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.

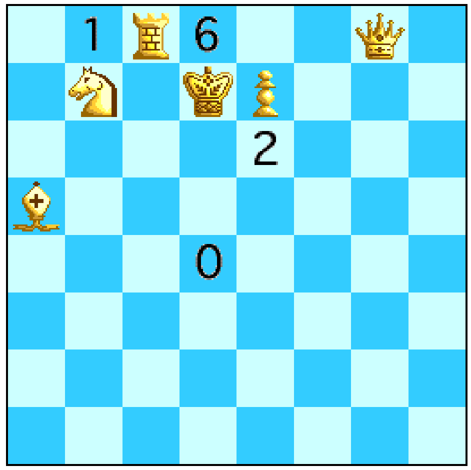

Fig. 1. An example puzzle with four numbered squares and its unique solution.

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

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:

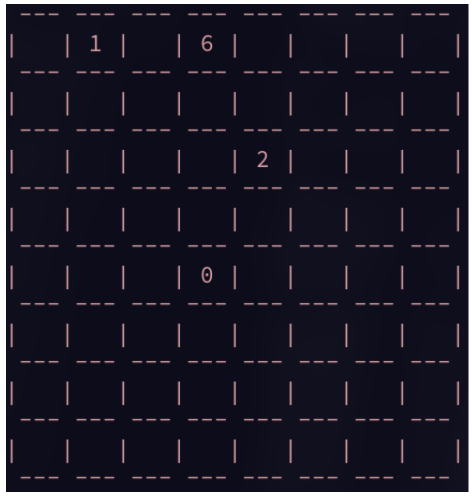

Fig. 2. The textual representation of the puzzle show in figure1 (without its solution).

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:

Fig. 3. The textual representation of the puzzle show in figure1 (along side its solution).

4.3 Innermost display predicates

// TODO ?

5 Experiments and Results

5.1 Dimensional analysis

The impact of the quantity of numbered squares The problem in figure 1 is the simplest problem we found with an unique solution. Our solver finds that solution in about 0.02// TODO seconds. From the problems in the puzzle's web page, this is the one that is solved the fastest.

This puzzle has four numbered squares, one of which has the value 6. This is of note because, we always start constraining the pieces coordinates in a that they can attack the given numbered square. A numbered square with value 6 implies that all six pieces are attacking it, thus pruning the possible coordinates for the pieces by a lot.

We can compare this puzzle to the following which also has a single solution, but no numbered square with the value 6. It takes about 1.04 seconds to find the solution. This is significantly higher than the previous one even though there are the same number of numbered squares.

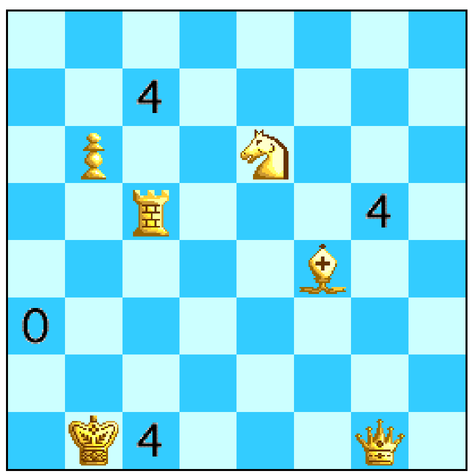

Fig. 4. A puzzle with four numbered squares and a unique solution.

The more numbered squares the puzzle has, the longer it takes to solve it.

The special case for the value 0 As previously stated, our constraints are posted aggressively on the attack for each for each numbered square. This is very inefficient when the numbered square we're dealing with has value 0 (no one can attack it). This led us to create a special case for these numbered squares where we explicitly constraint all pieces to not attack the numbered square from the get go, instead of trying to attack it first.

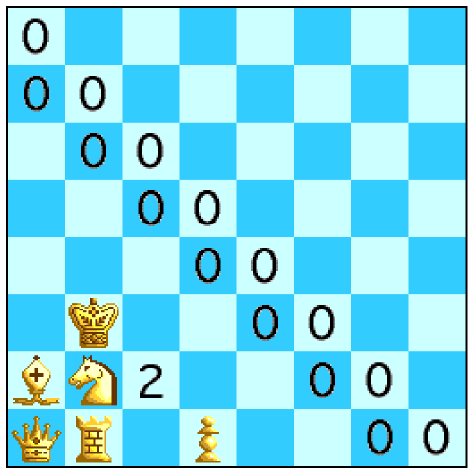

Fig. 5. A puzzle with four numbered squares and a unique solution.

The processing order of numbered squares When trying new puzzles, we noticed that changing the order of the inputted numbered squares, thus not changing the problem, but the processing order of the squares, had an effect on the speed of discovering solutions. This effect could be has mild has a 0.05 seconds difference or as extreme as some hours.

The most extreme case we found was following puzzle. This puzzle has a single solution and the program was tacking hours to find it. By changing the order the numbered squares are processed, we were able to reduce the processing time to 7.62 seconds.

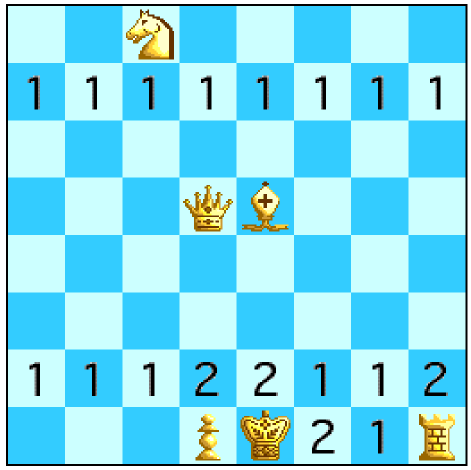

Fig. 6. A puzzle with four numbered squares and a unique solution.

6 Conclusions and Future Work

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

References

7 Annex

// TODO source code + extra graphs