Mate Finder

Generated by Doxygen 1.8.14

Contents

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

Class Documentation

2.1 Board Class Reference

`#include <Board.h>`

Public Member Functions

- Board ()
  Empty constructor, defaulting to the starting position.
- Board (const char *str)
  Read from FEN string.
- Board (const char *str, const bool file)
  Read from FEN notation from file or string and calculate legal moves.
- Piece::Piece getSquare (const square< int > pos) const
  Get what piece is occupying the given square.
- bool isCheck () const
  Get whether the player to move is checked.
- bool blackToMove () const
  Get whether black is to move.
- bool isMate () const
  Get whether the current position is mate.
- bool isDraw () const
  Get whether the current position is drawn.
- moveArray & getMoves ()
  Get the list of legal moves.
- void execMove (const move mv)
  Execute a move.
- void changeColor ()
  Change the color of the player that is to move.
- void clearBoard ()
  Clear the entire board.
- void setPiece (const square< int > sq, const Piece::Piece piece)
  Place a piece on a given square.
- Board cloneAndExecMove (const move mv) const
  Clone the current position and execute a move.
- void printBoard () const
  Print a formatted representation of the board.

Private Types

- enum **stateFlags** : char {
  checkMask = 0x01, **whiteCastleKingsideMask** = 0x02, **whiteCastleQueensideMask** = 0x04, **blackCastleKingsideMask** = 0x08,
  blackCastleQueensideMask = 0x10, **blackToMoveMask** = 0x20, **drawMask** = 0x40 }

Private Member Functions

- int fromStr (const char *str)

  Read the FEN notation from a string.
- int fromFile (const char *fileName)

  Read the FEN notation from a file.
- void calcMoves ()

  Calculate legal moves.
- void getPieceMoves (moveArray &result, const check &kingEnv, const square< int > curPos, const square< int > kingPos)

  Calculate the legal moves of the piece on square<int> curPos.
- void checkDir (moveArray &result, const check &kingEnv, const square< int > basePos, const square< int > dir) const

  Check the possible moves of a piece along some file, rank or diagonal.
- check getCheck (const square< int > kingPos)

  Get the details about a possible check at kingPos.
- bool firstPiece (check &result, const square< int > curPos, const square< int > dir, const int friendlies) const

  Investigate the possibility of attacks from dir to curPos (recursive) with heatmap.
- bool isAttacked (const square< int > piecePos) const

  Check whether the square<int> at piecePos is attacked.
- bool hasAttacker (square< int > pos, const square< int > dir) const

  Invesitgate the possibility of attacks from dir at curPos.
- bool isFriendly (const Piece::Piece piece) const

  Find out if a piece is friendly.
- bool isFriendly (const square< int > pos) const

  Find out if a square is occupied by a friendly piece.
- Board (const Board &other, const move mv)

  Private constructor to circumvent calculating the possible legal moves twice.
- void flipBoard ()

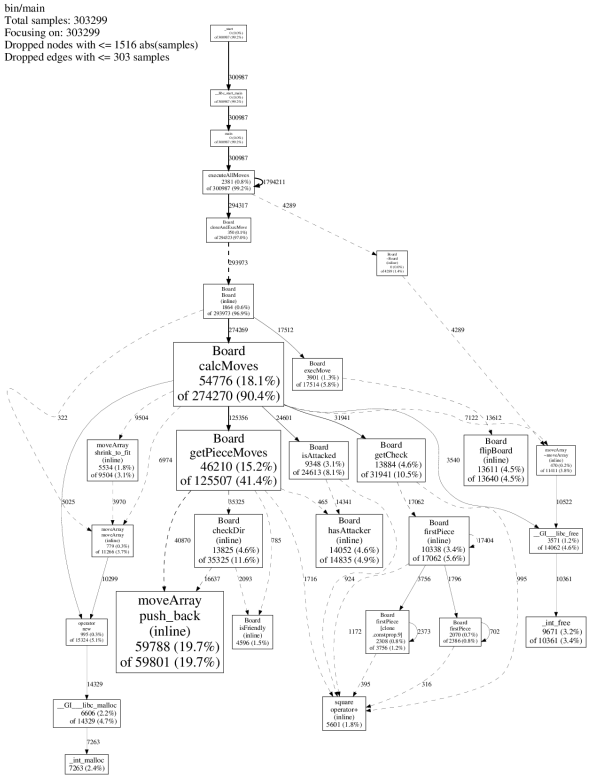
  Flip the board such that pawns always advance in the direction of increasing rank.

