

MATER – A Simple Checkmate Searching Program

© 2019 Valentín Albillo

Abstract

MATER is a simple checkmate searching program I wrote in 1998 which, given a legal chess position, the side to move and the maximum number of moves to give checkmate, it will search and output either a move which gives checkmate in that number of moves (or less), or the fact that there's no such move. It can also search for only-checks mates. Eight worked examples are included.

Keywords: *mater, checkmate search, chess, MS DOS command, Turbo Pascal 7.0, Borland Pascal 7.0.*

1. Introduction

MATER is a simple checkmate searching program, implemented as an *MS-DOS* command. Given a legal position in FEN notation, the side to move, and the maximum number of moves to give checkmate, it will search and output either a move which checkmates in that number of moves or less, or the fact that there's no such move.

Main features:

- Full legal move generation, including full legal castling, *en passant* captures and underpromotions.
- Can find checkmates in any number of moves, up to 64, subject only to available memory and time.
- Can find *general* checkmates or checkmates in which the mating side gives *only checks*. This is useful for finding much faster those long sequences of checks that end in checkmate.
- Accepts positions in FEN notation and checks them for legality and syntax.
- Searches for mates iteratively starting from *mate in 1* and thus always finds the shortest possible mate.
- The search can be halted immediately at any moment by simply pressing any key.
- Outputs the first move that gives the required checkmate, or the fact that one doesn't exist.
- Shows progress for each iteration, including time taken.
- Outputs final total time, as well as the number of nodes examined.
- *MS DOS* command format: takes its parameters from the command line and sets **ERRORLEVEL** with a code indicating the outcome of the search, for testing either from a batch file or a calling program.
- Can be invoked from any program which can execute *MS DOS* commands (i.e: *Visual Basic* for *Win*).
- The output can be redirected to a file, the printer, another program, etc.
- If executed without parameters, it displays the correct syntax and an example.
- Fully commented, easy to understand, no-tricks didactic source code to help study and use in your own chess programs.

What it isn't:

Being a didactically-oriented example, it isn't heavily optimized nor does it use the most efficient but highly complex algorithms there are for board representation and move generation. The emphasis has been placed instead on *didactic* code simple to understand.

As a result, its running speed is nothing remarkable and can't generally compete with more elaborate code. However, some heuristics have been used to speed the search, and in the case of searching for checkmates consisting of a *sequence of checks*, it can find them much *faster* than a more general approach. For instance, in the last example it finds a *checkmate in 12* in ~ 0.40 seconds on an ancient 200 Mhz CPU. See **Examples** below.

2. Usage Instructions

The calling syntax from the *DOS* command line is the following:

```
MATER [FEN position] [max.moves (1..64)] [all/checks (a/c)]
```

where:

- *FEN position* is the board position (including the side to move) in FEN notation. For example:

the standard initial board is: `rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR/ w`

- *max.mov (1..64)* is the maximum number of moves to give checkmate, from 1 to 64.
- *all/checks (a/c)* specifies the type of checkmate to search for:
 - *a*: general checkmates, all legal moves for the mating side will be tried.
 - *c*: only checking moves for the mating side will be tried.

MATER is implemented as an *MS DOS* command, to be executed from the *DOS* prompt either in a real *MS DOS* system, in a *DOS* shell from within *Windows 16/32 bits* or *Linux DOSEMU*, in *DOSBOX* or even from another program which can execute *DOS* commands. Memory requirements are absolutely minimal.

If executed without parameters or with wrong/illegal parameters, *MATER* terminates immediately after printing a message describing what went wrong (if anything) and a brief resume of proper syntax, including an example:

```
>MATER
```

```
INPUT ERROR: wrong number of parameters
```

```
Syntax: MATER [FEN position] [max.mov (1..64)] [all/checks (a/c)]
```

```
Example: mater b7/PP6/8/8/7K/6B1/6N1/4R1bk/ w 5 a
```

```
Note: The output can be redirected: MATER ... > mat.txt
```

```
Exit: - mate in n found: ERRORLEVEL = n  
      - mate not found: ERRORLEVEL = 0  
      - halted by user: ERRORLEVEL = 9999
```

MATER accepts positions in FEN notation, including board and side to move. The FEN position should be like:

```
(row1) / (row2) / (row3) / ... / (row8) / (side to move, b or w)
```

The following is checked and has to be true for a legal FEN position:

- Only characters 'p','n','b','r','q','k' (black) and 'P','N','B','R','Q','K' (white) can represent pieces.

For example: `3S4/2C5/ ...` is illegal, `S` and `C` aren't valid piece descriptors.

- Exactly 8 rows should be specified.

For example: `b7/PP6/8/8/7K/6B1/6N1/ w` is illegal, only 7 rows specified.

- In each row, exactly 8 squares must be specified, either using piece descriptors or a number representing consecutive blank squares in the row.

For example: `b7/PP6/8/8/7K/6B1/6N/4R1bk/ w` is illegal, only 7 squares in that row.

- Each side can have a maximum of 8 pawns, a maximum of 16 pieces, and exactly one king.

For example: `b7/PP6/8/8/7k/6B1/6N1/4R1bk/ w` is illegal, there's no white king.

- The last row should end with a slash (/) and then the side to move (b or w) must be specified, separated by at least one space.

For example: `b7/PP6/8/8/7K/6B1/6N1/4R1bk/w` is illegal, no space before the 'w'.

Apart from the FEN board position (including side to move) you must specify a *maximum number of moves* in which to search for checkmate (from 1 to 64), and if a *general* checkmate should be found (*all* moves are considered for the mating side) or a *special* checkmate in which *only checks* are considered for the mating side.

If the syntax is incorrect, you will get *one* of these messages:

```
INPUT ERROR: max.mov should be an integer
INPUT ERROR: max.mov should be between 1 and 64
INPUT ERROR: Must specify A (all Moves) or C (checks only)
INPUT ERROR: Must specify who moves: B (Black) or W (White)
INPUT ERROR: Illegal FEN position
```

Note:

The routine that performs the parsing, checking and conversion from FEN notation to a board position is called *FEN2Posit*. It accepts a string representing the FEN position, fills an internal structure with the corresponding board position and returns a *boolean* indicating if it was successful or not. Also, this routine automatically assigns *castling rights* depending on whether king and rooks are on their original positions or not. Although the board structure does permit it, no provision has been taken to allow the user to specify castling rights and/or the square for a possible initial *en passant* pawn capture. See the *Program Listing* below.

Once the syntax is correct, *MATER* prints its name and version, a copyright notice, the text representation of the board position together with *castling rights* and *en passant* square, the position in FEN notation, the maximum number of moves to checkmate and the type of checkmate (*all* moves or *only checks*) being sought. For instance:

```
MATER: Mate searching program v1.1. (c) Valentin Albillo

      a b c d e f g h
8 # b - - - - - - # White can't castle long
7 # P P - - - - - # White can't castle short
6 # - - - - - - - # Black can't castle long
5 # - - - - - - - # Black can't castle short
4 # - - - - - - K # No en passant square
3 # - - - - - B - #
2 # - - - - - N - #
1 # - - - - R - b k #
# # # # # # # # #

FEN: b7/PP6/8/8/7K/6B1/6N1/4R1bk/ w

Max. no. of moves to mate: 3
Searching *all moves* for: White
```

Then, an iterative and recursive search is performed, beginning with a *checkmate in 1*, until either finding a mate or reaching the maximum number of moves specified. This ensures the shortest possible checkmate is found, i.e: if the user specified a *mate in 6* but a *mate in 5* does actually exist, it will be immediately returned by the search.

For each root move searched a dot (.) is printed, and after all moves have been searched the time taken for that iteration is printed and the next iteration begins:

```
Mate in 1.....0.01
Mate in 2.....0.18
Mate in 3.....
```

Note: you can immediately stop the search at any moment by simply pressing a key. *MATER* will then terminate at once after printing a message specifying it was halted by the user.

Once a checkmate has been found, the search stops immediately, and *MATER* terminates after printing the mating move and some additional information such as total *time* used (in seconds) and *number of nodes* visited, like this:

```
Mate in 3 with b7xa8=N, t=0.10, nodes=1978
```

Note: the output of *MATER* can be *redirected* to the printer, a file or as the input to another command. This is done as with all other *DOS* commands, consult your *DOS* documentation for details on redirection.

