An efficient minimax implementation for noughts and crosses

Izaak van Dongen

July 12, 2018

Contents

List of Listings

1 Introduction

This is a set of programs that enable a user to play noughts and crosses "locally with a friend" (themselves), or to play against a computer, or even to watch the computer desperately trying to win against itself. It also supports arbitrarily sized boards, and probably several cases of overly optimised approaches to problems like the detection of a winning condition on a board.

Internally, the board is represented as one-dimensional array (arraylist) of any of {`None, True, False`}, corresponding to {Empty, Crosses, Noughts}. The single dimensionality is actually pretty natural, as a noughts and crosses board is not fundamentally a list of lists, but a set of tiles with some subsets considered to be "in a row". The board is enumerated from the top-left tile, left-to-right, top-to-bottom. When x-y indexing is used, the co-ordinate (x, y) corresponds to the index $3y + x$. This has a couple of nice properties which will come up later.

Minmax is implemented by a single "optimise" function that recursively searches the tree, with several specialisations over general minmax. These include for example not having to keep maximising after encountering a win, as the evaluation heuristic is known to only be able to evaluate the board as {win, lose, draw}.

The tests for winning conditions are implemented in listing 2. They work by instantiating a single list of indices for each subset considered a "row", and then creating an array indexed by board indices of lists of these subsets, where the subsets stored at a board index represent all the rows that tile is a part of.

Further explanation of implementation details may be included in docstrings in the source.

I decided to develop in Python as I had some quite ambitious ideas about the algorithm, which I thought would be easier to express by using the more functional side of Python. Python is also nicely expressive, which makes it easier to develop without getting bogged down in syntax. Similarly, I wrote a command-line program because it doesn't involve using Lazarus.

2 Implementation

Listing 1 has some boilerplate classes that are shared throughought, to give some abstract representation of the game.

```
"""
Base module with a couple of shared types
"""

from enum import Enum

class GameFinish(Exception):
    """
    Exception to throw when a game is finished
    """
    pass

class Win(GameFinish):
    """
    If a game is won
    """
    pass

class Draw(GameFinish):
    """
    If a game is drawn
    """
    pass

class State(Enum):
    """
    Enum to represent the possible states of a board
    """
    DRAW = 0
    X_WIN = 1
    O_WIN = 2
    NEUTRAL = 3
```

Listing 1: `base.py`: Some shared base classes

Listing 2 implements all of the "checking" - this is mostly the generation of rows and groups, and then some simple list comprehensions and generator expressions to evaluate a board using these.

The subsets are represented as Python `range`s. This is possible as each row is a linear succession of indices in this linearisation of the board. These ranges are implemented very efficiently, using only integer arithmetic to generate intermediate range values or to determine membership.

Because these groups are used often, they are stored in a registry and generated when needed. This is all handled by the module itself, so a user only needs to call `get_groups`.

```python
"""
Functions implementing grid checks, efficiently.
"""

from argparse import ArgumentParser

from base import State
from formatting import get_board_template

def get_args():
    """
    Get size of board
    """
    parser = ArgumentParser(description=__doc__)
    parser.add_argument("n", type=int, help="size of board to check")
    return parser.parse_args()

def _make_groups(n):
    """
    This function should not be accessed directly - use get_groups.  Construct
    the list of "groups" present on the board - rows, columns, diagonals. Groups
    are represented as Python ranges because they're all around suitable -
    efficient, linear etc. Return a list of lists of these groups, where
    indexing by board position returns the list of groups that that position is
    contained in. This is pretty memory-efficient as each actual list is only
    stored once, and the remainder of the data structure is primarily pointers.
    """
    rows = ([r for i in range(n)
               for r in [range(i * n, i * n + n), range(i, i + n ** 2, n)]]
          + [range(0, n ** 2, n + 1), range(n - 1, n ** 2 - n + 1, n - 1)])

    return [[r for r in rows if i in r] for i in range(n ** 2)]

# The registry for groups
GROUP_REGISTRY = {}

def get_groups(n):
    """
    Access a list of groups. This checks if the groups have been generated and
    cached already, and if not, does so, caching them in the process, and then
    returns.
    """
```

```python
    if n in GROUP_REGISTRY:
        return GROUP_REGISTRY[n]
    else:
        GROUP_REGISTRY[n] = _make_groups(n)
        return GROUP_REGISTRY[n]

def is_run(board, pos, n):
    """
    Check if one tile is a part of any complete groups.
    """
    return any(len(set(board[i] for i in group)) == 1 for group in get_groups(n)[pos])

def get_state(board, n):
    """
    Get state of board. This function is given no information about position
    so is necessarily much slower. If you do know the last played tile, use
    is_run instead, as this only checks all groups pertaining to that tile.
    """
    for pos, m in enumerate(board):
        if m is not None:
            if is_run(board, pos, n=n):
                if m:
                    return State.X_WIN
                return State.O_WIN
    if board.count(None) == 0:
        return State.DRAW
    return State.NEUTRAL

def show_groups(n):
    """
    Demo function which displays each groups using functions from formatting.py.
    """
    board_temp = get_board_template(n)
    for ind, pos in enumerate(get_groups(n)):
        print(ind, pos)
        for g in pos:
            dft = [" "] * n ** 2
            for i in g:
                dft[i] = "G"
            dft[ind] = 'M'
            print(board_temp.format(*dft), end="\n\n")

if __name__ == "__main__":
    args = get_args()
    show_groups(args.n)
```

