Chess Table-base Programming Project

[PhynnisFine/Chess-Tablebase (github.com)](https://github.com/PhynnisFine/Chess-Tablebase)

# How to Use:

A screenshot of a game

Description automatically generated

* Place pieces on the board using the coloured buttons on the left (P=pawn, N=knight, B=bishop, R=rook, Q=Queen, K=King). Only positions with 3 pieces (2 kings and a piece) will be evaluated, other positions will probably crash the program if it tries to evaluate them.
* Once a position is set up, keep clicking on the top move to watch the checkmate.
* Change which colour is next to move using the buttons in the top left.
* Import FEN can be used to import a position using the FEN representation by copying it into the text box below.
* Evaluate a position using the Go! button in the top right, results are displayed in the list box below.
* Click on a move in the list box to play the move on the board.
* Buttons on the right are developer test buttons, apart from the Table Name button, which can be used to enter a set of pieces (e.g. KQvK) and generate that table.

# What is a table base?

A chess table base is a database of all possible positions with a low number of pieces, and whether they will end in a checkmate for either side, or a draw, with best play from both sides. If the position will end in a checkmate, the number of ‘ply’ or ‘half moves’ (since a move in chess is one move from each player) before that checkmate will occur is also stored. This is generally referred to as the DTM – the Depth To Mate. For example, a position where white can get checkmate in one move would be DTM 1. Since 2017, table bases have been complete for 7 pieces (including kings), covering 4\*10^14 positions.

When I first heard about this concept, I found it fascinating, because it seemed like the closest chess could get to being mathematically solved. This is very different to normal chess AI’s, since they use position evaluation (how likely a position is to be winning, based on number of pieces and other ideas) but rarely know for certain whether a checkmate is inevitable. Table bases also sounded very different to chess engine, since they work backwards from checkmates rather than forward from a given position.

I decided to attempt to create a complete 3-piece table base (so one where 1 side has a king and a piece, and the other side has a lone king), without researching the algorithms behind generating them first. The challenge would be figuring out how the algorithms to make this work while only knowing what the output should look like. I decided to base my interface design on a popular online table base: [syzygy tables](https://syzygy-tables.info/), since it seemed easy to use, and had no major issues.

However, I wanted to change one thing, partly to make my project feel unique, and partly because it was what I wanted to know. Syzygy tables normally displays DTZ – Depth To Zero, meaning zeroing move. This is a move that resets the 50-move rule: if 50 moves are played by each player without a ‘zeroing move’ the game ends in a draw. Therefore, in a real game, this value is important, since a DTZ of over 100 (2 players \* 50 moves) means a certain draw, even if a checkmate would have occurred eventually without the 50-move rule. But I am more interested in the theory behind table bases, so the only value I want to be displayed in any position is DTM, even in positions that will technically end in a draw.

Extra info:

[Endgame tablebase - Wikipedia](https://en.wikipedia.org/wiki/Endgame_tablebase)

# How does a table base work?

The specification for the table base comes in 2 parts. First, the evaluation part, which must take a set of pieces as input, and output the evaluation for every position with those pieces. Secondly, the interactive part, where the user inputs a position, and the system must output the evaluation for that position, as well as the evaluation for every move that could be made, to tell the user which move is best.

This is roughly how the evaluation algorithm works:

* Generate and store all possible positions with a given set of pieces.
* Store the evaluation of all checkmate positions as DTM 0 for whichever side is checkmating.
* Then, while there are still some DTM X positions left:
* Find all DTM X positions.
* For each position, undo all moves (find all positions that the DTM X position could have come from), and for each of those positions, assign an evaluation of DTM X+1, unless the evaluation is currently better (i.e. DTM <X+1).
* Find all DTM X+1 positions.
* For each of these positions, undo all moves again. However, for each of the new positions, we have to make sure that the losing side has no better (escape) options, so all legal moves must be made from that position to check if a better evaluation for that side can be found. If a better evaluation is not found, then the position is assigned an evaluation of DTM X+2.
* X+=2
* Find all DTM X positions and repeat.
* Eventually, one side will always have a better option in every position, and a max depth is reached, at which point evaluations can be permanently stored.

The interactive side of the system is much simpler:

* Find the input position in the table base and return the evaluation.
* Then generate all moves from that position, find all the resulting positions in the table base, and return the evaluations for all of those positions.

Of course, a complicated (actually overcomplicated because this was definitely a learning process, but still functional) data handling system is required for both algorithms, but this is the general idea. I decided to use CSV’s for the permanent table base storage, I was storing a string (the FEN) and 2 integers (the evaluations) for each position.