After *MATER* terminates for whatever reason, the *DOS* environment variable **ERRORLEVEL** is set to a value indicating the cause for termination. Possible values are:

ERRORLEVEL = n	indicates a checkmate in n was found
ERRORLEVEL = 0	indicates no checkmate was found
ERRORLEVEL = 9999	indicates the user halted <i>MATER</i> early by pressing a key

If desired, this value can be checked from inside a batch (.BAT) file or a calling program, to take appropriate further actions. Consult your *DOS* documentation for details on writing BAT files.

Remarks:

Due to its specialized goal, *MATER* uses a search strategy essentially different from the ones used by general chess-playing programs. For once, there's no need for mini-max, alpha-beta pruning, killer heuristics, null move, hash tables, positional scoring, etc., as we're not trying to find a move which has the *best score* in a position. We are only interested in a move that gives *checkmate* as soon as possible, no matter how foolish or bizarre it looks and no matter what sacrifices have to be made, thus no pruning is possible, all legal moves have to be generated and tested, as fast as possible. To that effect, some clever savings and heuristics are possible, let's discuss one.

For instance, at *maximum depth*, there's no need to generate all moves for the mating side, which is a complex procedure and takes time. Since we are at maximum depth, *only* moves that give check need to be generated because moves that don't can't possibly be giving checkmate. Also, there're no need to generate *fully-legal* moves but *pseudo-legal* ones, where "*pseudo*" means that the move *might* be leaving our own king under check (and thus, the move is actually *not* legal). Testing whether our king is under check is costly in terms of time, so generating pseudo-moves is faster and thus we proceed as follows:

At maximum depth we generate one pseudo-move, see if it gives check to the king being mated and if it does we see whether the enemy side has any move to get out of check. If not, then the enemy king will be checkmated *if and only if* our pseudo-move is actually *fully legal*, so then and only then is the legality of the pseudo-move ascertained (by just testing *now* if *our* king is under check), not sooner, saving considerable amounts of time.

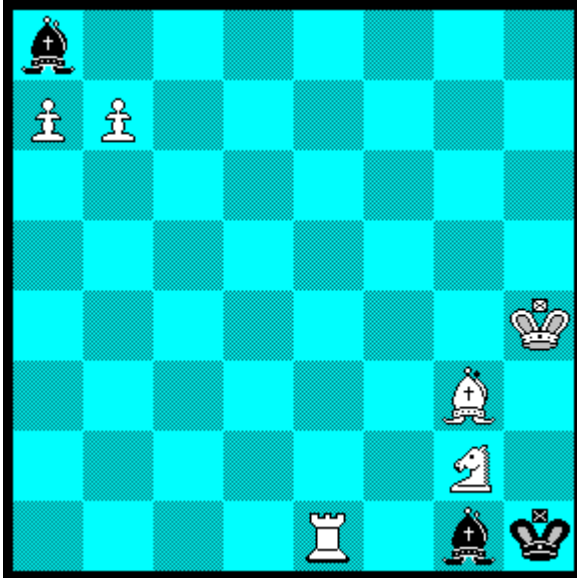
See other differences and a number of useful techniques applied in the commented **Program Listing** below.

3. Examples

Now some examples of *MATER* in action, finding 8 amazing checkmates including rare underpromotions, long sequences of checks, etc. All timings are for an ancient *Hewlett-Packard Vectra/Pentium MMX* at 200 Mhz.

3.1 Example 1

MATER b7/PP6/8/8/7K/6B1/6N1/4R1bk/ w 3 a



```
MATER: Mate searching program v1.1. (c) Valentin Albillo

      a b c d e f g h
# # # # # # # # # # White can't castle long
8 # b - - - - - - - # White can't castle short
7 # P P - - - - - - # Black can't castle long
6 # - - - - - - - - # Black can't castle short
5 # - - - - - - - - # No en passant square
4 # - - - - - - - K #
3 # - - - - - - B - #
2 # - - - - - - N - #
1 # - - - - R - b k #
# # # # # # # # # #

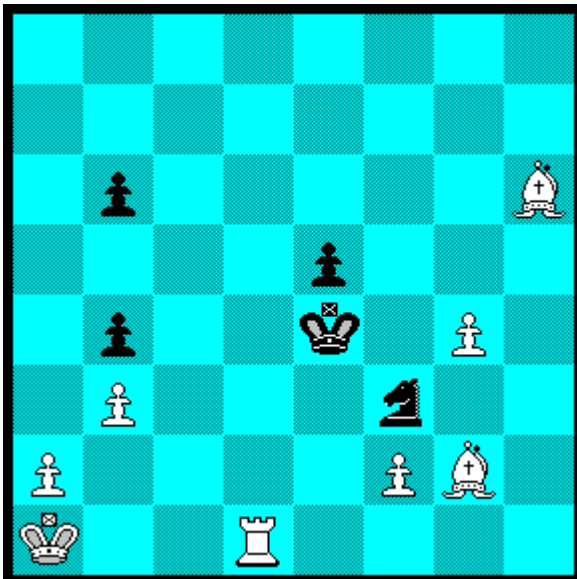
FEN: b7/PP6/8/8/7K/6B1/6N1/4R1bk/ w

Max. no. of moves to mate: 3
Searching *all moves* for: White

Mate in 1.....0.00
Mate in 2.....0.00
Mate in 3.....
Mate in 3 with b7xa8=N, t=0.10, nodes=1978
```

3.2 Example 2

MATER 8/8/1p5B/4p3/1p2k1P1/1P3n2/P4PB1/K2R4/ w 3 a



```
MATER: Mate searching program v1.1. (c) Valentin Albillo

      a b c d e f g h
# # # # # # # # # # White can't castle long
8 # - - - - - - - # White can't castle short
7 # - - - - - - - # Black can't castle long
6 # - p - - - - B # Black can't castle short
5 # - - - - p - - - # No en passant square
4 # - p - - k - P - #
3 # - P - - - n - - #
2 # P - - - - P B - #
1 # K - - R - - - - #
# # # # # # # # # #

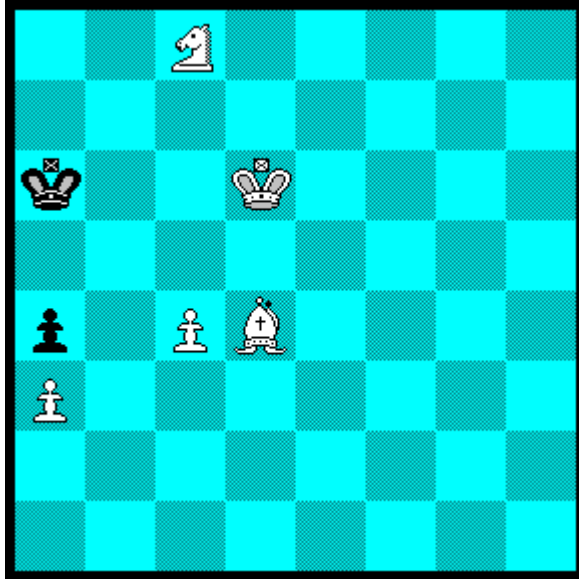
FEN: 8/8/1p5B/4p3/1p2k1P1/1P3n2/P4PB1/K2R4/ w

Max. no. of moves to mate: 3
Searching *all moves* for: White

Mate in 1.....0.00
Mate in 2.....0.00
Mate in 3.....
Mate in 3 with Bh6-c1, t=0.06, nodes=2178
```

3.3 Example 3

MATER 2N5/8/k2K4/8/p1PB4/P7/8/8/ w 4 a



MATER: Mate searching program v1.1. (c) Valentin Albillo

	a	b	c	d	e	f	g	h	
8 #	-	-	N	-	-	-	-	-	# White can't castle long
7 #	-	-	-	-	-	-	-	-	# White can't castle short
6 #	k	-	-	K	-	-	-	-	# Black can't castle long
5 #	-	-	-	-	-	-	-	-	# Black can't castle short
4 #	p	-	P	B	-	-	-	-	# No en passant square
3 #	P	-	-	-	-	-	-	-	#
2 #	-	-	-	-	-	-	-	-	#
1 #	-	-	-	-	-	-	-	-	#
	#	#	#	#	#	#	#	#	#

