Antichess Solver

Alexander Khoma (ak4109), Jakwanul Safin (js5142)

December 2019

Introduction

Suicide chess (or antichess) is a variant of chess where each player's goal is to lose all their pieces. The rules are the same as regular chess except:

1. Capture must be played if any captures are possible.
2. Kings are treated as normal pieces (no checks, checkmates, castling and pawns can promote to king)
3. Stalemates are a win for the stalemated player.

Our program attempts to find the winning move in positions of antichess. Like regular chess, antichess is a 2-player zero sum game, so it can be solved using the minimax algorithm. Our algorithm creates a tree representation of the game states, then runs minimax with increasing depth until it finds a winning position, or reaches a depth limit (which we set to 10). The end goal would be to give the program a set of positions that have a reasonably small number of pieces and see if it is possible to win the game in ≤ 5 moves (to win is to get rid of all your pieces, or get stalemated).

Sample Game

Below is an example game where it is possible for white to win in 4 moves. There is no immediate way to get rid of the knight, but if you push the B pawn up, you force black's A pawn to capture it, after which you can push your C pawn up 2 squares, forcing black to capture, then your king up, forcing black to capture again. Finally play your knight to E2 and force it to be captured, winning the game.

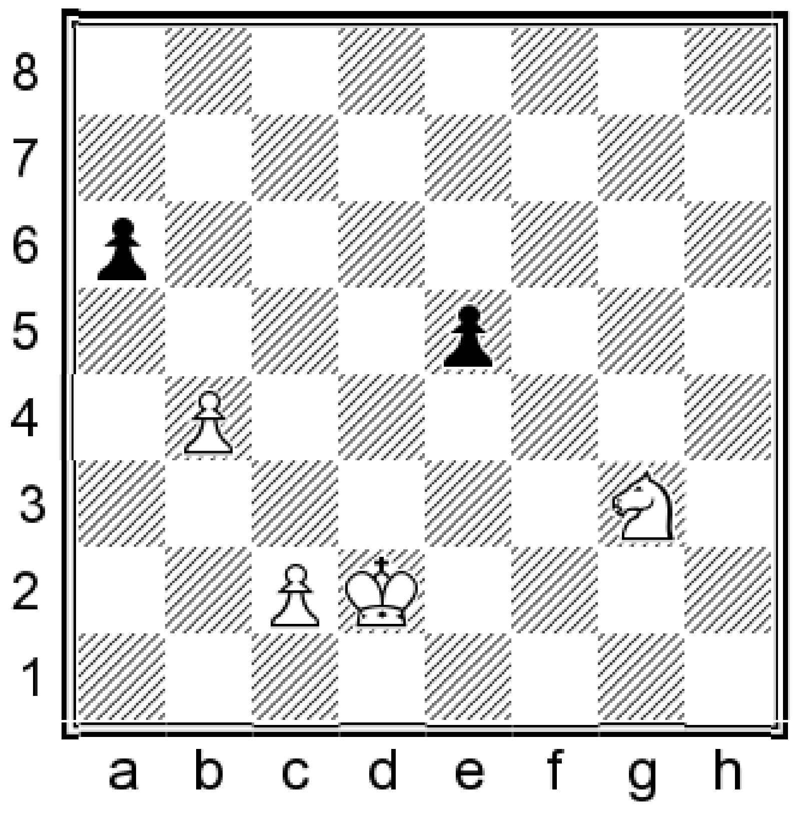

Figure 1: Winning sequence: **1.b5 axb5 2.c4 bxc4 3.Kd3 cxd3 4.Ne2 dxe2#**

This position can be solved in around 0.07 seconds. More complex positions that have many pieces on the board will require a large search depth to solve, and the minimax algorithm's run time is exponential in depth, so it wasn't very practical to keep searching past 5 moves.

Code

Overview

The bulk of the code that we used to for the game logic came from the chesshs library on hackage, written by Arno van Lumig. The library code is in `Chess.hs`, `Chess/Chess.FEN` and `Chess/Chess.PGN`. The chess engine was modified to incorporate the antichess variant's ruleset. And the functions `forcedMove` and `movesFrom` were added in order to efficiently implement our algorithm. `movesFrom` is a faster re-implementation of the default move generator.

`main.hs` becomes the main executable. After compiling, run

```
./main <positions_file> n
```

to evaluate the first `n` positions in the file.

`play.hs` is a second executable that allows the user to interactively play a game of suicide chess. `play` can be run without args or with a fen string as a parameter to create the starting board (sample fen boards can be found in the 'positions' document). Inputing 'best' will run our algorithm with a search deep of 3 moves, and will print out the expectation of winning, positive for white and negative for black, along with the best move.