I actually glossed over 2 details for simplicity: firstly, every position has 2 evaluations, a one for each player to move, and a colToMove variable (not an attribute of either the Board or Position object since both apply regardless of which player it is to move, although if I had more experience at the start I might have done it this way). The 2nd detail is that actually, after evaluating checkmate positions, some other positions can also be evaluated immediately. For example, a pawn could be promoted to a queen, so a different table base for a different set of pieces needs to be checked (since table bases a generated one set of pieces at a time). These other table bases for different sets of pieces are called ‘dependencies’, which you can see at the bottom of the syzygy tables page.

# The Result

Ultimately, I was successful in making a complete 3-piece table base. The only sets of pieces that need to be considered are KQvK (King and Queen vs King), KRvK and KPvK, since a single knight or bishop is always a draw. I could also only consider the cases where one side has the piece, since colours can be flipped for an equivalent evaluation. I am also happy with the interface, even if the board is not quite as interactive as the online version, and I left the pieces as letters.

I had never done a programming project on this scale before, and I am also not that familiar with c#, so I learnt a lot during the project, especially about basic object use, although I didn’t manage to use methods. This learning curve led to the project being a lot less organised than I would have liked, but I have at least managed to tidy the generating moves functions up a bit.

I want to work on the project next summer, because having finished the 3-piece table base I can see a lot of things I would do differently now, and a lot of optimisation potential, that might eventually allow me to make a 4-piece table base, for at least a few sets of pieces.

# Documentation

I really don’t have time to list all the functions at the moment, and they were badly organised anyway, but here are some important data representation methods:

Pieces:

* 0 = Empty Square
* 1 = Pawn
* 2 = Knight
* 3 = Bishop
* 4 = Rook
* 5 = Queen
* 6 = King
* If white, +8.

Evaluations:

* DTM X for white: 1000 – X
* DTM X for black: -1000 +x
* Draw: 0
* Illegal position: -9999
* Insufficient material ‘tableNum’: -1

TableIndex and TableNum:

* TableIndex is a number that uniquely represents every position in a given set of pieces. For example, in KQvK, TableIndex = (64^2)\*WKingPos +64\*BKingPos+QueenPos.
* TableNum is a number which uniquely represents every set of pieces which has an existing ‘table’ (database). Since there are only 3 tables for 3 pieces (coincidentally), each number was manually picked, but an automatic allocation should be developed for more pieces.

Display:

* 0-63 from left to right, top to bottom. Squares are always indexed Y,X (row is more significant than column).

Objects:

* Position:
  + Int[8,8] board (named before Board object created, should be changed to ‘pos’
  + Int WhiteEval (WhiteToMoveEval)
  + Int BlackEval
* Board:
  + Int[8,8] pos
  + Int tableIndex

# Still to do

* For chess algorithms:
  + Generate moves, pawn move, pinned checks (using checkRays)
  + Generate moves, already in check, everything apart from king moves
  + Undo moves, king moves *properly*
  + Undo moves, already in check (all of it)
* For Evaluation algorithm (increasingly longer term):
  + Use tableIndex numbers rather than boards and FEN’s for everything (including text file storage and in ‘Tablebase’ array of positions. This will require changing the position object, creating a tableIndexToBoard function, among other things
  + Change the iterative part of the algorithm so that it does not store a list of positions (or indexes) but updates all indexes immediately. This approach can be adapted for ‘potential mate in x’ (e.g check all escape options immediately after ‘undo moves’ for every position)
  + Update tableIndex to work for a variable number of pieces (atm it is fixed at 3)
  + Update tableBaseIndex to work with more pieces
  + Update GenerateAllPositions for same
  + Update the part of the algorithm that finds all dependencies so that it actually works for all cases.
  + Treat rotationally similar and symmetrical positions as the same positions, using a ‘genEquivBoards’ function that takes the board / index, flips and rotaes it in all unique ways, and returns an array/ list of all indexes. This reduces the storage requirements roughly 8 times. This change means that, when a new positions has an evaluation of DTM 5, all 8 similar positions must be checked, if 1 has an existing evaluation use that for all 8, if not assign the evaluation to the first in the list and move on. Since so many positions are ignored with this method, a null flag will be used (as an extra boolean attribute of the position object) so that empty positions can be passed over quickly. This will likely be the final required change before attempting to create a 4 piece tablebase.
  + Only use text files while the interface program is running, since speed is not a major issue. The tablebase array would not be required, saving a lot of memory.