Bark: An Attributed Graph Grammar Chess Engine

Davide Marincione

June 2, 2024

1 Introduction

Chess is a fairly complex game, with a relatively wide set of rules and a small but non-trivial branching factor (unlike Go, which has very simple rules but huge branching factor). This makes it a good candidate for projects such as Bark. Unlike traditional chess engines, Bark is not programmed in any language: it is instead a set of rules that can be applied to a graph, which is then used to play chess. This makes it a very complex and unwieldy project, as it requires a lot of work to define the rules and the graph structure. In this report, we will describe the rules and the graph structure used in Bark, how their mechanisms differ from a usual chess engine, and we will discuss the challenges and limitations of the project.

A small disclaimer Since this report is dealing with attributed graph grammars, it is full of words like "node" and "edge"; to avoid repetition, we will use the formatting **node** and ***edge*** to decrease the verbosity.

2 Representing the board

Chess cannot be played without a chessboard, therefore the first step in creating Bark was to define a graph structure that could represent the board.

How it is done Fortuitously, a chessboard is composed of 64 squares and, because of that, it can be conveniently be represented as a set of 64-bit integers, where each bit gives the value of a square with respect to specific information of the board. Because of this, a chessboard is usually represented with twelve 64-bit integers, one for each piece type and color. Not only is this representation very compact, but it also allows for very fast operations, as bitwise "magic bitboard" operations are at the core of every modern chess engine.

Bark's nodes Bark, however, does not use this representation. Instead, the board is composed of 64 **square** and 46 **slide_group** nodes. Each **square** is connected to its neighbors via a ***adj*** and to its **slide_group**s via a ***part_of***. Neighbors are defined only to be those **square**s with a common edge, and **slide_group**s are used to represent the groups of squares reachable in a single move by sliding pieces (bishop, rook or queen). Because of this, **slide_group** is actually an abstract class from which **bishop_like_group** and **rook_like_group** inherit. For ease of view and without real impact to the system: **square** is actually an abstract class which is inherited by **white_square** and **black_square**, **bishop_like_group** too is abstract, inherited by **left_group** and **right_group**, and **rook_like_group** is inherited by **ver_group** and **hor_group**. Finally, each **square** has a `file` and `rank` attribute, which are used to represent the position of the square on the board.