`Minimax.hs` contains the tree representation for the minimax game problem. The core data structure is a MinimaxTree which is a tree where every node contains a board object, representing the gamestate. Leaves, called 'Final', are final, game over, states which occur when a player has no playable moves. 'Root' is the starting position and all other nodes are 'Partial' data objects. Our algorithms lazily traverse MinimaxTree to find the best moves. `score` is the static heuristic evaluation function that gives an estimate of how good the current position is. `negaMax` is the main algorithm to compute the best scoring move.

`miniMaxWithMoves` runs `negaMax` on all of the children boards from the current position so that it can recover the best move rather than just return the score. `itDeep` is an iterative deepening function that calls `miniMaxWithMoves` with increasing depth until it finds a winning move (or passes a depth limit of 5 moves).

NOTE: Our algorithm actually only solves the boards approximately. We cannot realistically search the entire tree because the branching factor is too large, so we use the evaluation heuristic and only search the 5 best candidate moves at each depth level after the first one. The best moves tend to be ones that give your pieces away, and for all of the positions that we manually tested, our algorithm does find the true best answer (confirmed by the Stockfish engine on lichess.org).

Code Listing

`Chess.hs: forcedCapture`

```haskell
forcedCapture :: Color -> Board -> [((Int, Int), (Int, Int))]
forcedCapture clr brd = filter canCapture pairs
    where
        pieces = piecesOf clr brd
        otherPieces = piecesOf (otherColor clr) brd
        pairs = [(p1,p2) | p1 <- pieces, p2 <- otherPieces]
        canCapture ((x1, y1), (x2, y2)) = okMove x1 y1 x2 y2 brd
```

`Chess.hs: movesFrom`

```haskell
movesFrom :: Int -> Int -> Board -> [(Int, Int, Int, Int)]
movesFrom x y brd
  | isNothing piece          = []
  | clr ownpiece /= turn brd = []
  | otherwise                = movesFrom' ownpiece
    where
      piece = pieceAt x y brd
      ownpiece = fromJust piece
      owncolor = clr ownpiece

      mvFilter mvs
        | null right
            || not (opponentAt x2 y2) = left
        | otherwise                   = head right :[]
        where
          (left, right) = span (\(x2, y2) ->
              isNothing $ pieceAt x2 y2 brd) mvs
          (x2, y2) = head right

      opponentAt x2 y2 = isJust target &&
          clr (fromJust target) /= owncolor
        where target = pieceAt x2 y2 brd

      rookFrom xx yy =
        (mvFilter $ zip [xx-1, xx-2..0] [yy, yy..]) ++
        (mvFilter $ zip [xx+1, xx+2..7] [yy, yy..]) ++
        (mvFilter $ zip [xx, xx..] [yy-1, yy-2..0]) ++
        (mvFilter $ zip [xx, xx..] [yy+1, yy+2..7])

      bishopFrom xx yy =
        (mvFilter $ zip [xx-1, xx-2..0] [yy-1, yy-2..0]) ++
        (mvFilter $ zip [xx+1, xx+2..7] [yy+1, yy+2..7]) ++
        (mvFilter $ zip [xx+1, xx+2..7] [yy-1, yy-2..0]) ++
        (mvFilter $ zip [xx-1, xx-2..0] [yy+1, yy+2..7])
```

```
inBounds x2 y2 = 0 <= x2 && x2 < 8 && 0 <= y2 && y2 < 8

movesFrom' (Piece _ Rook)   = rookFrom x y
movesFrom' (Piece _ Bishop) = bishopFrom x y
movesFrom' (Piece _ Queen)  = rookFrom x y ++
  bishopFrom x y
movesFrom' (Piece _ Knight)
  | null caps = empty
  | otherwise = caps
    where
      locs = filter (uncurry inBounds) [
        (x+2, y+1), (x+1, y+2), (x-2, y+1),
        (x-1, y+2), (x+2, y-1), (x+1, y-2),
        (x-2, y-1), (x-1, y-2) ]
      caps = filter (\(x2, y2) -> opponentAt x2 y2) locs
      empty = filter (\(x2, y2) -> isNothing $
        pieceAt x2 y2 brd) locs

movesFrom' (Piece _ King)
  | null caps = empty
  | otherwise = caps
    where
      locs = filter (uncurry inBounds) [
        (x+1, y+1), (x, y+1), (x-1, y+1),
        (x+1, y), (x-1, y), (x+1, y-1),
        (x, y-1), (x-1, y-1)]
      caps = filter (\(x2, y2) -> opponentAt x2 y2) locs
      empty = filter (\(x2, y2) -> isNothing $
        pieceAt x2 y2 brd) locs

movesFrom' (Piece White Pawn)
  | not $ null captures = captures
  | y == 7 || isJust (pieceAt x (y + 1) brd) = []
  | y == 1 && isNothing (pieceAt x (y + 2) brd) =
      [(x, y + 1), (x, y + 2)]
  | otherwise = [(x, y + 1)]
  where
    enpassantCaps = filter (\(x2, y2) ->
        enpassant brd == Just (x2, y2)) $
        filter (uncurry inBounds) [
          (x + 1, y), (x - 1, y)]
    captures = filter (\(x2, y2) -> opponentAt x2 y2) $
      (enpassantCaps ++ filter (uncurry inBounds)
      [(x + 1, y + 1), (x - 1, y + 1)])

movesFrom' (Piece Black Pawn)
  | not $ null captures = captures
  | y == 0 || isJust (pieceAt x (y - 1) brd) = []
  | y == 6 && isNothing (pieceAt x (y - 2) brd) =
      [(x, y - 1), (x, y - 2)]
  | otherwise = [(x, y - 1)]
  where
    enpassantCaps = filter (\(x2, y2) ->
        enpassant brd == Just (x2, y2)) $
      filter (uncurry inBounds) $
      [(x - 1, y), (x + 1, y)]
    captures = filter (\(x2, y2) -> opponentAt x2 y2) $
      (enpassantCaps ++ filter (uncurry inBounds)
      [(x - 1, y - 1), (x + 1, y - 1)])
```