Listing 2: `checking.py`: Implementation of minimax

When the module is run directly, it dumps all of the groups for a given board size. Listing 3 has a sample of this behaviour (and also demonstrates the nice board formatting).

```
% python checking.py 3
0 [range(0, 3), range(0, 9, 3), range(0, 9, 4)]
   0   1   2
 0 M | G | G
  ---+---+---
 1   |   |
  ---+---+---
 2   |   |

   0   1   2
 0 M |   |
  ---+---+---
 1 G |   |
  ---+---+---
 2 G |   |

   0   1   2
 0 M |   |
  ---+---+---
 1   | G |
  ---+---+---
 2   |   | G

1 [range(0, 3), range(1, 10, 3)]
   0   1   2
 0 G | M | G
  ---+---+---
 1   |   |
  ---+---+---
 2   |   |

   0   1   2
 0   | M |
  ---+---+---
 1   | G |
  ---+---+---
 2   | G |

2 [range(0, 3), range(2, 11, 3), range(2, 7, 2)]
   0   1   2
 0 G | G | M
  ---+---+---
 1   |   |
  ---+---+---
 2   |   |

   0   1   2
 0   |   | M
  ---+---+---
 1   |   | G
```

```
  ---+---+---
 2   |   | G

   0   1   2
 0   |   | M
  ---+---+---
 1   | G |
  ---+---+---
 2 G |   |
```