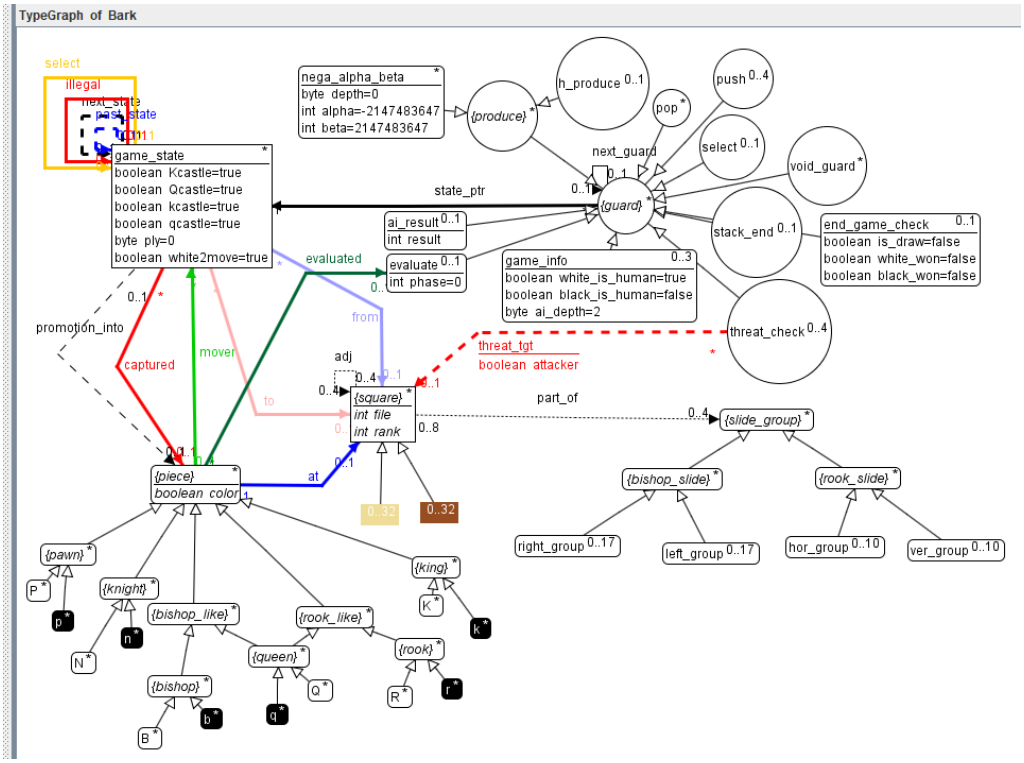

Figure 1: The type graph of the system.

Construction by rules We could have set up the board by hand (as we actually do to position the pieces and starting the game), but that would have been quite tedious because of the multiple edges needed to make it complete. Instead, we use a set of rules to produce it, which are as follows:

- `produce_first_square` creates a first square of the board (the white top left square, `file=0` and `rank=7`), if it has not been created yet. This is then used as a seed for the rest of the construction.

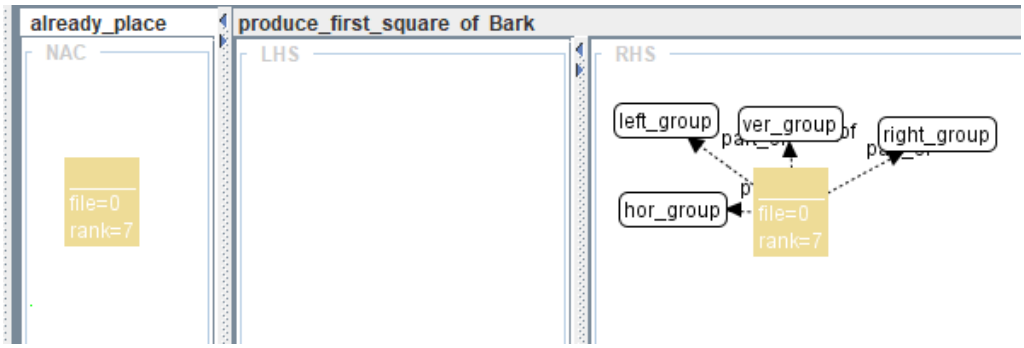

Figure 2: `produce_first_square`

- `produce_top_rank_b/w` respectively create black and white squares of the top rank, if the they have not been created yet.

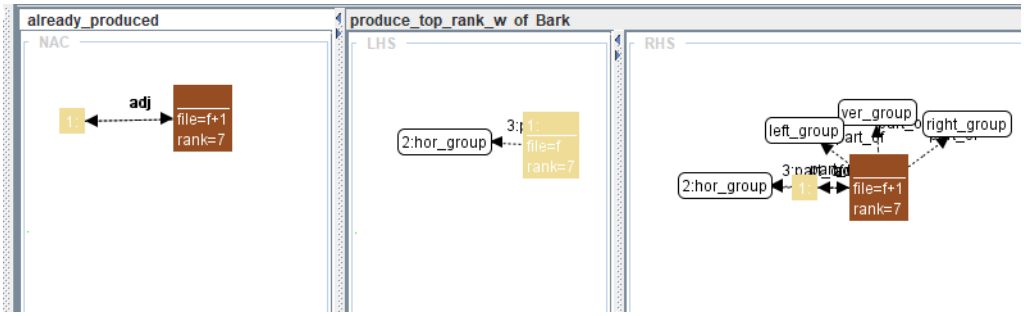

Figure 3: `produce_top_rank`

- `produce_first_file_b/w` respectively create the black and white squares of the first file, if they have not been created yet.