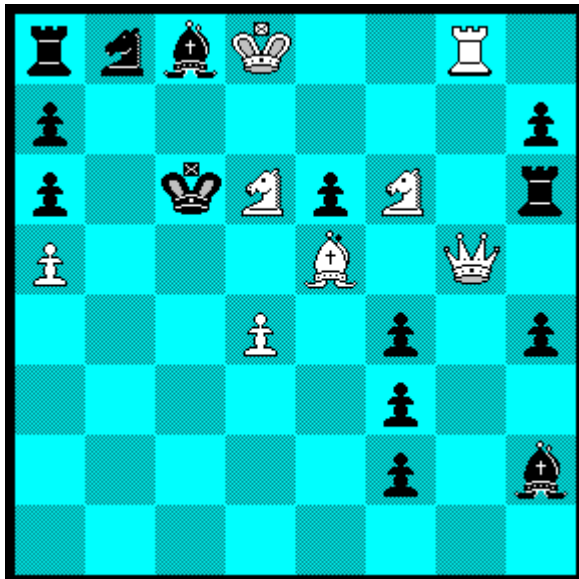
FEN: 2N5/8/k2K4/8/p1PB4/P7/8/8/ w

Max. no. of moves to mate: 4
Searching *all moves* for: White

Mate in 1.....0.00
Mate in 2.....0.00
Mate in 3.....0.11
Mate in 4....
Mate in 4 with Kd6-c7, t=0.57, nodes=15315

3.4 Example 4

MATER rnbK2R1/p6p/p1kNpN1r/P3B1Q1/3P1p1p/5p2/5p1b/8/ w 4 a



MATER: Mate searching program v1.1. (c) Valentin Albillo

	a	b	c	d	e	f	g	h	
8 #	r	n	b	K	-	-	R	-	# White can't castle long
7 #	p	-	-	-	-	-	p	# White can't castle short	
6 #	p	-	k	N	p	N	-	r	# Black can't castle long
5 #	P	-	-	-	B	-	Q	-	# Black can't castle short
4 #	-	-	-	P	-	p	-	p	# No en passant square
3 #	-	-	-	-	-	p	-	-	#
2 #	-	-	-	-	-	p	-	b	#
1 #	-	-	-	-	-	-	-	-	#
	#	#	#	#	#	#	#	#	#

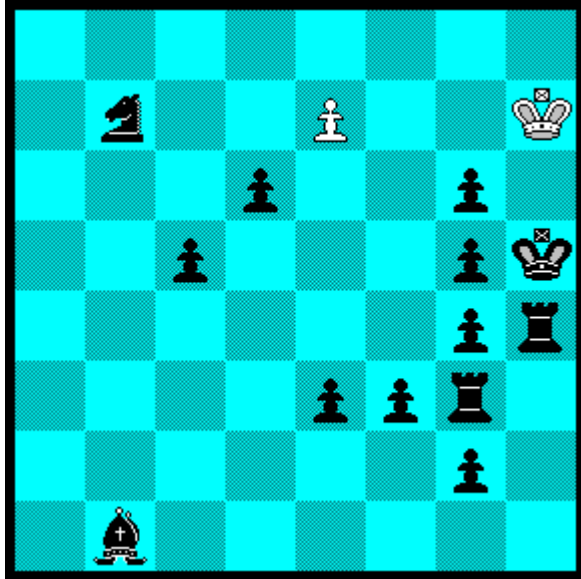
FEN: rnbK2R1/p6p/p1kNpN1r/P3B1Q1/3P1p1p/5p2/5p1b/8/ w

Max. no. of moves to mate: 4
Searching *all moves* for: White

Mate in 1.....0.00
Mate in 2.....0.04
Mate in 3.....2.50
Mate in 4.....
Mate in 4 with Qg5-g2, t=67.01, nodes=1203866

3.5 Example 5

MATER 8/1n2P2K/3p2p1/2p3pk/6pr/4ppr1/6p1/1b6/ w 3 a



MATER: Mate searching program v1.1. (c) Valentin Albillo

	a	b	c	d	e	f	g	h	
8 #	-	-	-	-	-	-	-	-	# White can't castle long
7 #	-	n	-	-	P	-	-	K	# White can't castle short
6 #	-	-	-	p	-	-	p	-	# Black can't castle long
5 #	-	-	p	-	-	-	p	k	# Black can't castle short
4 #	-	-	-	-	-	-	p	r	# No en passant square
3 #	-	-	-	-	p	p	r	-	#
2 #	-	-	-	-	-	-	p	-	#
1 #	-	b	-	-	-	-	-	-	#
	#	#	#	#	#	#	#	#	#

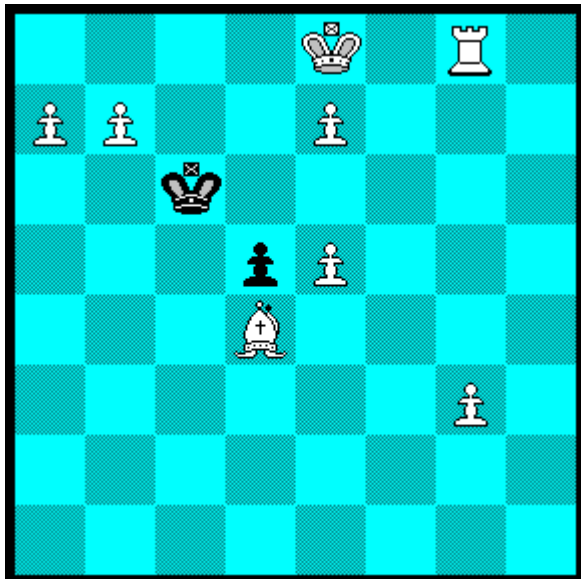
FEN: 8/1n2P2K/3p2p1/2p3pk/6pr/4ppr1/6p1/1b6/ w

Max. no. of moves to mate: 3
 Searching *all moves* for: White

Mate in 1.....0.00
 Mate in 2.....0.00
 Mate in 3.....
Mate in 3 with Kh7-g7, t=0.15, nodes=1170

3.6 Example 6

MATER 4K1R1/PP2P3/2k5/3pP3/3B4/6P1/8/8/ w 3 a



MATER: Mate searching program v1.1. (c) Valentin Albillo

	a	b	c	d	e	f	g	h	
8 #	-	-	-	-	K	-	R	-	# White can't castle long
7 #	P	P	-	-	P	-	-	-	# White can't castle short
6 #	-	-	k	-	-	-	-	-	# Black can't castle long
5 #	-	-	-	p	P	-	-	-	# Black can't castle short
4 #	-	-	-	B	-	-	-	-	# No en passant square
3 #	-	-	-	-	-	-	P	-	#
2 #	-	-	-	-	-	-	-	-	#
1 #	-	-	-	-	-	-	-	-	#
	#	#	#	#	#	#	#	#	#

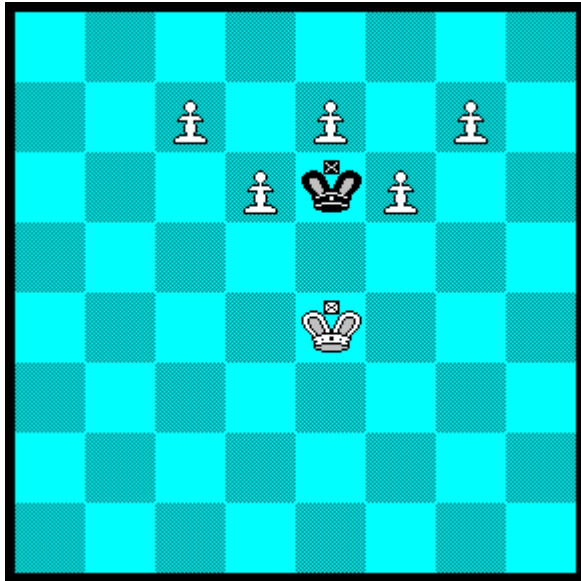
FEN: 4K1R1/PP2P3/2k5/3pP3/3B4/6P1/8/8/ w

Max. no. of moves to mate: 3
 Searching *all moves* for: White

Mate in 1.....0.00
 Mate in 2.....0.00
 Mate in 3.....
Mate in 3 with b7-b8=R, t=0.20, nodes=4805