Listing 3: Output of `checking.py`

Listing 4 shows the implementation of minimax with all previously described bells and whistles. When called directly, it can demonstrate its approach to a particular board state, shown in listing 5. My favourite part of this program is the generator `generate_moves`.

```python
"""
Implementation of the special case of the minmax algorithm, suited to noughts
and crosses.
"""

from textwrap import indent
from traceback import extract_stack

from base import Win
from checking import State, is_run
from formatting import print_board, strfboard, syms
from interface import SquareBoard, isqrt

from argparse import ArgumentParser

def get_args():
    """
    Get arguments if a demo run is being executed. A demo run will just
    determine the move to be used against a preset board, verbosely by default.
    """
    parser = ArgumentParser(description=__doc__)
    parser.add_argument("-b", "--board", type=SquareBoard, default='_o__x____',
                        help='The initial board state')
    parser.add_argument("-q", "--quiet", action="store_true",
                        help="do not print minmax tree")
    return parser.parse_args()

# boolean-indexed array to get states compactly and quickly
state_from_bool = [State.O_WIN, State.X_WIN]

def optimise(evaluations, is_crosses, minimise):
    """
    "Optimise" a sequence of results for a given player. This simultaneously
    implements both minimisation and maximisation with a bit of Boolean logic.
    It also knows how to short-circuit - if maximising, a win is known to be the
```

```
    best possible case and vice verse.
    """
    LOSE_STATE, WIN_STATE = state_from_bool[minimise ^ (not is_crosses)],
    ↪  state_from_bool[minimise ^ is_crosses]
    draw_seen = False
    for e in evaluations:
        if e == LOSE_STATE:
            return e
        elif e == State.DRAW:
            draw_seen = True
    if draw_seen:
        return State.DRAW
    return WIN_STATE

def generate_moves(board, is_crosses):
    """
    Generate possible moves on a board for a certain player. This works by
    mutating the actual board array for each possible move, followed by yielding
    both the move and the board. This is useful as it allows "iteration" over
    moves, while also being memory-efficient (which leads to time efficiency as
    there is no allocation overhead).
    A finally clause implements the restoration of the board, which guarantees
    that the board will retain its state from before after this function exits,
    even if the function is interrupted by, for example, a break.
    """
    for ind, i in enumerate(board):
        if i is None:
            try:
                board[ind] = is_crosses
                yield ind, board
            finally:
                board[ind] = None

def evaluate_board(board, is_crosses, crosses_playing, prev_move, depth, n,
      verbose=False):
    """
    Evaluate a board-state for a given player. This recursively generates moves,
    evaluates them and optimises them.
    Allows printing diagnostics with the verbosity parameter. It will indent
    depending on the current length of the callstack, which helps keep track of
    the recursion.
    """
    verbose and print(indent("Examining as {} {}:\n{}"
                            .format(syms[crosses_playing], depth,
                               indent(strfboard(board, n), ' ')),
                        ' ' * len(extract_stack())))
    if is_run(board, prev_move, n):
        state = state_from_bool[not crosses_playing]
        verbose and print(indent("State here: {}"
                    .format(state), " " * len(extract_stack())))
        return state
```

```python
    elif depth == len(board):
        verbose and print(indent("Draw here", " " * len(extract_stack())))
        return State.DRAW
    else:
        return optimise(
                (evaluate_board(board, is_crosses, not crosses_playing,
                                move, depth + 1, n, verbose=verbose)
                   for move, board in generate_moves(board, crosses_playing)),
                 is_crosses, not crosses_playing ^ is_crosses)

def get_computer_move(board, is_crosses, n, verbose=False):
    """
    Apply board evaluation to all possible moves and
    select, in order:
    - A winning move
    - A drawing move
    - Any move (which will be a losing move)
    """
    # optimisation: play here for an empty board, because searching through the
    # whole board's tree is known to be unnecessary
    if all(i is None for i in board):
        return 0
    WIN_STATE = state_from_bool[is_crosses]
    moves = generate_moves(board, is_crosses)
    draw = None
    for move, board in moves:
        ev = evaluate_board(board, is_crosses, not is_crosses, move,
                            len(board) - board.count(None), n, verbose=verbose)
        if ev == WIN_STATE:
            verbose and print("Win incoming")
            return move
        elif ev == State.DRAW:
            verbose and print("Draw forcable")
            draw = move
    if draw is not None:
        return draw
    return board.index(None)

def do_computer_move(board, is_crosses, n, verbose=False):
    """
    Wraps get_computer_move to print some stuff, mutate the board and check for
    winning conditions.
    """
    move = get_computer_move(board, is_crosses, n, verbose=verbose)
    board[move] = is_crosses
    print("Computer plays at ({}, {})".format(move % n, move // n))
    print_board(board, n)
    if is_run(board, move, n):
        raise Win("I'm sorry, Dave. I'm afraid I can't do that.")

if __name__ == "__main__":
```

```python
    args = get_args()
    board = args.board
    n = isqrt(len(args.board))
    print(board, n)
    print_board(board, n)
    do_computer_move(board, True, n, verbose=not args.quiet)
```