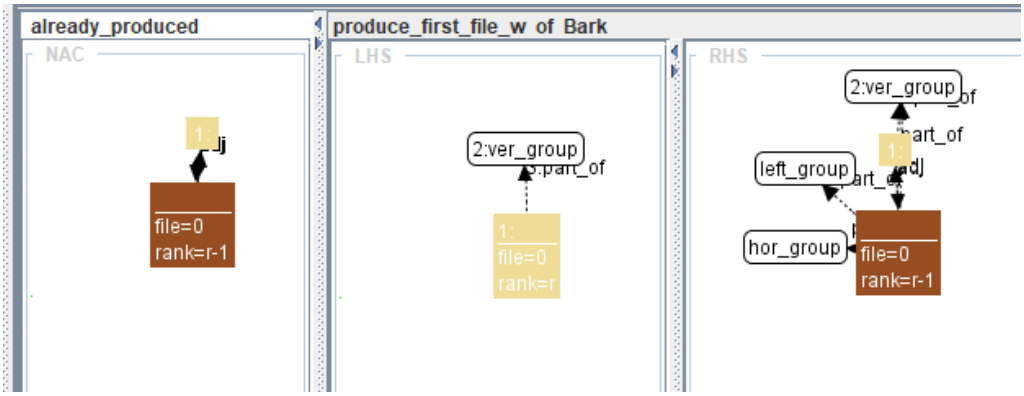

Figure 4: `produce_first_file`

These first rules create the top rank and the left file of the board. The rest of the board is then created by applying:

- `produce_square_b/w` creates a black or white square, if it has not been created yet.

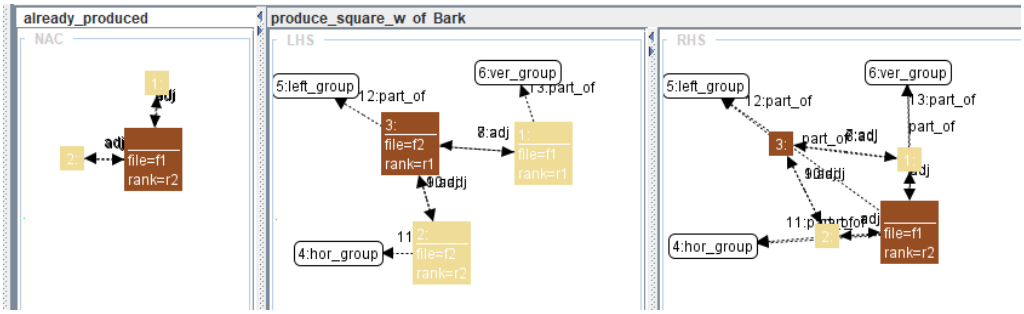

Figure 5: `produce_square`

- `add_right_group` adds a **right_group** to squares on the last file, if it has not been added yet.

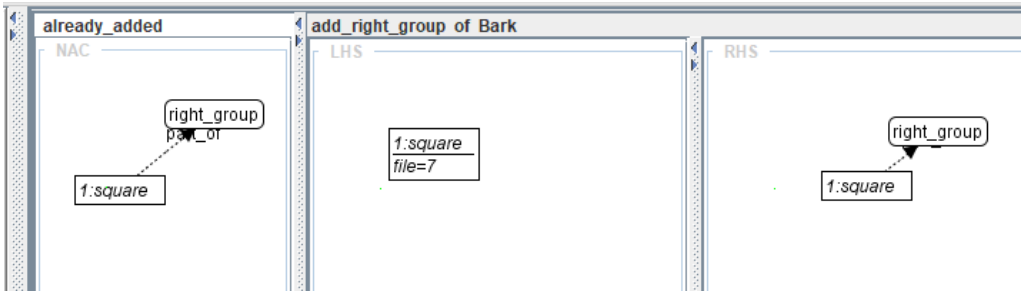

Figure 6: `add_right_group`

- `produce_right_group` connects top-right to bottom-left the squares to the same group, if it has not been done yet.