3.7 Example 7 { several underpromotions are involved here ...}

MATER 8/2P1P1P1/3PkP2/8/4K3/8/8/ w 3 a



```

MATER: Mate searching program v1.1. (c) Valentin Albillo

      a b c d e f g h
# # # # # # # # # # White can't castle long
8 # - - - - - - - # White can't castle short
7 # - - P - P - P - # Black can't castle long
6 # - - - P k P - - # Black can't castle short
5 # - - - - - - - # No en passant square
4 # - - - - K - - - #
3 # - - - - - - - #
2 # - - - - - - - #
1 # - - - - - - - #
# # # # # # # # # #

FEN: 8/2P1P1P1/3PkP2/8/4K3/8/8/ w

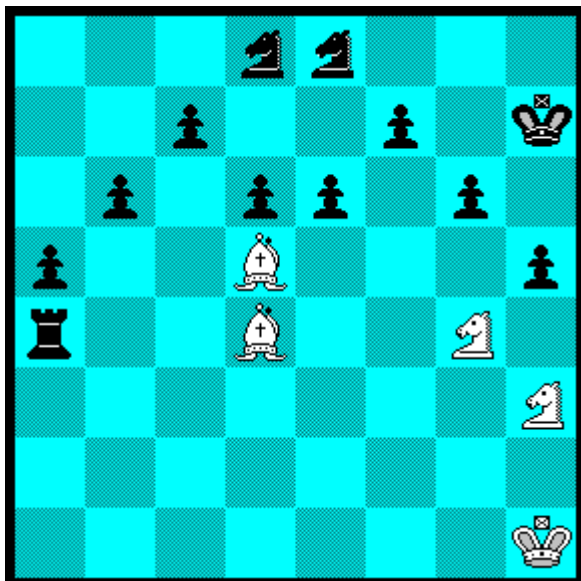
Max. no. of moves to mate: 3
Searching *all moves* for: White

Mate in 1.....0.00
Mate in 2.....0.00
Mate in 3.....
Mate in 3 with e7-e8=B, t=0.13, nodes=2288

```

3.8 Example 8 { here the black king is checkmated after a long sequence of checks ...}

MATER 3nn3/2p2p1k/1p1pp1p1/p2B3p/r2B2N1/7N/8/7K/ w 12 c



```

MATER: Mate searching program v1.1. (c) Valentin Albillo

      a b c d e f g h
# # # # # # # # # # White can't castle long
8 # - - - n n - - - # White can't castle short
7 # - - p - - p - k # Black can't castle long
6 # - p - p p - p - # Black can't castle short
5 # p - - B - - - p # No en passant square
4 # r - - B - - N - #
3 # - - - - - - N #
2 # - - - - - - - #
1 # - - - - - - K #
# # # # # # # # # #

FEN: 3nn3/2p2p1k/1p1pp1p1/p2B3p/r2B2N1/7N/8/7K/ w

Max. no. of moves to mate: 12
Searching only checks for: White

Mate in 1.....0.00
Mate in 2.....0.00
Mate in 3.....0.00
Mate in 4.....0.00
Mate in 5.....0.00
Mate in 6.....0.01
Mate in 7.....0.03
Mate in 8.....0.06
Mate in 9.....0.11
Mate in 10.....0.20
Mate in 11.....0.38
Mate in 12.....
Mate in 12 with Nh3-g5, t=0.40, nodes=9342

```


4. Program Listing

This is the *Turbo Pascal* source code for *MATER* (42 Kb), which can be compiled with version 7.0 or higher of *Turbo Pascal* or *Borland Pascal* to produce an executable program. Can also be easily adapted to compile with *Borland Delphi* and other Pascal implementations (and even to run under *MS Visual Basic for DOS/Windows*).

Disclaimer:

Please note that apart from making freely available the source code that follows and heavily commenting the source code and documenting the executable (including examples), I cannot give any further support so don't ask me any questions about this software, or ask for advice on how to integrate it with your own programs, etc. I simply do not have the time. Thanks for understanding.

Source code:

```
{ $A+,B-,D-,N-,E-,F-,G+,I-,L-,R-,S-,V-,M 65520,0,655360 }

(* this previous line are optimizing compiler directives *)

(* ***** *)
(* *)
(* MATER: Mate searching program - (c) Valentin Albillo 1998 *)
(* *)
(* This program or parts thereof can be used for any purpose *)
(* whatsoever as long as proper credit is given to the copyright *)
(* holder. Absolutely no guarantees given, no liabilities of any *)
(* kind accepted. Use at your own risk. Your using this code in *)
(* all or in part does indicate your acceptance of these terms. *)
(* *)
(* ***** *)

program mater;

Uses Crt, Dos;

type strg80 = string[80]; strg2 = string[2]; strg1 = string[1];
    strg20 = string[20]; strg3 = string[3];

const maxmov = 200; maxpcs = 16; NA = true; FullLegal = true;

    (* definition of pieces and other constants *)

    WhitePawn = 1; BlackPawn = -1; Pawn = 1; Blank = 0;
    WhiteKnight = 2; BlackKnight = -2; Knight = 2; Out = 7;
    WhiteBishop = 3; BlackBishop = -3; Bishop = 3; White = 1;
    WhiteRook = 4; BlackRook = -4; Rook = 4; Black = -1;
    WhiteQueen = 5; BlackQueen = -5; Queen = 5; None = 0;
    WhiteKing = 6; BlackKing = -6; King = 6;
    Top = 22; Bot = 99;

    (* classes of every possible move: captures, castling, en passant *)

    tCAP = -1; tANY = 1;
    tPKN = Knight; tPBI = Bishop; tPRK = Rook; tPQN = Queen;
    tPAS = 8; tOO = 6; tOOO = 7;
    vOO = 50; vOOO = 30;
```

```

type
  Squares = set of 1..120;

  TArrayBoolean = array[Black..White] of boolean;
  TArraySquares = array[Black..White] of Squares;

  (* type for Position variables: squares plus status *)

  tPosit = record
    Board: array[1..120] of integer;
    KingCastle: TArrayBoolean;
    QueenRookCastle: TArrayBoolean;
    KingRookCastle: TArrayBoolean;
    EnPassantSquare: integer;
  end;

  (* type for Move variables: origin, destination, class, value *)

  tMove = record
    SqFrom: integer;
    SqTo: integer;
    MoveClass: integer;
    MoveVal: integer;
  end;

  tMoves = array[1..maxmov] of tMove;

  tPieces = array[1..maxpcs] of integer;

  tsPieces = array[Black..White] of record
    pk: integer;
    nfig: integer;
    Posi: tPieces;
  end;

const
  (* values of the different pieces, normal or promoted *)

  vBLNK = 0;
  vPW = 100; vKN = 300; vpKN = 200; vBI = 300; vpBI = 200;
  vRK = 500; vpRK = 400; vQN = 900; vpQN = 800; vKI = 9999;

  name: array[Pawn..King] of strgl = ('', 'N', 'B', 'R', 'Q', 'K');

  value: array[BlackKing..WhiteKing] of integer
    = (vKI, vQN, vRK, vBI, vKN, vPW, vBLNK, vPW, vKN, vBI, vRK, vQN, vKI);

  (* printed representations of the pieces, empty squares, out board *)

  Shape: array[BlackKing..Out] of char
    = ('k', 'q', 'r', 'b', 'n', 'p', #250, 'P', 'N', 'B', 'R', 'Q', 'K', '#');

  vDIRPWN = 10;

  (* direction offsets available when moving each piece *)

  DirPawn: array[1..3] of integer = ( 10,  9, 11);
  DirKnight: array[1..8] of integer = (-21, -19, -12, -8,  8, 12, 19, 21);
  DirBishop: array[1..4] of integer = (-11, -9,  9, 11);
  DirRook: array[1..4] of integer = (-10, -1,  1, 10);
  DirQueen: array[1..8] of integer = (-11, -10, -9, -1,  1,  9, 10, 11);
  DirKing: array[1..8] of integer = (-11, -10, -9, -1,  1,  9, 10, 11);