`Minimax.hs`

```
module Minimax( MinimaxTree(..)
              , moveList
              , score
              , boardOf
              , next
              , nextBoards
              , minimaxFrom
```

```haskell
                 , fromRoot
                 , negaMax
                 , itDeep
                 ) where

import Data.Maybe
import Data.List
import Chess
import Control.Parallel.Strategies(using, parList, rseq)

-- Get all possible moves
moveList :: Board -> [(Int, Int, Int, Int)]
moveList brd
  | null caps = empty
  | otherwise  = caps
    where
      empty = [(x, y, fst z, snd z) | x <- [0..7], y <- [0..7], z <- movesFrom x y brd]
      caps = filter (\(_, _, x2, y2) -> isJust $ pieceAt x2 y2 brd) empty

{- Data structure encoding minimax tree
 - Root is starting position
 - Parital is a node
 - Final is a leaf
 -}
data MinimaxTree = Root (Board) [MinimaxTree]
                 | Partial (Board) [MinimaxTree] (MinimaxTree)
                 | Final (Board) (MinimaxTree)

boardOf :: MinimaxTree -> Board
boardOf (Root brd _)      = brd
boardOf (Partial brd _ _) = brd
boardOf (Final brd _)     = brd

next :: MinimaxTree -> [MinimaxTree]
next (Root _ nxt)      = nxt
next (Partial _ nxt _) = nxt
next _                 = error("Next called on leaf")

nextBoards :: Board -> [Board]
nextBoards brd = [(fixedMove x y x2 y2) | (x,y,x2,y2) <- moveList brd]
    where
      fixedMove x y x2 y2 = case move' x y x2 y2 brd of
          Right new_brd -> new_brd
          Left _        -> error("Invalid move given")

minimaxFrom' :: MinimaxTree -> Board -> MinimaxTree
minimaxFrom' parent brd
    | null nextPos  = Final brd parent
    | otherwise     = partial
      where
        partial = Partial brd (map (minimaxFrom' partial) nextPos) parent
        nextPos = nextBoards brd

--Builds the minimax tree from a given node
minimaxFrom :: Board -> MinimaxTree
minimaxFrom brd = root
    where
      root = Root brd (map (minimaxFrom' root) nextPositions)
      nextPositions = nextBoards brd

fromRoot :: MinimaxTree -> [Board]
fromRoot (Root brd _)           = brd:[]
fromRoot (Partial brd _ parent) = brd:(fromRoot parent)
fromRoot (Final brd parent)     = brd:(fromRoot parent)

-- Static evaluation. 100 = win.
score :: MinimaxTree -> Int
score (Final brd _) = if turn brd == White then 100 else -100
score mmTree        = max (-99) $ min 99 heuristic
    where
      brd       = boardOf mmTree
      mult      = if turn brd == White then 1 else -1
```

```haskell
        qBlack    = length $ piecesOf Black brd
        qWhite    = length $ piecesOf White brd
        flexMe    = length (next mmTree)
        flexOp    = length (moveList brd{turn =
                      if turn brd == White then Black else White})
        heuristic = 10 * (qBlack `div` qWhite - 1) + mult * (flexMe - flexOp)

-- Simplified minimax
negaMax :: Int -> MinimaxTree -> Int
negaMax _ (Final brd _)        = if (turn brd == White) then 100 else -100
negaMax n mmTree
  | n == 0     = mult * (score mmTree)
  | otherwise = -minimum (map (negaMax (n-1)) nb `using` parList rseq)
    where
      nb = take 4 $ sortOn (negate . (*mult) . score ) $ next mmTree
      brd = boardOf mmTree
      mult = if (turn brd == White) then 1 else -1

-- Minimax that stores the best move.
miniMaxWithMoves :: Int -> MinimaxTree -> (Int, [Char])
miniMaxWithMoves n mmTree =
    maximumBy (\(x,_) (y,_) -> compare x y) $
    zip results (map stringMove $ moveList $ boardOf mmTree)
    where
      stringMove (x1,y1,x2,y2) = (posToStr (x1,y1)) ++ (posToStr (x2,y2))
      nb = take 8 $ sortOn (negate . score ) $ next mmTree
      results = map (negate . (negaMax $ n-1)) nb
                        `using` parList rseq

-- Calls minimax with increasing depth until answer.
itDeep :: Int -> Int -> Board -> (Int, [Char], Int)
itDeep depth limit brd
  | abs (fst results) == 100 || depth > limit = ret
  | otherwise = itDeep (depth+1) limit brd
    where
      ret = (fst results, snd results, depth)
      results = miniMaxWithMoves depth $ minimaxFrom brd
```

