Generating Chess Variants for Simulation on General Game Players

Ben Reis
Johns Hopkins University
3501 Saint Paul st.
Baltimore, MD 21218, USA
breis3@jhu.edu

Andrew Zhu
Johns Hopkins University
3700 N. Charles St.
Baltimore, MD 21218, USA
azhu8@jhu.edu

Abstract

General Game Playing (GGP) programs can learn to play any game it is given a description for. In our project, we aim to use these programs to help generate fun and balanced variants of chess. Our system generates descriptions of these variants using the Game Description Language (GDL). By analyzing gameplay results between pairs of GGP players, and by using genetic algorithms to build variants stochastically, our system is able to traverse the search space of chess variants to find one that is hopefully enjoyable.

1 Introduction

Strategy games are a popular passtime for humans, and also a topic of great interest to artificial intelligence researchers. AI researchers are mostly focused on the aspects of play that reflect intelligent decision making, but humans also have a great deal of interest in the fun of the game.

We have found little, if any, formal study of what sort of qualities separate strategy games that humans enjoy from those that they don't, although there are some simple heuristics that designers use. It is hard to test these heuristics, though, since there is no easy way to generate large numbers of games to test, assuming a researcher had a group of people willing to test them, and even then such a group of people is unlikely to contain a reasonable distribution of skilled and unskilled players.

Such players could be found in the output of AI researchers focused on general game playing, but these players require very carefully encoded games. These games are currently handwritten by researchers, usually encoding games that already exist.

Automated game design is a relatively new topic in artificial intelligence research.(1) Currently, researchers are focused on simple videogames that can serve as proofs-of-concept; creating complex strategy games is a substantially different task.

We focus on chess because it is a good model of a larger genre of positional strategy games, which are both interesting to humans and easily amenable to generic analysis, potentially by computers. Principles developed for creating and balancing variants of chess can then be generalized and expanded to cover a much wider field of games.

2 Encoding Chess-Like Games

Researchers in automated game design are presently focused on short, simple videogames with few rules. As far as we know, it has been many years since someone attempted to construct a system for generating chess-like games, and the only language we could discover for encoding such games is no longer being maintained.

On the analysis side of things, an extension of Datalog called the Game Description Language has become the standard language for formally specifying games in a manner that can be understood by generic computer players. As there are a number of existing players and masters that make use of GDL, we decided to use it as well.

As discussed in (1), Pell (1) did a great deal of work to define ”chess-like” games formally, in his Metagame workbench for general game players.

While we did not use the specification language he defined, his terminology and mathematical groundwork was quite useful. The key terms used here are described below.

Direction Vector A direction vector $\langle x, y \rangle$ is simply a vector over the chessboard. x represents a difference in file and y represents a difference in rank. A positive vector represents a move away from white and to white's right. For example, one of a knight's possible moves can be described by $\langle 1, 2 \rangle$, and a white pawn's standard move is in the direction $\langle 0, 1 \rangle$.

Symmetry Pell defines *forward*, *side*, and *rotation* symmetries for his vectors. *Forward* and *side* symmetries negate the y and x values of the vector, respectively, while the *rotation* symmetry swaps them.

Leap A leap describes a move where intervening squares on the board do not matter; only the destination square has any effect on the move's legality, and only the destination can be affected.

Ride A ride is a series of leaps in a particular direction, sometimes with a maximum number of repetitions. All squares between the start and the end of a ride must be empty. A pawn's first-turn move is a ride with length 2.

Hop A hop is a ride where some of the squares must be contain a piece, possibly of a specific type.

3 Search Algorithms

Genetic algorithms seemed like a good fit with the simulation based strategy that we picked: both genetic algorithms and the simulations involve randomness in their execution. Because our chess variants have different movements rules for individual pieces, we created an encoding of these movement rules. This encoding is then stochastically mutated and recombined with other encodings during the iterations of the algorithm.