```

```

(* promotion squares, first rows, en passant *)

sqpromo: TArraySquares = ([Bot-7..Bot], [], [Top..Top+7]);
sqprime: TArraySquares = ([32..39] , [], [82..89]);
sqpassc: TArraySquares = ([72..79] , [], [42..49]);

sqRooksq: array[Black..White] of integer = (Top, None, Bot-7);
sqRooksk: array[Black..White] of integer = (Top+7, None, Bot);

(* empty board Position, including borders and status *)

ZeroPosit: tPosit = (Board:

( Out, Out, Out, Out, Out, Out, Out, Out, Out, Out,
Out, Out, Out, Out, Out, Out, Out, Out, Out, Out,
Out, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Out,
Out, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Out,
Out, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Out,
Out, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Out,
Out, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Out,
Out, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Out,
Out, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Out,
Out, Out, Out, Out, Out, Out, Out, Out, Out, Out,
Out, Out, Out, Out, Out, Out, Out, Out, Out, Out);

KingCastle: (false,NA,false);
QueenRookCastle: (false,NA,false);
KingRookCastle: (false,NA,false);
EnPassantSquare: None);

(***** assorted main program variables *****)

var f,flag,kpallmov: boolean;
Posit: tPosit;
i,j,k,l,m,n,p,q,dum,Turn: integer;
xmov: tMove;
a,b,c: char;
t1,t2: real;
Nodes: longint;
kpmaxmov,cod: integer;
Device: Text;

(***** procedures and functions *****)

(* converts a string up to 80 characters to uppercase *)

FUNCTION upc(s: strg80): strg80;
var i: integer;
begin
for i:=1 to length(s) do s[i]:=upcase(s[i]); upc:=s;
end;

(* returns the actual time as a number of seconds plus hundredths *)

FUNCTION time: real;
var h,m,s,c: word;
begin
gettime(h,m,s,c); time:=3600.0*h+60.0*m+s+c/100.0;
end;

```

```

(*)
  converts internal square references to algebraic notation for printing or displaying moves
*)

FUNCTION sq2al(n: integer): strg2;
begin
  sq2al:=chr(n mod 10+63+32)+chr(58-n div 10);
end;

(* prints the position: board, castling rights, en passant square *)

PROCEDURE PrintBoard;
var i: integer;
begin
  writeln(Device, '   a b c d e f g h');
  write(Device, ' ');

  with Posit do
  begin
    for i:=11 to 110 do
      begin

        write(Device, ' ' + Shape[Board[i]] + ' ');

      case i of
        20: if KingCastle[White] and QueenRookCastle[White]
            then write(Device, ' White can castle long')
            else write(Device, ' White can''t castle long');

        30: if KingCastle[White] and KingRookCastle[White]
            then write(Device, ' White can castle short')
            else write(Device, ' White can''t castle short');

        40: if KingCastle[Black] and QueenRookCastle[Black]
            then write(Device, ' Black can castle long')
            else write(Device, ' Black can''t castle long');

        50: if KingCastle[Black] and KingRookCastle[Black]
            then write(Device, ' Black can castle short')
            else write(Device, ' Black can''t castle short');

        60: if EnPassantSquare<>None
            then write(Device, ' En passant square: ',EnPassantSquare)
            else write(Device, ' No en passant square');

      end;

      if i mod 10 = 0 then
      begin
        writeln(Device);

        case i of
          20, 30, 40, 50,
          60, 70, 80, 90: write(Device, 10 - i div 10);
          100: write(Device, ' ');
        end;

      end;

    end;

  end;

  writeln(Device);
end;

```

```

end;

(*)
converts a string representing a FEN position to an internal representation.
Returns true if the FEN notation is correct, false otherwise.

Illegalities checked include: more than one king per side or none, more than 8 pawns per side,
more than 16 pieces per side, more or less than 8 rows defined, more or less than 8 squares
per row defined, pieces other than p,n,b,r,q,k.

The castling rights are automatically assigned depending on rooks and king positions.
No en passant square is considered.
*)

FUNCTION FEN2Posit(fen: strg80): boolean;

const kMAXFIL = 8; kMAXPIECES = 16; kKINGS = 1; kMAXPAWNS = 8;

label sig, fenerr;

var c: char;
    p,i,q,n,nwk,nbk,nwp,nbp,nb,nw,nfil: integer;

begin

with Posit do
begin
p:=1; i:=pred(Top); q:=0; nwk:=0; nbk:=0;
nwp:=0; nbp:=0; nb:=0; nw:=0; nfil:=0;
sig:
if p>length(fen) then goto fenerr;
c:=fen[p];

case upcase(c) of
'P','N','B','R','Q','K':
begin
if q=kMAXFIL then goto fenerr;
inc(i);

case c of
'p': begin Board[i]:=BlackPawn; inc(nb); inc(nbp); end;
'n': begin Board[i]:=BlackKnight; inc(nb); end;
'b': begin Board[i]:=BlackBishop; inc(nb); end;
'r': begin Board[i]:=BlackRook; inc(nb); end;
'q': begin Board[i]:=BlackQueen; inc(nb); end;
'k': begin Board[i]:=BlackKing; inc(nb); inc(nbk); end;
'P': begin Board[i]:=WhitePawn; inc(nw); inc(nwp); end;
'N': begin Board[i]:=WhiteKnight; inc(nw); end;
'B': begin Board[i]:=WhiteBishop; inc(nw); end;
'R': begin Board[i]:=WhiteRook; inc(nw); end;
'Q': begin Board[i]:=WhiteQueen; inc(nw); end;
'K': begin Board[i]:=WhiteKing; inc(nw); inc(nwk); end;
end;

inc(q); inc(p); goto sig;
end;

'1'..'8':
begin
n:=ord(c)-ord('0'); inc(q,n);
if q>kMAXFIL then goto fenerr;
for j:=1 to n do Board[i+j]:=Blank;

```

```

    inc(i,j); inc(p); goto sig;
end;

'/:
begin
    if q<>kMAXFIL then goto fenerr;
    inc(i,2); q:=0; inc(p); inc(nfil);
    if nfil<kMAXFIL then goto sig;
    if p<=length(fen) then goto fenerr;
end;

else
begin
    goto fenerr;
end;

end;

(* error if there's an illegal number of Kings, Pawns, or pieces *)

if (nbk<>kKINGS) or (nwk<>kKINGS) then goto fenerr;
if (nbp>kMAXPAWNS) or (nwp>kMAXPAWNS) then goto fenerr;
if (nb>kMAXPIECES) or (nw>kMAXPIECES) then goto fenerr;

(*
    automatically attempt to assign castling rights based upon the position of Kings and Rooks
*)

if Board[26]=BlackKing then KingCastle[Black]:=true else KingCastle[Black]:=false;

if Board[Top]=BlackRook then QueenRookCastle[Black]:=true
    else QueenRookCastle[Black]:=false;

if Board[29]=BlackRook then KingRookCastle[Black]:=true else KingRookCastle[Black]:=false;

if Board[96]=WhiteKing then KingCastle[White]:=true else KingCastle[White]:=false;

if Board[92]=WhiteRook then QueenRookCastle[White]:=true
    else QueenRookCastle[White]:=false;

if Board[Bot]=WhiteRook then KingRookCastle[White]:=true
    else KingRookCastle[White]:=false;

(*
    in this version, no en passant square is assigned for a given position
*)

EnPassantSquare:=None;

end;

FEN2Posit:=true;
exit;

fenerr:

    FEN2Posit:=false;

end;

(*
    returns a string representing the class of a move: promotion and to which piece,