Private Attributes

- Piece::Piece board [8][8]
- char state
- signed char enPassant
- moveArray legalMoves

2.1.1 Detailed Description

Objects of the Board class store exactly one position on the chess board. Methods of this class include methods to calculate the all legal moves in the position, and executing a move to obtain a new position. A lot of optimization has gone into the methods of Board and their internal communication. The following flowchart shows the respective contribution of the methods to the total runtime of the program. It was created using google-pprof.



2.1.2 Constructor & Destructor Documentation

2.1.2.1 Board() [1/2]

```
Board::Board (
            const Board & other,
            const move mv )  [private]
```

Private constructor to circumvent calculating the possible legal moves twice.

This constructor is used when a move is executed. All the pieces must be copied, but copying all the legal moves is not necessary, as they are recalculated anyway. So, this private constructur is only to be used internally, when calling the *cloneAndExecMove* function.

Parameters

in	*other*	This is the board that is to be cloned.
in	*mv*	This is the move that is to be executed.

2.1.2.2 Board() [2/2]

```
Board::Board (
            const char * str,
            const bool file )
```

Read from FEN notation from file or string and calculate legal moves.

Board constructor creating Board object from either a file or a string

Parameters

in	*str*	Either the FEN string or the filename
in	*file*	Boolean whether to read file or string (true is file)

2.1.3 Member Function Documentation

2.1.3.1 calcMoves()

```
void Board::calcMoves ( )  [private]
```

Calculate legal moves.

This function calculates all possible moves, and stores them in *legalMoves* member. Additionally, it updates the *check* flag, to indicate whether the player that is to move is check. Finally, it also updates the *draw* flag. If there are too few pieces on the board for any mate to occur, then this flag is set.

2.1.3.2 checkDir()

```
void Board::checkDir (
            moveArray & result,
            const check & kingEnv,
            const square< int > basePos,
            const square< int > dir ) const  [inline], [private]
```

Check the possible moves of a piece along some file, rank or diagonal.

This function helps the getPieceMoves function. It checks to which squares a long range piece (i.e. queen, rook or bishop) can move in a given direction, and appends these moves to the array of legal moves.

Parameters

out	*result*	This is the array of moves which the found legal moves are appended to.
in	*kingEnv*	Through this *check* object, the information concerning whether a given move resolves check is supplied to the function.
in	*basePos*	This is the position of the piece whose legal moves are investigated.
in	*dir*	This is the direction in which the legal moves are being checked.

2.1.3.3 cloneAndExecMove()

```
Board Board::cloneAndExecMove (
            const move mv ) const
```

Clone the current position and execute a move.

Clone the board and execute the given move. Return the resulting position.

Parameters

in	*mv*	The move to be executed.

Returns

This function returns a new Board object, containing the new position, after the move was executed.

2.1.3.4 execMove()

```
void Board::execMove (
            const move move )
```

Execute a move.

Execute a move on a given board and consequently flip the board and recalculate the legal moves.

Parameters

in	*move*	This is the move that is to be executed.

2.1.3.5 firstPiece()

```
bool Board::firstPiece (
            check & result,
            const square< int > curPos,
            const square< int > dir,
            const int friendlies ) const  [private]
```

Investigate the possibility of attacks from dir to curPos (recursive) with heatmap.

This function helps the getCheck function to investigate the influence of one of the 8 directions on the output of the getCheck function.

Parameters

out	*result*	The check struct that getCheck is working on.
in	*curPos*	The current position that is being investigated.
in	*dir*	The direction in which is being checked whether there are attacks going on.
in	*friendlies*	This value determines the number of friendly pieces that were already encountered. If this is 1, and an attacking piece is encountered, then *result* will indicate that this piece is pinned.

Returns

This function returns a boolean. True is returned, if the square indicated by *curPos* is under attack from the direction indicated by *dir*. Otherwise, false is returned.

2.1.3.6 flipBoard()

```
void Board::flipBoard ( )  [private]
```

Flip the board such that pawns always advance in the direction of increasing rank.

This function flips the board. More precisely, it mirrors the position of all pieces along the axis between the 4th and 5th rank. So, for example, a piece on e3 is placed on e6, and a piece on a8 is placed on a1. This function is used to ensure that the player that is to move always plays with its pawns advancing to ever higher ranks.