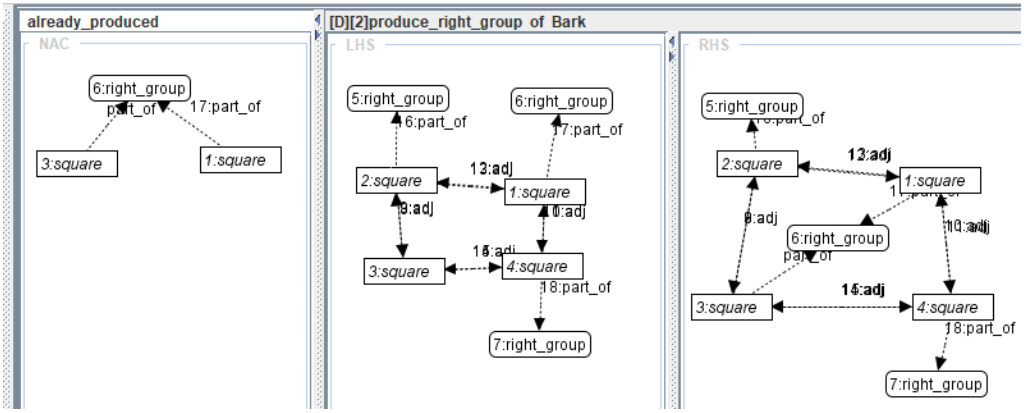

Figure 7: `produce_right_group`

These rules are then applied in a loop until the board is complete. Of course, this is just a *topological* description of the board, and, indeed, applying these rules doesn't result in a good-looking graph.

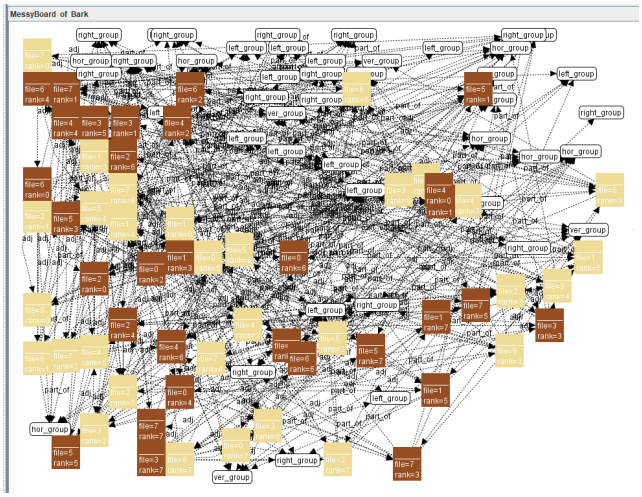

Figure 8: The board after applying the rules.

To make it comprehensible, we order it by hand.