```

```

    castling short, castling long, en passant capture, none of these, unknown
*)

FUNCTION cl2(m: integer): strg3;
var s: strg3;
begin

    case abs(m) of
        tANY: s:='';      (* nothing special *)
        tPKN: s:='N';    (* subpromotion to a knight *)
        tPBI: s:='B';    (* subpromotion to a bishop *)
        tPRK: s:='R';    (* subpromotion to a rook *)
        tPQN: s:='Q';    (* promotion to a queen *)
        tOO: s:=' OO';   (* castling short *)
        tOOO: s:=' OOO'; (* castling long *)
        tPAS: s:=' e.p'; (* en passant capture *)
        else s:=' ???';  (* unknown; should never happen *)
    end;

    cl2:=s;
end;

(* returns a string representing textually a move (i.e: b7xc8=N+) *)

FUNCTION PrintMove(var m: tMove): strg20;
begin
    with m do
        if (SqFrom=0) and (SqTo=0) then (* should never happen *)
            PrintMove:='(no move) '
        else
            if MoveClass<0 then (* it's a capture *)

                PrintMove:=name[Posit.Board[SqFrom]] + sq2al(SqFrom) + 'x' + sq2al(SqTo) + cl2(MoveClass)

            else (* it's not a capture *)

                PrintMove:=name[Posit.Board[SqFrom]] + sq2al(SqFrom) + '-' + sq2al(SqTo) + cl2(MoveClass);
            end;
        end;
    end;

    (*
    fills a record with number, types, and positions of all the pieces for a given side,
    special provision for the king
    *)

PROCEDURE PosPieces(var sPieces: tsPieces);
var i,c: integer;
begin
    fillchar(sPieces,sizeof(sPieces),0);

    with Posit do for i:=Top to Bot do (* scan the whole board *)
        begin
            c:=Board[i];
            if c<>Blank then if c<>Out then
                if c<0 then (* it's a Black piece *)
                    with sPieces[Black] do
                        begin
                            inc(nfig); Posi[nfig]:=i; if c=BlackKing then pk:=i;
                        end
                    else (* it's a White piece *)
                        with sPieces[White] do
                            begin
                                inc(nfig); Posi[nfig]:=i; if c=WhiteKing then pk:=i;
                            end
                        end;
                end;
            end;
        end;
    end;
end;

```

```

        end;
    end;
end;

(* returns true if a given side's king is in check, false otherwise *)

FUNCTION InCheck(Color,pkm,pke: integer): boolean;
var i,fig,cfig,cbi,crk,cqn,sq,c,d,ncolo: integer;
begin

    InCheck:=true; (* we assume beforehand it is in check *)

    (* first we test if it is near the enemy king *)

    case abs(pkm-pke) of
        1,9..11: exit;
    end;

    ncolo:=-Color; cbi:=ncolo*Bishop; crk:=ncolo*Rook; cqn:=ncolo*Queen;

    with Posit do
        begin

            for i:=1 to 4 do
                begin

                    (* test if it is under the attack of a Bishop or a Queen *)

                    d:=DirBishop[i]; sq:=pkm;

                    repeat
                        inc(sq,d); c:=Board[sq];
                    until c<>Blank;

                    if c=cbi then exit else if c=cqn then exit;

                    (* test if it is under the attack of a Rook or a Queen *)

                    d:=DirRook[i]; sq:=pkm;

                    repeat
                        inc(sq,d); c:=Board[sq];
                    until c<>Blank;

                    if c=crk then exit else if c=cqn then exit;
                end;

                (* test if it is under the attack of a Knight *)

                cfig:=ncolo*Knight;
                for i:=1 to 8 do if Board[pkm+DirKnight[i]]=cfig then exit;

                (* test if it is under the attack of a Pawn *)

                cfig:=ncolo*Pawn;
                for i:=2 to 3 do if Board[pkm+ncolo*DirPawn[i]]=cfig then exit;

            end;

            (* the King is not in check *)

            InCheck:=false;
        end;
    end;
end;

```



```

end;

(*
generates moves for a given side. Options permit generation of all moves or a single move,
full legal moves or pseudo-legal moves (own King may be left under check), etc.
All chess rules are implemented, including all five rules for legal castling, promotions,
underpromotions and en passant captures
*)

PROCEDURE GenerateMoves(Color: integer;
                        sq: integer;
                        var Movesf: tMoves;
                        var nMovesf: integer;
                        pkm,pke: integer;
                        legal: boolean;
                        single: boolean;
                        var Found: boolean);

var sq2,c,fig,cfig,
    i,d,ncolo: integer;
    Posit2: tPosit;
    v,cap: integer;

(* tests the pseudo-legal move for full legality and records it *)

PROCEDURE TestRecordMove(cfig,clas,vlm: integer);
begin
  if legal then
    begin
      Posit2:=Posit;

      with Posit do
        begin

          (* we try tentatively the pseudo-legal move *)

          Board[sq2]:=cfig; Board[sq]:=Blank;
          if clas=-tPAS then Board[sq2+vDIRPWN*Color]:=Blank;

          (* if then our King is left in check, it's not legal *)

          if InCheck(Color,pkm,pke) then
            begin
              Posit:=Posit2; exit;
            end;

          if single then
            begin
              Found:=true; Posit:=Posit2; exit;
            end;

          end;
          Posit:=Posit2;
        end;
      end;

      (*
      either it was legal or we specified pseudo-legal moves. Now we record the move in the array
      *)

      inc(nMovesf);
    end;
  end;
end;

```

```

with Movesf[nMovesf] do
begin
  SqFrom:=sq; SqTo:=sq2; MoveClass:=clas; MoveVal:=vlm;
end;

end;

(*
tests the pseudo-legal pawn move for full legality and records it.
Takes proper care of promotions and subpromotions.
*)

PROCEDURE TestRecordPawn;
begin
  v:=value[abs(c)];
  if v=0 then cap:=tANY else cap:=tCAP;

  (* if the pawn promotes, test the 4 possible (sub)promotions *)

  if sq2 in sqpromo[Color] then
begin
  TestRecordMove(Color*Queen ,tPQN*cap,v+vQN); if Found then exit;
  TestRecordMove(Color*Rook ,tPRK*cap,v+vRK); if Found then exit;
  TestRecordMove(Color*Bishop,tPBI*cap,v+vBI); if Found then exit;
  TestRecordMove(Color*Knight,tPKN*cap,v+vKN); if Found then exit;
end
else
begin
  TestRecordMove(Pawn, cap, v); if Found then exit;
end;
end;

(* tests if castling (long or short) is legal *)

PROCEDURE TestCastling;
var i: integer;

(*
Legal castling requirements:

1) the King hasn't moved
2) the appropriate Rook hasn't moved
3) the squares between the King and the Rook are empty
3) the King isn't in check at the moment
4) the square over which the King must pass isn't under attack
5) the square where the King will be placed isn't under attack
*)
label sig;
begin
with Posit do
begin

  if not KingCastle[Color] then exit; (* the King has moved *)
  pkm:=sq;

  (* test castle short *)

  if KingRookCastle[Color] then
begin
  for i:=succ(pkm) to pkm+2 do if Board[i]<>Blank then goto sig;
  if InCheck(Color,pkm,pke) then exit;

```

```

    for i:=succ(pkm) to pkm+2 do
      if InCheck(Color,i,pke) then goto sig;

    if single then begin Found:=true; exit; end;
    inc(nMovesf);

    (* Ok, record short castling *)

    with Movesf[nMovesf] do
      begin
        SqFrom:=sq; SqTo:=pkm+2; MoveClass:=tOO; MoveVal:=vOO;
      end;

    end;

    (* test castle long *)
sig:   if QueenRookCastle[Color] then
      begin
        for i:=pkm-3 to pred(pkm) do if Board[i]<>Blank then exit;
        if InCheck(Color,pkm,pke) then exit;

        for i:=pkm-2 to pred(pkm) do
          if InCheck(Color,i,pke) then exit;

        if single then begin Found:=true; exit; end;

        (* Ok, record long castling *)

        inc(nMovesf);

        with Movesf[nMovesf] do
          begin
            SqFrom:=sq; SqTo:=pkm-2; MoveClass:=tOOO; MoveVal:=vOOO;
          end;

        end;
      end;
    end; (* TestCastling *)

begin (* GenerateMoves *)
  Found:=false; inc(Nodes);

  with Posit do
    begin
      cfig:=Board[sq]; ncolo:=-Color; nMovesf:=None;

      case abs(cfig) of
        Pawn: begin
          d:=ncolo*vDIRPWN; sq2:=sq+d; c:=Board[sq2];

          if c=Blank then (* the pawn advances *)
            begin
              TestRecordPawn; if Found then exit;

              (*
                if the pawn is on his original square it can advance 1 or 2 squares on its move
              *)

              if sq in sqprime[Color] then
                begin