2.1.3.7 fromFile()

```
int Board::fromFile (
            const char * fileName )  [private]
```

Read the FEN notation from a file.

Initialize a Board object from a file containing a string in FEN notation.

Parameters

fileName	The name of the file.

Returns

This function returns an error code, according to the following table:

Code	Description
0	Success.
1	The syntax of str is wrong.

2.1.3.8 fromStr()

```
int Board::fromStr (
            const char * str )  [private]
```

Read the FEN notation from a string.

Initialize a Board object from a string in FEN notation.

Parameters

in	*str*	The string in FEN notation

Returns

This function returns an error code, according to the following table:

Code	Description
0	Success.
1	The syntax of str is wrong.

2.1.3.9 getCheck()

```
check Board::getCheck (
            const square< int > kingPos )  [private]
```

Get the details about a possible check at kingPos.

This function finds out what is going on in relation to the king of the player that is to move. In particular, it returns a check struct containing information about pinned pieces and squares that can resolve check. Moreover, it updates the check flag.

Parameters

in	*kingPos*	The position of the king.

Returns

This function returns a check object, containing information about pinned pieces and, in case of check, how many pieces are attacking the king, and, in case of 1 attacking piece, which squares will resolve the check.

2.1.3.10 getPieceMoves()

```
void Board::getPieceMoves (
            moveArray & result,
```

```
            const check & kingEnv,
            const square< int > curPos,
            const square< int > kingPos )  [private]
```

Calculate the legal moves of the piece on square<int> curPos.

This function helps the calcMoves function. It investigates the legal moves that one particular piece can make, and appends these to the temporary moves array created by calcMoves.

Parameters

out	*result*	This is the array that the newly found moves are being appended to.
in	*kingEnv*	The function uses this data structure to check whether the piece is pinned, or helps resolving a check.
in	*curPos*	This is the square where the piece, whose legal moves are begin investigated, is located.
in	*kingPos*	This is the square where the king of the player who is to move is located. Its value is used to speed up special cases of taking en passant.

2.1.3.11 hasAttacker()

```
bool Board::hasAttacker (
            square< int > pos,
            const square< int > dir ) const  [private]
```

Invesitgate the possibility of attacks from dir at curPos.

This function helps the isAttacked function to determine whether an attack over a long distance from a fixed direction occurs.

Parameters

in	*pos*	The position of which is determined whether it is under attack.
in	*dir*	The direction in which is being checked whether there is an attacker.

Returns

This function returns a boolean, indicating whether there is an emeny piece attacking the square *pos* from direction *dir*.

2.1.3.12 printBoard()

```
void Board::printBoard ( ) const
```

Print a formatted representation of the board.

Give a nice overview of the current position in the terminal window

2.1.4 Member Data Documentation

2.1.4.1 board

```
Piece::Piece Board::board[8][8]  [private]
```

This character array stores the position of the pieces on the board. Every entry in the character array corresponds to one square. The values of these characters correspond to the widely used FEN notation.

Value	ASCII representation	Piece
0x20	Space	None
0x4B	K	White king
0x51	Q	White queen
0x52	R	White rook
0x42	B	White bishop
0x4E	N	White knight
0x50	P	White pawn
0x6B	k	Black king
0x71	q	Black queen
0x72	r	Black rook
0x62	b	Black bishop
0x6E	n	Black knight
0x70	p	Black pawn

A convenient feature of this notation is that determining whether a piece is black can be done by doing an AND operation with the bitmask 0x20.

2.1.4.2 enPassant

```
signed char Board::enPassant  [private]
```

This signed character stores whether the opponent in the last move advanced one of his pawns by two squares. If he did not, then the value is -1. Otherwise, it is in the range 0-7, where the value corresponds to the file on which the pawn was advanced in the last move.

2.1.4.3 legalMoves

```
moveArray Board::legalMoves  [private]
```

Here, all the legal moves that the player to move can play are stored. We have chosen not to go with the std↩ ::vector<move> implementation, as our custom implementation saves about 15% of runtime.

2.1.4.4 state

```
char Board::state  [private]
```

This character contains seven flags. These flags store some auxiliary information about the position. They are listed in the table below.

