

Chess-Num - PLOG 2020

FEUP-PLOG, Class 3MIEIC03, Group 3

Faculty of the engineering of the university of Porto

Abstract. This paper is a brief analysis of our solution of the Chess-Num problem, developed in the context of the PLOG U.C. The solution is implemented in sicstus prolog using its finite domain constraints library (clpfd). TODO copiar conclusao para aqui (conclusões e principais resultados)

1 Introduction

In this paper we describe our solution to the Chess-Num problem, which can solve any instance of the puzzle, generate a random solution, and present the result in a human readable way. We start by describing the problem, afterwards we explain our implementation, and then we analyze the solution/approach.

We weren't able to find any other approaches/references to this problem.

2 Problem Description

The Chess-Num problem is a chess-related puzzle in which, given a set of numbered cells in the chess board, one tries to place the six different chess pieces (rook, queen, king, bishop, knight pawn) in such a way that the number of each given cell corresponds to the number of pieces attacking that cell. The source of this problem has a description of this problem and examples of boards and their solution.

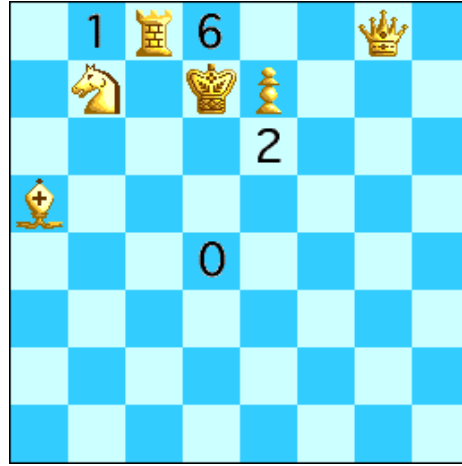


Fig. 1. An example puzzle with four numbered squares and its unique solution.

3 Approach

3.1 Decision Variables

The decision variables correspond to the coordinate pair of each piece. They are all within the domain $[0, 7]$ (inclusive). Furthermore, all of these coordinate pairs are distinct both between each other, as well from the given numbered cells' coordinates.

3.2 Constraints

The first restriction placed is applied to the coordinates' pieces. It ensures that all of them and the chosen cells' coordinates are all different. To achieve this we use the library predicate `all_distinct/1` with a list of indices for each coordinate $[X, Y]$, each index corresponding to $Y * 8 + X$.

The second set of restrictions is applied for each given cell, and it ensures that, for each one, its value corresponds to number of attacking placed pieces. For this, the `value/3` predicate is used. It first calculates a boolean value for each piece through the use of the predicate `value[Piece_Name]/3`. The calculated value corresponds to 1 if the piece is attacking the cell and to 0 if it isn't attacking. A sum is then used to verify that the sum of all pieces' values correspond to the given cell's value.

The predicates `valueKing/3`, `valueKnight/3`, `valuePawn/3` are straightforward, as collisions aren't possible with these pieces. Their implementation applies restrictions based on chess' rules. `valueKing/3` yields 1 if the given cell is orthogonally adjacent to the king. `valueKnight/3` yields 1 if the given cell makes a 'L' pattern with the knight. `valuePawn/3` yields 1 if the given cell is

diagonally and adjacent to the pawn. If any of these predicates yield 1, constraints are placed to the pieces' coordinates to ensure that the chess' rules are followed. During the program execution, these constraints may be too rigid, causing the predicate to fail and backtrack, lifting the previous placed restrictions and yielding 0.

The predicates `valueQueen/3`, `valueBishop/3`, `valueRook/3` behave similarly to the previous explained predicates when it comes to applying chess rules. The rook can attack horizontally or vertically, the bishop can attack diagonally and the queen can attack horizontally, vertically or diagonally. The main difference is that all of them need also to ensure that no other piece is placed between their attack path and the given cell. For this purpose, we use the predicate `others_is_not_between/3`, which calls `is_not_between/3` for each other piece with a given attack path (diagonal, vertical or horizontal) correspondent to the attacking piece's attack move. `is_not_between/3` yields 1 and restricts the given piece to not be within attack and yield 0 through backtrack. The `others_is_not_between/3` uses this result to verify that all of the members of the given list aren't in the attacking piece's path. If any other piece blocks the attack path, this predicate yields 0.

4 Solution Presentation

The main (outermost) predicates that allow for a problem/solution visualization are the `display_board/1` and the `display_board/2` predicates.

4.1 The `display_board(+NumberedSquares)` predicate

This predicate will draw a chess board with the given numbered squares coordinates showing the given values. This is used to show a problem without its solution. It should be noted that the predicates used to visually represent a solution do so "on-the-fly". This means that only the input data structures are used instead of a *game board* structure being generated and displayed.

The call `display_board([[1, 0]-1, [3, 0]-6, [4, 2]-2, [3, 4]-0]).` yields the following:

	1		6				
				2			
			0				

Fig. 2. The textual representation of the puzzle show in fig1 1 (without its solution).

4.2 The `display_board(+NumberedSquares, +Coords)` predicate

Similarly to `display_board/1`, this predicate will draw a chess board with the given numbered cells. Along side those, the pieces in the given coordinates will also be represented. The representation of each piece is as follows: King - **K**, Queen - **Q**, Rook - **R**, Bishop - **B**, Knight - **Kn**, and Pawn - **P**.

The call `display_board([[1, 0]-1, [3, 0]-6, [4, 2]-2, [3, 4]-0], [[3, 1], [6, 0], [2, 0], [0, 3], [1, 1], [4, 1]])` yields the following:

	1	R	6			Q	
	Kn		K	P			
				2			
B							
			0				

Fig. 3. The textual representation of the puzzle show in fig1 1 (along side its solution).

4.3 Innermost display predicates

// TODO ?

5 Experiments and Results

6 Conclusions and Future Work

// TODO site do puzzle + slides do prof + docs do sicstus ?

References

7 Annex

// TODO source code + extra graphs