```

```

        inc(sq2,d); c:=Board[sq2];
        if c=Blank then
            begin
                TestRecordPawn; if Found then exit;
            end;
        end;
    end;
end;

(* now we test if the pawn can capture *)

for i:=2 to 3 do
    begin
        sq2:=sq+ncolo*DirPawn[i];

        (* check for an en passant capture *)

        if sq2=EnPassantSquare then
            begin
                if sq2 in sqpassc[Color] then
                    begin
                        TestRecordMove(Pawn,-tPAS,vPW);
                        if Found then exit;
                    end;
                end
            end
        else
            begin
                c:=Board[sq2];
                if c<>Blank then
                    if c<>Out then
                        if c div ncolo>0 then (* the pawn captures *)
                            begin
                                TestRecordPawn; if Found then exit;
                            end;
                        end;
                    end;
                end;
            end;
        end;
    end;
end;

Knight: for i:=1 to 8 do (* eight possible moves for the knight *)
    begin
        sq2:=sq+DirKnight[i]; c:=Board[sq2];
        if c<>Out then
            if c div Color<=0 then
                begin
                    v:=value[abs(c)];
                    if v=0 then cap:=tANY else cap:=tCAP;
                    TestRecordMove(cfig,cap,v); if Found then exit;
                end;
            end;
        end;
    end;

Bishop: for i:=1 to 4 do (* four directions for the Bishop *)
    begin
        sq2:=sq;
        repeat
            inc(sq2,DirBishop[i]); c:=Board[sq2];
            if c<>Out then
                if c div Color<=0 then
                    begin
                        v:=value[abs(c)];
                        if v=0 then cap:=tANY else cap:=tCAP;
                        TestRecordMove(cfig,cap,v); if Found then exit;
                    end;
                end;
            until c<>Blank; (* repeat until blocked *)
        end;
    end;
end;

```

```

end;

Rook: for i:=1 to 4 do (* four directions for the Rook *)
begin
sq2:=sq;
repeat
inc(sq2,DirRook[i]); c:=Board[sq2];
if c<>Out then
if c div Color<=0 then
begin
v:=value[abs(c)];
if v=0 then cap:=tANY else cap:=tCAP;
TestRecordMove(cfig,cap,v); if Found then exit;
end;
until c<>Blank; (* repeat until blocked *)
end;

Queen: for i:=1 to 8 do (* eight directions for the queen *)
begin
sq2:=sq;
repeat
inc(sq2,DirQueen[i]); c:=Board[sq2];
if c<>Out then
if c div Color<=0 then
begin
v:=value[abs(c)];
if v=0 then cap:=tANY else cap:=tCAP;
TestRecordMove(cfig,cap,v); if Found then exit;
end;
until c<>Blank; (* repeat until blocked *)
end;

King: begin
for i:=1 to 8 do (* eight directions for the King *)
begin
sq2:=sq+DirKing[i]; c:=Board[sq2]; pkm:=sq2;
if c<>Out then
if c div Color<=0 then
begin
v:=value[abs(c)];
if v=0 then cap:=tANY else cap:=tCAP;
TestRecordMove(cfig,cap,v); if Found then exit;
end;
end;

(* besides, test if the King can castle *)

TestCastling; if Found then exit;

end;
end; (* case *)

end; (* with Posit *)

end; (* GenerateMoves *)

(*
returns true if a given side has at least one legal move in a given position, false otherwise.
To speed the search in near-mate positions, it considers King's moves first.
*)

FUNCTION AnyMovSide( Color: integer;

```

```

        var sPieces: tsPieces;
            pkm,pke: integer): boolean;

var i,nMovesf: integer;
    Movesf: tMoves;
    Found: boolean;

begin
  with sPieces[Color] do
    begin

      (* first, generate at least one move for the King *)

      GenerateMoves(Color,pkm,Movesf,nMovesf,pkm,pke,FullLegal,true,Found);

      (* the King has at least one legal move. Exit *)

      if Found then begin AnyMovSide:=true; exit; end;

      (*
        the King has no legal moves. Generate at least one move for all pieces but the King
      *)

      for i:=1 to nfig do if Posi[i]<>pkm then
        begin
          GenerateMoves(Color, Posi[i], Movesf, nMovesf, pkm, pke, FullLegal, true, Found);

          (* there's at least one legal move available. Exit *)

          if Found then begin AnyMovSide:=true; exit; end;
        end;

      end;

      (* no legal moves available. The King is either checkmated or stalemated *)

      AnyMovSide:=false;

    end;

  (*
    actually makes a given move in a given position, updating the board and all status:
    castling rights, en passant square, and promotions
  *)

PROCEDURE PerformMove(var xmov: tMove; Color: integer; var pk: integer);
  var c,ncolo: integer;
  begin
    with xmov, Posit do
      begin

        (* update the board *)

        c:=Board[SqFrom]; ncolo:=-Color;
        Board[SqFrom]:=Blank; Board[SqTo]:=c; EnPassantSquare:=None;

        case abs(c) of
          Pawn: begin

            (* check if an en passant capture is now possible *)

            if abs(SqFrom-SqTo)=20 then

```

```

        EnPassantSquare:=(SqFrom+SqTo) div 2;

    (*
    if it was a promotion, replace the pawn with the promoted piece
    *)

    case abs(MoveClass) of
        tPKN,tPBI,
        tPRK,tPQN: Board[SqTo]:=Color*abs(MoveClass);
        tPAS: Board[SqTo+vDIRPWN*Color]:=Blank;
    end;

end;

King: begin
    pk:=SqTo;

    (* the King has moved. Remove castling rights *)

    if KingCastle[Color] then
        begin
            KingCastle[Color]:=false;
            QueenRookCastle[Color]:=false;
            KingRookCastle[Color]:=false;
        end;

    (* if it has castled, move also the Rook *)

    case MoveClass of
        tOO: begin
            Board[pred(SqTo)]:=Color*Rook;
            Board[SqFrom+3]:=Blank;
        end;
        tOOO: begin
            Board[succ(SqTo)]:=Color*Rook;
            Board[SqFrom-4]:=Blank;
        end;
    end;

end;

Rook:
    (* the Rook has moved. Remove its castling right *)

    if SqFrom=sqRooksq[Color] then
        QueenRookCastle[Color]:=false
    else
        if SqFrom=sqRooksk[Color] then KingRookCastle[Color]:=false;
    end;

    (*
    some piece has moved to the Rook's original position. Remove its castling rights
    *)

    if SqTo=sqRooksq[ncolo] then QueenRookCastle[ncolo]:=false else
    if SqTo=sqRooksk[ncolo] then KingRookCastle[ncolo]:=false;

end;

end;
end;

(*
accepts as parameters the position, the side to move, and a maximum number of movements to consider

```

and searches for a move that gives mate in that number of moves or less. If there's such a move, it returns true, else it returns false. Depending on a parameter it can search among all legal moves for the mating side or only checks.

*)

```

FUNCTION FindMate(Color: integer;
                 prof: integer;
                 maxm: integer;
                 var jmov: tMove;
                 onlychk: boolean): boolean;

label nxt,mat;

var x_nMoves,y_nMoves,pkm,
    pke,cfig,ncolo,i,
        j,k,k2,dum: integer;
    sf1,sf2: tsPieces;
    xmov: tMove;
    xMoves,yMoves: tMoves;
    Posit2,Posit3: tPosit;
    prof1,profm,Found,Stalemate: boolean;

begin
    prof1:=prof=1; profm:=prof=maxm; ncolo:=-Color; cfig:=King*Color;

    (* find out the number, positions and types of every piece and in particular the Kings *)

    PosPieces(sf1); pkm:=sf1[Color].pk; pke:=sf1[ncolo].pk;
    Posit2:=Posit;

    with sf1[Color] do
        for k:=1 to nfig do
            begin

                (*
                 generate moves for each piece individually, full legal at all depths except at
                 maximum depth, where only pseudo-legal moves are generated, to save time
                *)

                GenerateMoves(Color, Posi[k], xMoves, x_nMoves, pkm, pke, not profm, false, Found);

                (*
                 at maximum depth, allow for the user to interrupt the search by pressing any key
                *)

                if profm then
                    if keypressed then
                        begin
                            writeln(Device); writeln(Device);
                            writeln(Device, 'HALTED BY USER');
                            close(Device);
                            halt(9999);
                        end;

                (* now perform each move one by one *)

                for i:=1 to x_nMoves do
                    begin
                        xmov:=xMoves[i];

                        (* only at minimum depth, show progress on the screen *)