Bit	Keyword	Description
0	checkMask	This flag is set if the player to move is checked.
1	whiteCastleKingsideMask	This flag is set if the player to move has not yet moved his king and kingside rook.
2	whiteCastleQueensideMask	This flag is set if the player to move has not yet moved his king and queenside rook.
3	blackCastleKingsideMask	This flag is set if the player that is not to move has not yet moved his king and kingside rook.
4	blackCastleQueensideMask	This flag is set if the player that is not to move has not yet moved his king and queenside rook.
5	blackToMoveMask	This flag is set if black currently is to move.
6	drawMask	This flag is set if there is a stalemate, or if giving checkmate with the pieces that are left on the board is not possible. In both cases, the game is considered to be drawn.

Notice that it is no coincidence that the blackToMoveMask corresponds to the fifth bit. The fifth bit being set in a character yields 0x20, which is exactly the mask that can be used to determine whether a piece is black or white.

The documentation for this class was generated from the following files:

- src/Board.h
- src/Board.cpp

2.2 check Struct Reference

`#include <Board.h>`

Public Attributes

- int len

  The number of pieces attacking the king.
- char heatMap [8][8]

2.2.1 Detailed Description

The check struct stores all the information that is somewhat related to the king, and is used for internal communication within the *calcMoves* method only. On the one hand, it stores the number of pieces that are giving check to the king, and if the king is checked, which squares can be used to resolve this check. Furthermore, it stores whether a piece is pinned, and if so, in which direction.

2.2.2 Member Data Documentation

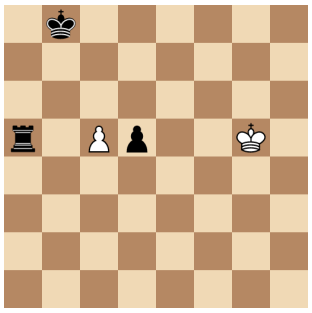
2.2.2.1 heatMap

`char check::heatMap[8][8]`

All the 64 character entries of the heatmap correspond to exactly one square on the chess board. The encoding of the heatmap depends on the piece that is occupying the square. The table below lists all the cases.

Occupation of the square	Value	Description
Empty	0	No peculiarities.
	1	Placing a friendly piece here will resolve check.
Enemy piece	0	No peculiarities.
	1	Taking this piece will resolve check.
Friendly piece	0	No peculiarities.
	1	This piece is pinned, such that it can only move in the northwest-southeast direction.
	2	This piece is pinned, such that it can only move in the horizontal direction.
	3	This piece is pinned, such that it can only move in the northeast-southwest direction.
	4	This piece is pinned, such that it can only move in the vertical direction.

Some edge cases are not included in the check structure. For example, consider the following board:



If white now decides to take the black pawn en passant, then white checks himself, so this is not a legal move. Yet the white pawn is not considered to be pinned, as it can advance without checking anyone. This, then, is a situation that cannot be stored in the check struct, and needs to be handled separately.

The documentation for this struct was generated from the following file:

- src/Board.h

2.3 DFS Class Reference

`#include <DFS.h>`

Public Member Functions

- DFS (Board *_start, unsigned int _maxDepth, bool _turbo)

  The constructor.
- DFS (Board *_start)

  Default constructor, using the default values.
- DFSresult search ()

Private Member Functions

- template<bool T>
  DFSresult best_outcome (Board, unsigned int)

Private Attributes

- unsigned int maxDepth

  The maximum number of half-moves to be executed from the initial position.
- unsigned int curDepth

  The number of half-moves at which the current branches that are under investigation are being pruned.
- Board * start

  A pointer to the Board object storing the initial position.
- bool turbo

  A boolean variable storing whether the search should be conducted using turbo mode, affecting the behavior of the best_outcome method.

2.3.1 Detailed Description

The DFS class performs the Depth First Search through all the possible positions. It allows two possible search modes. One is the standard mode of search. It searches through the entire tree of possible game continuations with a given depth, and returns the best move sequence of all these continuations. The other is referred to as the turbo mode. This mode is aimed at finding checkmates for the player that initially is to move. Using this mode, the algorithm will correctly find forced mate sequences, if they exist, but in other cases it might not identify the best game continuation for the opponent.

2.3.2 Constructor & Destructor Documentation

2.3.2.1 DFS()

```
DFS::DFS (
            Board * _start,
            unsigned int _maxDepth,
            bool _turbo )
```

The constructor.

DFS or depth first search constructor.

Parameters

in	*_start*	Starting board pointer
in	*_maxDepth*	Maximal depth to search.
in	*_turbo*	Turbo mode.

