Bark: An Attributed Graph Grammar Chess Engine

Davide Marincione

May 30, 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.

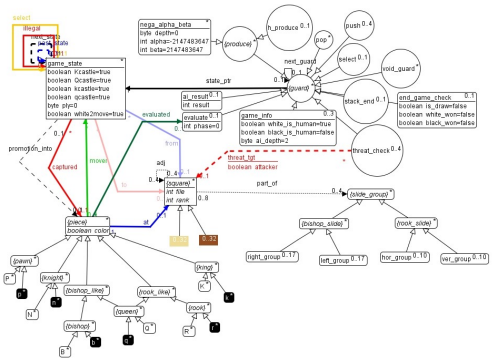

Figure 1: The type graph of the system.

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.

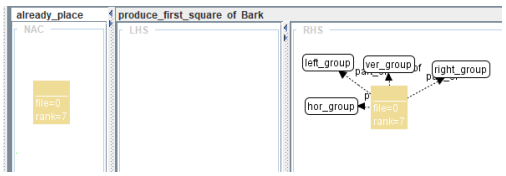

Figure 2: `produce_first_square`

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

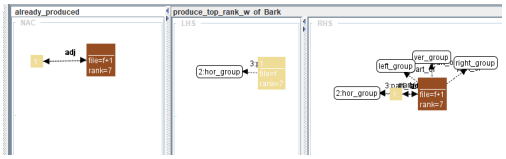

Figure 3: `produce_top_rank`

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

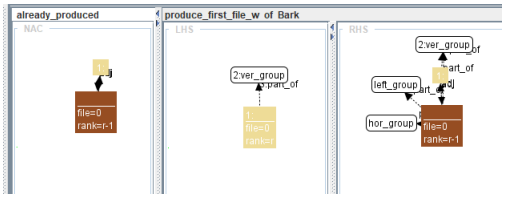

Figure 4: `produce_first_file`

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.

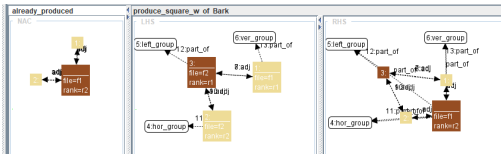

Figure 5: `produce_square`

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

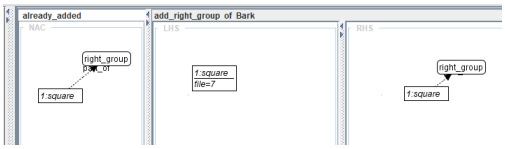

Figure 6: `add_right_group`

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

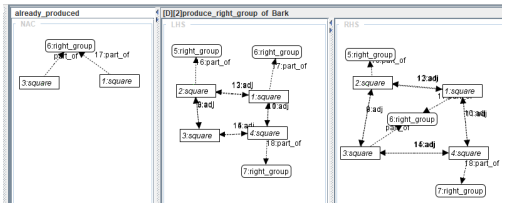

Figure 7: `produce_right_group`

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.

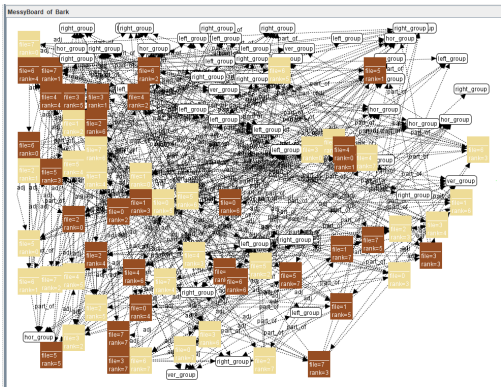

Figure 8: The board after applying the rules.

To make it comprehensible, we order it by hand.

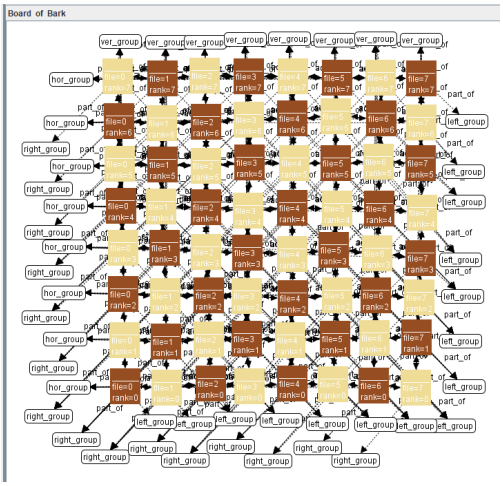

Figure 9: The board after hand-made ordering.