```



```

if prof1 then write(Device, '.');

(* perform the move *)
PerformMove(xmov,Color,pkm);

if profm then

  (*
   at max depth our move is only pseudo-legal; to save time, we'll test it for full
   legality only if it gives check. If not, it can't possibly give checkmate
   and will be discarded so no need to waste time making sure it's fully legal
  *)

  if InCheck(ncolo,pke,pkm) then
    begin

      (*
       it gives check so: is it fully legal ? if not, discard it and go to next move
      *)

      if InCheck(Color,pkm,pke) then goto nxt;

      (* it was fully legal. If also a capture, we need to obtain a list of pieces *)
      if xmov.MoveClass<0 then PosPieces(sf2) else sf2:=sf1;

      (* now test if the enemy (under check) has a legal move. Else it's checkmate *)
      if AnyMovSide(ncolo,sf2,pke,pkm) then goto nxt;

      goto mat;

    end
  else goto nxt;

  (*
   if we are looking for checkmates in which all the mating side moves are checks,
   we skip moves which do not give check
  *)

  if onlychk then if not InCheck(ncolo,pke,pkm) then goto nxt;

  (* we assume the enemy may be stalemated *)

  Stalemate:=true;
  if xmov.MoveClass<0 then PosPieces(sf2) else sf2:=sf1;

  with sf2[ncolo] do
    for k2:=1 to nfig do
      begin

        (* now we generate moves for the enemy's reply *)

        GenerateMoves(ncolo,Posi[k2],yMoves,y_nMoves,pke,pkm,FullLegal,false,Found);

        (* if there's at least one, there's no stalemate *)

        if y_nMoves<>None then
          begin
            Stalemate:=false; Posit3:=Posit;
          end
        end
      end
    end
  end

```

```

        for j:=1 to y_nMoves do
            begin

                (* perform each enemy reply on the board *)

                PerformMove(yMoves[j],ncolo,dum);

                (* and recursively search for a checkmate in this new position *)

                if not FindMate(Color, succ(prof), maxm, jmov, onlychk) then goto nxt;
                Posit:=Posit3;
                end;

            end;
        end;

        (* if we specified only checks, we've found a checkmate *)

        if onlychk then goto mat;

        (*
            test if the enemy is in check. If so, we've found a checkmate.
            Else, it's stalemated, continue the search
        *)

        if Stalemate then
            if InCheck(ncolo,pke,pkm) then goto mat else goto nxt;
mat:      (* a checkmate was found; set flags and exit *)

            if prof1 then jmov:=xmov;
            FindMate:=true; Posit:=Posit2;
            exit;

nxt:      (* next move; restore the position and iterate again *)

            Posit:=Posit2; pkm:=pk;

            end;
        end;

        (* no checkmate has been found *)

        FindMate:=false;

        end;

        (*
            iteratively calls FindMate for increasing number of moves, and shows the time used
            after each iteration fails to find a checkmate
        *)

        FUNCTION SearchMate(Color: integer;
            minm: integer;
            maxm: integer;
            var nmov: integer;
            var xmov: tMove;
            check: boolean): boolean;

        var i: integer;
        begin
            SearchMate:=false;

```

```

for i:=minm to maxm do
begin
  write(Device, 'Mate in ', i:2);

  if FindMate(Color,1,i,xmov,check) then
  begin
    SearchMate:=true; nmov:=i; exit;
  end;

  (* show the time for each unsuccessful iteration *)

  writeln(Device, time-t1:0:2);
end;
end;

(* outputs some messages, calls SearchMate, and outputs the result *)

PROCEDURE SolveMate;
var  nmov: integer;
     ExitVal: integer;

begin

  n:=kpmaxmov;
  writeln(Device, 'Max. no. of Moves to mate: ',kpmaxmov);

  if kpAllmov then flag:=false else flag:=true;

  if flag then write(Device, 'Searching only checks for: ')
    else write(Device, 'Searching *all Moves* for: ');

  if Turn=White then writeln(Device, 'White') else writeln(Device, 'Black');

  writeln(Device);

  (* go find the checkmate *)

  Nodes:=None;
  t1:=time;
  f:=SearchMate(Turn,1,n,nmov,xmov,flag);
  t2:=time;

  (* output the result of the search *)

  if f then (* checkmate found *)
  begin
    writeln(Device);
    writeln(Device, 'Mate in ',nmov,' with ',PrintMove(xmov)
      ,', t=',t2-t1:0:2,', Nodes=',Nodes);
    ExitVal:=nmov; (* ERRORLEVEL = nmov *)
  end
  else (* checkmate not found *)
  begin
    writeln(Device);
    writeln(Device, 'No mate in ',n,', t=',t2-t1:0:2,', Nodes=',Nodes);
    ExitVal:=0; (* ERRORLEVEL = 0 *)
  end;

  close(Device);
  halt(ExitVal);

```

```

end;

(***** Main program *****)

(*
  outputs the program's identification, copyright, accepts and checks command line parameters,
  outputs error messages for any input errors and calls SolveMate
*)

label entrynok;

begin

  (*
    the following line allows for redirection, sending all output to the standard output,
    either the screen or redirected to a file, the printer, etc.
  *)

  assign(Device, ''); rewrite(Device); writeln(Device);

  (* show name, version, author, and copyright *)

  writeln(Device, 'MATER: Mate searching program v1.1. (c) Valentin Albillo 1998');
  writeln(Device);

  (* check command line parameters *)

  if paramcount<>4 then
    begin
      writeln(Device);
      writeln(Device, 'INPUT ERROR: wrong number of parameters!');
    end;

entrynok:

  writeln(Device);
  writeln(Device, ' Syntax: MATER [FEN posit] [max.mov (1..64)] [all/checks (a/c)]');
  writeln(Device, 'Example: mater b7/PP6/8/8/7K/6B1/6N1/4R1bk/ w 5 a');
  writeln(Device);
  writeln(Device, ' Note: The output can be redirected: MATER ... > mat.txt');
  writeln(Device, ' Exit: - mate in n found: ERRORLEVEL = n');
  writeln(Device, '       - mate not found: ERRORLEVEL = 0');
  writeln(Device, '       - halted by user: ERRORLEVEL = 9999');
  writeln(Device);

  close(Device);
  halt;

end;

val(paramstr(3), kpmxmov, cod);

if (cod<>0) then
  begin
    writeln(Device);
    writeln(Device, 'INPUT ERROR: max.mov should be an integer');
    goto entrynok;
  end;

if (kpmxmov<1) or (kpmxmov>64) then
  begin
    writeln(Device);
    writeln(Device, 'INPUT ERROR: max.mov should be between 1 and 64');
  end;

```

```

        goto entrynok;
    end;

    if upc(paramstr(4))='A' then kpallmov:=true else
    if upc(paramstr(4))='C' then kpallmov:=false else
    begin
        writeln(Device);
        writeln(Device, 'INPUT ERROR: Must specify A (all Moves) or C (checks only)');
        goto entrynok;
    end;

if upc(paramstr(2))='W' then Turn:=White else
    if upc(paramstr(2))='B' then Turn:=Black else
    begin
        writeln(Device);
        writeln(Device, 'INPUT ERROR: Must specify who moves: B (Black) or W (White)');
        goto entrynok;
    end;

    (* reset the position and try to parse the FEN notation *)

    Posit:=ZeroPosit;

    if not FEN2Posit(paramstr(1)) then
    begin
        writeln(Device);
        writeln(Device, 'INPUT ERROR: Illegal FEN position');
        goto entrynok;
    end;

    (* output the board and go search for the checkmate *)

    PrintBoard;
    writeln(Device, 'FEN: ',paramstr(1),' ',paramstr(2)); writeln;
    SolveMate;

    (* that's all, folks *)

end.

```

Copyrights

Copyright for this paper and its contents is retained by the author. Permission to use it for non-profit purposes is granted as long as the contents aren't modified in any way and the copyright is acknowledged.

I am also the copyright holder for all the source code featured here but you can download and use it in all or in part as you wish, including its use as part of your own programs, be they freeware or commercial. The only thing you can't do is claim you wrote it. Credit would be appreciated. No warranty of any kind is given at all. NO SUPPORT OF ANY KIND IS OFFERED. I don't claim the code is suitable or adequate for any purpose whatsoever, nor that it's bug free. I shall not be held responsible for any damage its use can inflict. You use it at your own risk. Your using the code indicates acceptance of these terms. If you are not willing to accept them, do not use the code.