We encode each chess variant to be a set of pieces. Because the game players are responsible for actually placing the pieces on the board (see the Implementation section), our encoding only deals with the movement rules of those pieces. Each piece in a variant has a list of movement rules (squares a piece can move into if not capturing a piece) and a list of capture rules (squares a piece can move into if capturing an enemy piece). Each of these rules consists of a direction vector, a ride length, and a list of symmetries that the rule is subject to.

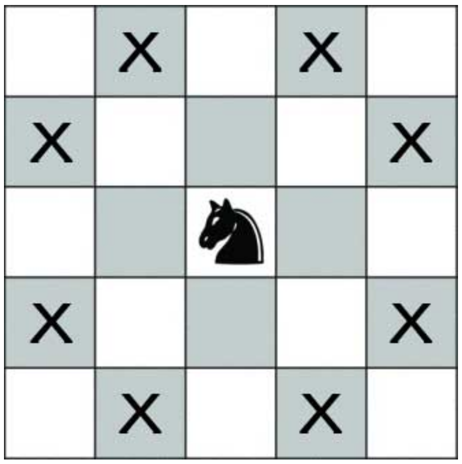

Figure 1: Knight Movement

Here are some ways that regular chess pieces would be encoded:

Knight (Figure 1)

Movement Rules:

1. direction:(1,2),
   rideLength:1,
   symmetries:[SIDE, FORWARD, DIAGONAL]

Capture Rules:

1. direction:(1,2),
   rideLength:1,
   symmetries:[SIDE, FORWARD, DIAGONAL]

Queen

Movement Rules:

1. direction:(1,0),
   rideLength:7,
   symmetries:[SIDE, DIAGONAL]

Capture Rules:

1. direction:(1,1),
   rideLength:7,
   symmetries:[SIDE, DIAGONAL]

Recall that diagonal symmetry is symmetry over the $y = x$ axis. This was renamed from Pell's rotational symmetry. This allows the knight movement rule to be encoded simply as one move with all 3 kinds of symmetry. Also recall that a ride is defined as consective movements where no intervening square is occupied, except for perhaps the last square on a capture move. The queen can traverse the entire board if not blocked, so she has a ride length of 7 to represent 7 consecutive movements in the direction (1,0) or (1,1). Both of the examples here have the same rules for capture and movement. A pawn would have two different rules in order to encode that a pawn can move forward if not blocked by another piece, and can only capture diagonally. Every piece in our encodings has side symmetry (across the y axis) for aesthetic reasons.

When running the algorithm, there are N surviving variants after a round of selection. During the generation phase, we generate an additional N candidate variants for the next round using mutation and recombination. A candidate is mutated by adding new pieces to its set of pieces, by deleting pieces, or by changing the movement rules of its existing pieces. Two candidates are recombined into a new candidate by picking randomly among the two parents' pieces to create a new set of pieces. The original N variants also remain in the pool of survivors, unchanged.

After the generation phase comes the selection phase, where candidates are evaluated using the GGP players, and then some candidates are thrown out before the next generation phase. In keeping with the spirit of games, we decided to use tournament selection to select N survivors from the $2N$ candidates. A fitness score is calculated for each candidate based on the results from the GGP players, and then candidates are randomly paired with each other. The candidate with the higher fitness score in each pair is selected to survive. Note that in this scheme, the candidate with the best fitness score will always survive, and candidates with higher scores are more likely to survive, but not guaranteed.

Fitness values are calculated based on statistics collected by the GGP players. To keep games fair, we would like the win rate to be about $1/2$ for each player, so the fitness value is calculated based on deviation from this win rate. We currently use only the win rate of white vs black to determine fitness, but we would like to use other metrics in the future to tailor our results. Incorporating the average length of games would help make sure that games aren't too short or too long. Pieces also should not be too complicated for human users to remember, so pieces with few movement rules should be preferred over those with long lists of rules.