2.3.3 Member Function Documentation

2.3.3.1 best_outcome()

```
template<bool T>
DFSresult DFS::best_outcome (
            Board board,
            unsigned int depth )  [private]
```

Calculate what the best outcome is for a single node. First of all, it is checked whether the current position is a mate or a draw. If this is the case, this is reported to the calling function. Next, all the moves that are possible in the position are investigated one by one. For every move, it is returned to this function whether it results in a win, loss, draw, or an undecided position, and how many moves it takes. This function, then, chooses the best option among all of these moves, and returns that to the calling function. In turbo mode, at odd depths of search (so when the opponent is to move), not necessarily the best move is returned, but just a move that avoids being checkmated, whenever possible.

Parameters

in	*board*	The board to consider
in	*depth*	The current depth

Returns

This function returns a DFSresult object containing the best state possible of the previous node the depth at which the state occurred and the moves leading to that state from this node.

2.3.3.2 search()

```
DFSresult DFS::search ( )
```

Do the actual search. The search is conducted as follows. First of all, all the possible moves are investigated with depth 1. If this does not yield a decisive result, then the depth at which is searched is increased by 2. This is done until the maximum depth is reached. Increasing the depth like this does not increase the total time complexity of the program, and is therefore perfectly suitable for obtaining preliminary results without having to conduct the entire search, while not giving rise to unacceptable amounts of overhead. Only at the maximum depth level, progress reports are being shown.

Returns

This function returns a DFSresult object containing the best state possible from the start board the worstcase depth at which the state occurred and the moves leading to that state from the position.

The documentation for this class was generated from the following files:

- src/DFS.h
- src/DFS.cpp

2.4 DFSresult Struct Reference

`#include <DFS.h>`

Public Attributes

- int state
- unsigned int depth
- std::stack< move > moves

2.4.1 Detailed Description

The DFSresult structure contains the result of the Depth First Search. It stores whether the given position is a win, a loss, a draw, or undecided, and it contains the move sequence the search routine considered best.

2.4.2 Member Data Documentation

2.4.2.1 depth

`unsigned int DFSresult::depth`

The member variable depth stores the length of the resulting stack, so it equals the number of half-moves that the stack contains.

2.4.2.2 moves

`std::stack<move> DFSresult::moves`

The member variable moves stores the best move sequence that the search algorithm came up with. It uses the std::stack implementation.

2.4.2.3 state

`int DFSresult::state`

This is the return state of the search routine. It is encoded as follows.

Value	Description
-2	The position is winning for the player that currently is to move, provided that he plays correctly.
0	The position is a draw.
1	The position is still undecided. There is no forced mate possible within the number of moves that the search algorithm used as its maximum depth.
2	The position is losing for the player that currently is to move, provided that the opponent plays correctly.

The documentation for this struct was generated from the following file:

- src/DFS.h

2.5 move Struct Reference

Public Member Functions

- void **printMove** (bool blackToMove)

Public Attributes

- square< void > **start**
- square< void > **end**
- Piece::Piece **promoteTo**

The documentation for this struct was generated from the following file:

- src/Move.h

2.6 moveArray Struct Reference

Public Member Functions

- **moveArray** (const int n)
- **moveArray** (const moveArray &other)
- moveArray & **operator=** (const moveArray &other)
- moveArray & **operator=** (moveArray &&other)
- move & **operator[ ]** (const int n) const
- int **size** () const
- void **push_back** (const move toAdd)
- moveArray **shrink_to_fit** ()

Public Attributes

- int **num**
- int **ctr**
- move * **moves**

The documentation for this struct was generated from the following file:

- src/MoveArray.h

2.7 square< T > Struct Template Reference

Public Member Functions

- **square** (const T x, const T y)
- template<typename newT >
  square (const square< newT > &other)
- template<typename newT >
  square< typename std::common_type< T, newT >::type > **operator+** (const square< newT > &other) const
- square< T > & **operator+=** (const square< T > &other)
- square< T > **operator+** (const square< void > &other) const
- square< T > & **operator+=** (const square< void > &other)

Public Attributes

- T **x**
- T **y**

The documentation for this struct was generated from the following file:

- src/Square.h

2.8 square< void > Struct Template Reference

Public Member Functions

- **square** (const int x, const int y)
- template<typename newT >
  square (const square< newT > &other)
- template<typename newT >
  square< newT > **operator+** (const square< newT > &other)
- template<typename newT >
  square< void > & **operator+=** (const square< newT > &other)

Public Attributes

- signed char **x**:4
- signed char **y**:4

The documentation for this struct was generated from the following file:

- src/Square.h

Index