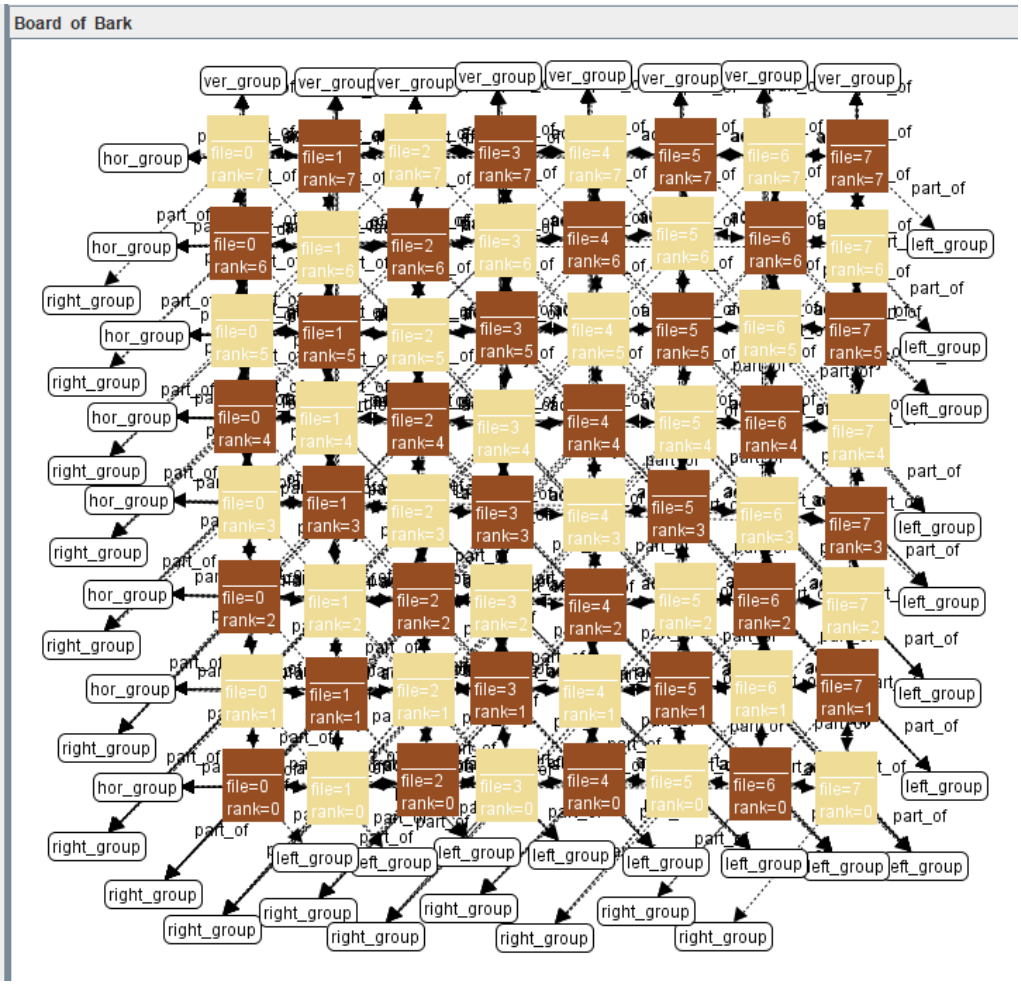

Figure 9: The board after hand-made ordering.

3 Representing game states and enabling complex behaviors

Pieces are represented as nodes connected (via an ***at***) to the squares they are on. With a boolean `color` attribute (`false` for black pieces and `true` for white ones), **piece** is the parent class, which is inherited by the **pawn**, **knight**, **bishoplike**, **rooklike**, and **king** nodes. Specifically, the **bishoplike** and **rooklike** nodes are further inherited by **bishop**, **rook**, and **queen** nodes, the latter of which is the only example of multiple inheritance in the system (as it moves like a rook and a bishop). A game state is not only composed by the pieces' positions on the board, but also by the information about the current rights to move, the castling rights, the last move, and the move counters. This information is stored, explicitly via attributes and implicitly via edges, in the **game_state**.

Not only structure Until now, we've only focused on defining a representation for the game, without including a description for its behavior. Graph rewriting mechanisms by themselves are enough to define a description for the rules of a game such as chess. But, to simplify the design of complex algorithms, we define another set of nodes all inheriting from **guard**. These compose a stack of operations which are used to enable only a subset of the rules at a time. Thanks to these, we were able to define the generation of moves, the evaluation of the board, the exploration of the game tree, and more.