4 Implementation

4.1 Generating Pieces

In creating our encoding, we decided to leave out more complex movements like the Hop from Pell's encoding, which requires a piece to jump over another piece in order to move. We decided that keeping the encoding simple would make translation from encoding to GDL easier, and that we could always add complexity to the encoding later.

Another consideration is the fact that encoding are not unique. Because of the symmetry parameters, there are different ways to encode the same basic direction. We resolved this by feeding the translator a revised version of the encoding with all duplicates removed, instead of enforcing extra conditions while creating the encoding, in order to again keep the GDL conversion simpler. This means that certain mutations that change the encoding of a candidate variant may not actually change the moveset or the GDL of that candidate.

4.2 Encoding the Game

Encoding chess in GDL presents a number of problems due to the complexity of the game's state, which involves a great deal of information on which pieces have moved at all (for pawn moves and castling) and which pieces might move to certain places (for check and related concepts). All of this is in addition to the 32 pieces that may be on the board and the 64 cells that make it up.

The work of Alex Landau in writing a slightly trimmed-down version of chess, as well as his more general work in GDL was invaluable for the code generation process. The fundamental arithmetic clauses we use are his directly (1), and many other elements of our code is based on his Speedchess (1) implementation of the game.

Other than a few counters and a phase marker, game state is tracked entirely by which squares contain pieces. Other cells are essentially ”brought into existence” as they are needed. This greatly reduces the ammount of state that players and the game master need to handle, but increases the size of the rules, as capture and non-capture moves must be handled separately.

To reduce the number of games which need to be evaluated, we have players place pieces themselves, purchasing them from a list that defines the game being played. In an extra phase before the game begins, players simultaneously place one piece each, as they choose, anywhere on a number of ranks associated with the variant they are playing. Each must place exactly one king, and the king must be placed last. There are no pawns unless the piece generation algorithm produces them. Once both players have placed their king, the game begins.

To simplify both the GDL code generation process and the resulting output, we have chosen to omit moves whose legality depends on facts other than the current state of the board. We also ignore the notion of check; players are instead required to capture the enemy king. Since GDL requires that games be finite in length, we force a draw after a number of moves, as is common in implementations of boardgames which can draw.

4.3 Running the players

We used the GGP server from GGP.org to host our games. This server sends data over a network using HTTP requests to communicate with the players. Our GGP Player was the award winning CadiaPlayer, which uses refined Monte Carlo tree search methods to make gameplay decisions. After the server is connected to its two players, it sends the game description and game state to each player, and gives the active player a limited amount of time (15 seconds) to respond with a valid move. Gameplay continues in this fashion until a terminal state is reached. Our program is capable of starting up multiple instances of the server and multiple players to speed up the running time.

5 Results

We were not able to run the code on chess variants because we were not able to work out all the bugs with generating the GDL code from an encoding. However, we were able to execute some benchmarks that give some idea about how the solver will perform if our system was fully functional. We also were able to test most parts of the system individually to make sure that they run according to the specification.

Before testing with chess variants, we tested the players with Tic-Tac-Toe, which is a very simple game with a decision tree that can be constructed very easily. As expected, the CadiaPlayer solvers are able to play this game optimally, and always ends the game in a draw when facing against itself. However, the CadiaPlayer still takes all the time it is given before making a decision. It is therefore critical to choose an appropriate amount of time for a given game in order to speed up computation so that gameplay doesn’t hang while the CadiaPlayer is already finished, or while it traversing very deep levels of the decision tree with no real hope of learning much more. The players were still able to play optimally when given only 3 seconds to respond with a move in Tic-Tac-Toe.

We tested the solver on harder games like connect four and chess. The solver was able to play connect four competitively, but was not very competent at chess, and seemed to make somewhat random movements. This is likely due to the large branching factor of chess, since there are so many moves to consider. Connect four is somewhat complex, but because there are only up to 8 possible moves to consider in any given state, so the decision tree is still manageable. We were aware of the large branching factor of chess from the start of our project, and we decided to stick with smaller boards with a few pieces on each side to lessen the problem.

