The k Knights problem - Documentation

Matheus Ribeiro Alencar

1 Introduction

In this problem, we work on a n x n chessboard, with $n \in \mathbb{N}$, the problem consists in putting the highest number of Knights in a way that no knight can attack other knight. This problem is a variation of the *8 queens problem*, see more at `https://en.wikipedia.org/wiki/Eight_queens_puzzle#:~:text=The%20eight%20queens%20puzzle%20is,in%20the%20mid%2D19th%20century`.

The core idea of the heuristic used, is to apply knights in the chessboard in a way that they restrict the smallest possible number of positions. This was the heuristic used to evaluate how good a given position is for us to place a knight on it. Afterwards, we see the best solution and compare it with the heuristic (the heuristic is not always the optimal solution, in some cases we will see it is way worse than the optimal solution).

1.1 Link to the notebook

Link to the collaboratory with the code: `https://colab.research.google.com/drive/1wbjZCI8e4ugvC-eqcC1lKysV8LoSVjj8?usp=sharing`

1.2 Heuristic

We utilize a matrix to represent our chessboard, each number $i \in \{1, 2, 3, ..., n^2\}$ of the matrix indicates the ith knight placed in the board. Note that n^2 is the number of squares setting our upper limit, rarely we are going to be able to put n^2 knights in our matrix. The 'X' characters in our matrix represent the positions where we have at least one knight that can attack it.

We also return the number of knights in the board, this number is given by the highest number in the matrix.

For the heuristic, we create a copy matrix, where for every position we will check if we apply a knight on that position how many squares will be restricted. For avoiding conflicts with already placed knights we decrease from the positions with 0, $\{-8, -7, ..., -1, 0\}$ that is, in the end we will check on the matrix the best square to place a knight (best square is the square with value v where $v \leq 0$ and v is the highest possible).

1.3 About the main program, our binary tree and all possible configurations

We add every possible solution of the chessboard to our queue, the first and last element to be removed from the queue will be a board with zero squares occupied. When the empty board leaves the queue it creates 2 children, the first one being the chessboard with an 'X' marked on the first available position, in the second children we put a 'K' on the first available square.

The 'X' represents the non existence of a knight on that position, the position marked with 'X' will be maintained throughout our execution. The 'K' marking represents a square occupied by a knight, this marking will restrict some positions on our chessboard accordingly to the knight's possible attacks.

1.3.1 Pruning

In regards of optimization our program, a prune was implemented. Along with our heuristic it shortens our chessboard instances and reduces drastically the analized chessboards and solutions. The prune is implemented in the beginning of the *branch* function on the following way:

1. We go through our chessboard P;
2. For every number of elements 0 and 'K' we increment 1 to a counter;
3. In the end, our counter will have the highest number possible of knights on that specific board configuration (ignoring knight's move restrictions);
4. We compare the counter value to the value of the good solution found by the heuristic;
5. If the counter value is equal or higher than the value found by the heuristic, we keep on going in that branch;
6. Otherwise we prune that branch.

2 Functions

2.1 initialize_chessboard(dim)

Function to initialize a chessboard with dimensions *dim x dim* with 0's. Our chessboard is represented by a matrix with every element equal to zero.

2.2 place_attacks(pos_x,pos_y,chessboard)

Function to restrict the squares of our chessboard based on the position we are currently putting a knight.

2.3 check_attacks(chessboard)

Main function from our heuristic, in this function we create an auxiliary matrix and check how many positions every square, if occupied by a knight, would restrict. It returns the positions(x,y) of the best square possible.

2.4 place_knights(chessboard)

Function which executes our heuristic, based on it we call other subfunctions such as *check_attacks*, *place_attacks* and in the end *reset_chessboard*.

2.5 reset_chessboard()

Function that will end our heuristic part. On it we pick our best chessboard configuration (one with the highest number of knights) by our heuristic. We also keep this number to compare posteriorly with the solutions we find in our queue and binary tree.

2.6 branch(P)

Function that will branch P in $P0$ and in $P1$. In $P0$ we mark the first avaiable position with 'X' and in $P1$ we mark with 'K'. Both 'X' and 'K' are non available squares, but only 'K' represents a knight. In this function its implemented our pruning by non-optimality, comparing it's result with the heuristic's one.

3 Tests

The solutions made by the heuristic are represented in our code with integers and characters 'X'. Integers represent the knights in our code, to facilitate our visualization we will just mark them with 'K'

3.1 Chessboard 3x3

For the $dim = 3$ we got less than 1 second to calculate a total of 16 possible solutions, we also found a value from the heuristic which is equal to the maximum number of knights possible.

- Heuristic_value = 5
- Maximum_value = 5

$$Heuristic\_configuration = \begin{bmatrix} X & K & X \\ K & K & K \\ X & K & X \end{bmatrix}$$

$$Maximum\_configuration = \begin{bmatrix} K & X & K \\ X & K & X \\ K & X & K \end{bmatrix}$$

3.2 Chessboard 4x4

For the $dim = 4$ we got about 2.12 seconds to calculate a total of 62 possible solutions, we also found a value from the heuristic which is equal to the maximum number of knights possible.

- Heuristic_value = 8
- Maximum_value = 8

$$Heuristic\_configuration = \begin{bmatrix} K & K & K & K \\ X & X & X & X \\ X & X & X & X \\ K & K & K & K \end{bmatrix}$$

$$Maximum\_configuration = \begin{bmatrix} K & X & K & X \\ X & K & X & K \\ K & X & K & X \\ X & K & X & K \end{bmatrix}$$

3.3 Chessboard 6x6

For the $dim = 6$ we got about 220.76 seconds to calculate a total of 25106 possible solutions, we also found a value from the heuristic which is different from the maximum number of knights possible.

- Heuristic_value = 15
- Maximum_value = 18

$$Heuristic\_configuration = \begin{bmatrix} K & X & K & X & K & X \\ X & K & X & K & X & K \\ K & X & K & X & K & X \\ X & X & X & X & X & X \\ X & X & X & X & X & X \\ K & K & K & K & K & K \end{bmatrix}$$

$$Maximum\_configuration = \begin{bmatrix} K & X & K & X & K & X \\ X & K & X & K & X & K \\ K & X & K & X & K & X \\ X & K & X & K & X & K \\ K & X & K & X & K & X \\ X & K & X & K & X & K \end{bmatrix}$$