RBG interface is not a barrier.
- Automatically generated RBG reasoners can still be significantly optimized.

Impact of Methodology

Influence of the Rules Implementation

Game matching for GGP

- A fair game matching is a common problem when comparing different GGP systems.
- By the *same games* we understand those that have isomorphic game trees, including win/draw/loss distinction in terminal states.
- Differences between game encodings can have from minimal to really huge impact on the obtained results.

Game matching in this paper

- Games in RBG are matched with the existing implementations in GDL.
- For other systems, some differences exist.
- Cases marked with the star are minor rule variations, so that the comparison remain meaningful.
- Chess and Arimaa in Ai Ai implement, among others, the threefold repetition rule, which is costly.

Influence of the Rules Implementation: Amazons Example

Orthodox version

- The player's single turn consists of moving a queen and shooting an arrow.
- First player moves: 2176.
- Average branching factor: 374 for the first player, 299 for the second.

Commonly used split

- The player turn is split in two: firstly a queen movement is selected, and then an arrow shot from this queen.
- The game tree that is not isomorphic with the orthodox version.
- It considerably reduces the branching factor and in the result the computation time.

Differences

- In RBG, split implementation is more than 6 times faster.

Comparison of efficiency of different variants of Amazons

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%

Even faster split variants

- It is possible to encode many more split variants.
- Some of them over two times faster than *split2*.

Random Generators and Benchmark Procedures

Choice of random generator

- We have found that the choice of a random number generator can substantially influence the reasoner's speed.
- This may occur when the number of simulations is huge enough.
- We have tested three methods, varying in speed and quality:
  - combining `std::uniform_int_distribution` with `std::mt19937`,
  - a reimplemented Java method from `java.util.Random`,
  - a modern unbiased drawing algorithm by D. Lemire combined with a fast Mersenne Twister `boost::random::mt11213b`.
- 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.

Benchmark procedure itself

- The whole benchmark procedure (time measurements, gathering statistics, etc.) can also significantly affect speed.

The impact of the used random generator

Game	RBG compiler		
	Default method	Java method	Lemire's method
Breakthrough	43,575	42,738	34,917
Connect4	804,326	1,052,897	1,075,988
English Draughts	56,269	58,615	59,044
English Draughts (split)	59,335	62,506	65,182
Reversi	13,910	13,961	14,140

Game	RBG game-specific		
	Default method	Java method	Lemire's method
Breakthrough	157,332	182,547	144,175
Connect4	2,139,403	4,230,855	4,965,320
English Draughts	188,143	249,169	251,900
English Draughts (split)	231,252	295,987	296,895
Reversi	182,228	213,313	219,012

Takeaway

- Comparing the efficiency in different game variants may result in huge skewing of the results.
- Comparisons in GGP domain are especially vulnerable.
- Sometimes what seems to be small difference may have large impact.
- In case of extremely fast computations, significant differences may be caused by the overlying interface.
- As well as by the underlying implementations of the common procedures.
- Some standardised benchmark methods for reasoners should be developed, to allow and ensure fair, reproducible, and transparent comparisons.

Conclusion

Regular Boardgames

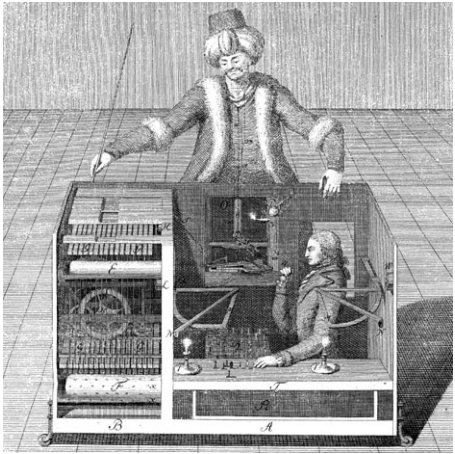
- A modern general game playing system.
- Open-source.
- Aiming for efficiency and knowledge-based analysis.
- Describing games via an abstract, concise, minimal, and well-defined formal language.
- With environment consisting of:
  - the game compiler to C++,
  - a network-based game manager,
  - a high-level API allowing writing AI in Python,
  - a database of games.

Reasoning efficiency

- We have described a few optimizations of the RBG compiler,
- performed extensive experiments comparing the efficiency of five modern GGP systems,
- discussed some issues, so far overlooked, regarding benchmark procedures.

Results

- RBG significantly outperforms systems based on other abstract languages,
- has a comparable (with a high variation) performance to game-specific reasoners of other systems as Ai Ai.
- There is still potential for optimization.



Thank you