We also extensively tested the genetic algorithm’s generation phase, making sure that it creates valid variants. We chose probabilities for each type of mutation (adding/deleting pieces, adding/deleting moves of pieces, changing movement vectors and symmetries) to make sure mutations were not too drastic.

6 Comparison to Proposal

We mainly stayed true to our original proposal while developing our project. The encoding was perhaps simpler in our implementation than in our original idea because we neglect the positioning of pieces, and leave that to the CadiaPlayer to decide best positions for each player. We did manage to include pawn like pieces, with capture rules that differ from their regular movement rules. We expanded on the implementation details greatly, bringing in the CadiaPlayer to actually simulate the GDL that we would generate. In the end, the program is built to do what was originally spelled out: take some board and some army conditions as input, and return a fun chess variant as output.

7 Future Work

7.1 Evaluating Games

It would be nice to figure out what makes a chess variant easier to learn for the CadiaPlayer. That would involve deeper analysis of the game states and analysis of the CadiaPlayer's internal states to figure out how the CadiaPlayer responds to changes between variants.

Other useful data could probably be found by running other players on the games generated, both to check what strategies do best in what games and to avoid making games that are in some sense only for Cadia.

7.2 Improving Performance

Almost all of our runtime is spent processing GDL through Cadia and the GGP server. Aside from bug fixes, the first obvious performance improvement to be made in this area would be the addition of *base* and *input* clauses to our generated code. These are optional (and somewhat nonstandard) clauses that many players use to speed up their analysis of games by never considering illegal moves or impossible game states.

7.3 Features Omitted

As discussed above, we omitted a number of features defined in (1), and features that might be considered ”typical” in chess variants.

Most obviously, our current generator does not (and cannot) produce the standard ”front rank” of pawns that appears in most versions of chess. Adding this in as an element that the generator could choose to add to a game would be fairly straightforward.

Defining ”check” in GDL is nontrivial, as discussed above, but rules around check are also not strictly necessary for chess. Optimal strategy in chess that does not have check, after all, is almost the same as in the game with check, absent the occasional forced stalemate. Since it would likely involve a substantial performance reduction, rules that require checking for ”threats” like check should likely be omitted absent major efficiency gains in automated players.

Our code currently lacks three major features of Metagame: leaps, promotions, and drops. Leaps, as stated above, are a movement option. A promotion is a rule by which a piece can change its type, usually upon reaching the opponent's back rank. Both of these would complicate the piece generator, since their mechanics and their value depend on the existence and value of other pieces. Pawn promotions, as in chess, would be very easy to implement, though; it would be a simple change to the generated GDL for pawns, once such was added. Drops are another more complicated thing to add, this time to the GDL. A drop is a move where one player adds a piece (usually one they have captured) to the board that was not there before. Adding drops would require that the GDL code track which pieces a player has captured, and which they have dropped. This would not be too difficult, since captures are already slightly separated from regular moves, but it would be nontrivial.

Metagame also allows for variation in the players' goal; this would be more or less straightforward to add to the game, depending on what alternate goals are allowed and what effects these alternate goals have on piece placement rules. If desired, different goals could be scored differently, allowing players to aim for alternate objectives if their primary goal is not likely to be achieved.

8 Conclusion

We have implemented the framework to achieve our goal of generating variants. Our encoding allows us to easily generate new variants to test. This separates the problems of figuring out what variants are interesting to test, and figuring out how to test a variant. We were unable to finish translation of encoding into game description, but once this part is plugged in, we should be able to test the variants that we have generated.

References

Alex Landau. Arithmetic in gdl. 2013.

Alex Landau. Rewriting chess (again). 2013.

Mark J Nelson and Michael Mateas. Towards automated game design. *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing*, pages 626–637, 2007.

Barney Pell. Metagame in symmetric chess-like games. In H J van den Herik and L V Allis, editors, *Heuristic Programming in Artificial Intelligence 3 - The Third Computer Olympiad*. Ellis Horwood, 1992.