`Main.hs`

```haskell
import Data.Maybe
import Chess; import Chess.FEN
import Minimax
import Control.Parallel.Strategies(using, parList, rseq)

import System.Environment(getArgs, getProgName)
import System.IO.Error(catchIOError, isUserError, isDoesNotExistError,
    ioeGetFileName, isPermissionError)
import System.Exit(die)

-- Main entry point. Solves board, returns score (White wins) + best first move
solve :: Board -> (Int, [Char], Int)
solve brd = itDeep 1 8 brd

-- Formats the result into something readable
parse :: (Int, [Char], Int) -> [Char]
parse (scr, mv, qmv)
  | scr == 100
      && odd qmv  = intro ++ ", White highly favored," ++ conc
  | scr == 100 = intro ++ ", Black highly favored," ++ conc
  | scr > 0     = intro ++ ", White favored (" ++ show scr ++
                        ")" ++ conc
  | otherwise      = intro ++ ", Black favored (" ++ show scr ++
                        ")" ++ conc
    where
      intro = "Best move: "      ++ show mv
      conc  = " search depth: " ++ show qmv

main:: IO()
main = do [filename, cases] <- getArgs
          contents <- readFile filename
```

```
         let brds = map (fromJust . fromFEN) . take (read cases) $ lines contents
             results = map parse (map solve brds `using` parList rseq)
         sequence_ $ map putStrLn results

  `catchIOError` \e -> do
    pn <- getProgName
    die $ case ioeGetFileName e of
      Just fn | isDoesNotExistError e -> fn ++ ": no such file"
              | isPermissionError e   -> fn ++ ": Permission denied"
      _       | isUserError e         -> "Usage: " ++ pn ++
                  " <filename> <# of test cases>"
              | otherwise             -> show e
```

`play.hs`

```
import Data.Maybe
import System.Environment(getArgs)
import Chess; import Chess.FEN; import Minimax

play :: Board -> IO()
play brd = do putStrLn $ show brd
              mv <- getLine
              if mv == "best"
              then putStrLn . show $ itDeep 1 6 brd
              else
                case move mv brd of
                  Right nxt_brd -> play nxt_brd
                  Left mvError  -> do
                    putStrLn $ show mvError
              play brd

main :: IO ()
main = do args <- getArgs
          let brd = case args of
                      []  -> Chess.FEN.defaultBoard
                      fen -> fromJust . fromFEN $ unwords fen
          play brd
```

Parallelism

We found that the simple method of creating sparks recursively works decently well. Using `rseq` in the recursive calls of the `negaMax` function led to good speedups. Each recursive call to the negaMax function creates at most 4 sparks except the first call which generates 8, and we only search up to a depth of 8, so there may be at most $8 * 4^7 = 131072$ sparks bring created at a time. But many of these sparks don't ignite new sparks so the actual amount is much less.

Even after running through 200 examples the spark pool did not overflow, so we very reliably run our algorithm. Additionally we ran our test cases in parallel, a la Marlow.

Benchmarks

We ran the program on our list of 200 randomly generated board positions, using a varying number of CPU cores.

N	Time (s)	Speedup
1	116.6	1x
2	60.3	1.93x
3	45.9	2.54x
4	40.5	2.88x
5	39.2	2.97x
6	37.8	3.08x
7	36.1	3.22x
8	35.1	3.32x

The performance gains start off linear, but soon drop off and peak at around 8 cores, which was 3.32 times faster than running it sequentially. At 8 cores, there are over 4 million sparks created and only 9800 of them are converted. We thought that this may be due to our strategy creating a new spark at every node in the graph, but running a modified strategy where spark creation was limited to depth 2 did not make a significant performance difference. Still, these are good performance gains. Running the program on 4 cores leads to around a 3x speedup, which suggests that our strategy works well and that searching the tree can be parallelized fairly effectively.