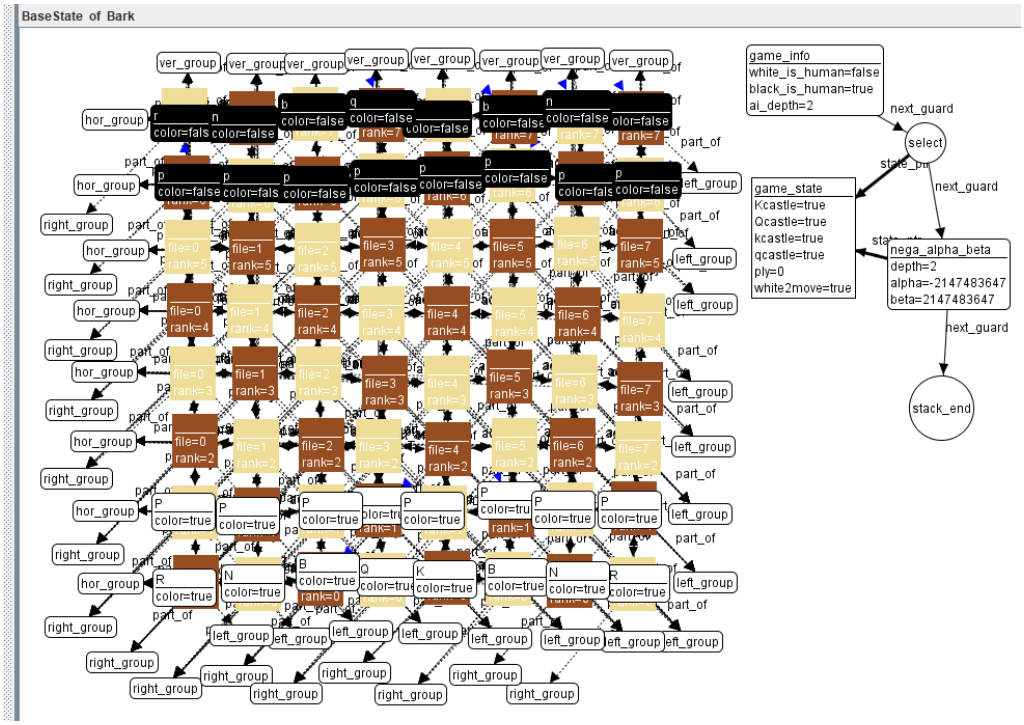

Figure 10: The starting configuration of the board.

Out of these **guard** nodes, there are three that are designed for the control of the stack itself:

- **game_info** acts both as the start of the stack and as a container for the game information (which player is human and the depth of the search).
- **void_guard** has a usage akin to a "no-op" operation, as it is used to block all rules until it is removed.
- **stack_end** is the end of the stack, and it is used to signal which guard is at the top of the stack.

4 Generating moves

How it is done As cited previously, in a traditional chess engine a lot of computations are done via "magic bitboards". These are perfect hashing algorithms which can trivialize a lot of checks, even the generation of movements. Without getting into the details, the moves are produced in a two-phase manner:

1. The result of every possible move is generated (via magic bitboards), without any forward checking (e.g. leaving the king in check), as that is usually too complex and slow to implement. The resulting set is called "pseudo-legal moves".
2. For every possible resulting board, the king is checked for being in check. If it is, the move is discarded. This filtered set is called "legal moves".

A further information to keep into account is that a chess engine usually enables "push" and "pop" operation on moves (i.e. to make a move and then undo it). This is done by keeping a stack of the boards, and pushing a new configuration every time a move is made. This is a very efficient way to implement the "undo" functionality, as it doesn't require to recompute the whole board every time. Furthermore, it is also used to implement the exploration of the game tree, as it allows backtracking to the previous state of the game.

How Bark does it, at large Bark, however, cannot use magic bitboards and, furthermore, can't afford to store a tree of board representations, as a single configuration spans many nodes and edges! We devise another system that, while extremely slower, is able to run on AGG.
We maintain a single board instance, which represents the state that the system is currently handling (be it

the current game state or a further state being pondered upon by the engine), and represent the game tree by connecting the **game_state** nodes: when pushing and popping the moves, instead of retrieving a stored configuration, we use the information present on the **game_state** to execute or revert the move by operating on the board instance. To implement push, pop and threat check operations we design multiple rules and their respective guards (**push**, **pop**, and **threat_check**).