Listing 4: `computer.py`: Implementation of minimax

```
% python computer.py -b "x__ _o_ o_x"
[True, None, None, None, False, None, False, None, True] 3
   0   1   2
 0 X |   |
  ---+---+---
 1   | O |
  ---+---+---
 2 O |   | X

    Examining as O 5:
        0   1   2
      0 X | X |
       ---+---+---
      1   | O |
       ---+---+---
      2 O |   | X
       Examining as X 6:
           0   1   2
         0 X | X | O
          ---+---+---
         1   | O |
          ---+---+---
         2 O |   | X
       State here: State.O_WIN
    Examining as O 5:
        0   1   2
      0 X |   | X
       ---+---+---
      1   | O |
       ---+---+---
      2 O |   | X
       Examining as X 6:
           0   1   2
         0 X | O | X
          ---+---+---
         1   | O |
          ---+---+---
         2 O |   | X
          Examining as O 7:
              0   1   2
            0 X | O | X
             ---+---+---
```

```
           1 X | O |
            ---+---+---
           2 O |   | X
            Examining as X 8:
                0   1   2
              0 X | O | X
               ---+---+---
              1 X | O | O
               ---+---+---
              2 O |   | X
               Examining as O 9:
                   0   1   2
                 0 X | O | X
                  ---+---+---
                 1 X | O | O
                  ---+---+---
                 2 O | X | X
               Draw here
            Examining as X 8:
                0   1   2
              0 X | O | X
               ---+---+---
              1 X | O |
               ---+---+---
              2 O | O | X
            State here: State.O_WIN
         Examining as O 7:
             0   1   2
           0 X | O | X
            ---+---+---
           1   | O | X
            ---+---+---
           2 O |   | X
         State here: State.X_WIN
      Examining as X 6:
          0   1   2
        0 X |   | X
         ---+---+---
        1 O | O |
         ---+---+---
        2 O |   | X
         Examining as O 7:
             0   1   2
           0 X | X | X
            ---+---+---
           1 O | O |
            ---+---+---
           2 O |   | X
         State here: State.X_WIN
      Examining as X 6:
          0   1   2
```

```
         0 X |   | X
          ---+---+---
         1   | O | O
          ---+---+---
         2 O |   | X
          Examining as O 7:
              0   1   2
            0 X | X | X
             ---+---+---
            1   | O | O
             ---+---+---
            2 O |   | X
          State here: State.X_WIN
       Examining as X 6:
           0   1   2
         0 X |   | X
          ---+---+---
         1   | O |
          ---+---+---
         2 O | O | X
          Examining as O 7:
              0   1   2
            0 X | X | X
             ---+---+---
            1   | O |
             ---+---+---
            2 O | O | X
          State here: State.X_WIN
Win incoming
Computer plays at (2, 0)
   0   1   2
 0 X |   | X
  ---+---+---
 1   | O |
  ---+---+---
 2 O |   | X
```

Listing 5: Output of `computer.py`

Listing 6 contains the code that formats boards as seen in previous listings (such as 5 and 3). It works by generating a "template" string, which is a string that is in a format that can be string-interpolated by `str.format`. It uses a similar registry principle to listing 2.

It makes most judicious use of Python various inline iteration capability, relegating the template generation to a single lexical line of code.

It can also demo its functionality when run directly, as shown in listing 7

```
"""
Pretty-printing OXO boards, using pre-calculated templates.
"""

```

```
from argparse import ArgumentParser
from random import choices

def get_args():
    """
    Get size of demo board in case of demo run
    """
    parser = ArgumentParser(description=__doc__)
    parser.add_argument("n", type=int, help="size of demo board")
    return parser.parse_args()

def _make_board_template(n):
    """
    Do not use this function. Use get_board_template instead.
    Generate a str.format compatible template to format a noughts and crosses
    board. This is a lot easier and faster than dynamically generating all of
    the "structure" of the board every time.
    """
    return ("{}\n{}"
              .format("".join(map("{:4}".format, range(n))),
                  "  {}".format("+".join(["---"] * n))
                     .join("\n\n").
                         join(map("{0[0]:2}{0[1]}".format,
                                  enumerate(["|".join([" {} "] * n)] * n)))))

# registry to cache templates
BOARD_REGISTRY = {3: _make_board_template(3)}

def get_board_template(n):
    """
    Get a template by checking to see if it has already been calculated and
    cached, and otherwise doing so before returning it. This layer of
    abstraction prevents any arduous calculation on module import, but rather
    incurs a slight penalty on first usage of the function (which is more likely
    through another function).
    """
    if n in BOARD_REGISTRY:
        return BOARD_REGISTRY[n]
    else:
        BOARD_REGISTRY[n] = _make_board_template(n)
        return BOARD_REGISTRY[n]

def strfboard(board, n):
    """
    Format standard board representation as string.
    """
    return get_board_template(n).format(*map(get_sym, board))

def print_board(board, n):
    """
    Print standard board representation as string.
```

```python
    """
    print("{}\n".format(strfboard(board, n=n)))

syms = "OX"

def get_sym(i):
    """
    Translate (None, True, False) to " XO"
    """
    if i is None:
        return " "
    return syms[i]

if __name__ == "__main__":
    args = get_args()
    n = args.n
    print("{0}x{0} template:\n{1}".format(n, get_board_template(n)))
    print("\nrandom {0}x{0} board:".format(n))
    print(strfboard(choices([None, True, False], k=n**2), n=n))
```

Listing 6: `formatting.py`: Formatting internal board arrays as strings

```
% python formatting.py 5
5x5 template:
   0   1   2   3   4
 0 {} | {} | {} | {} | {}
  ---+---+---+---+---
 1 {} | {} | {} | {} | {}
  ---+---+---+---+---
 2 {} | {} | {} | {} | {}
  ---+---+---+---+---
 3 {} | {} | {} | {} | {}
  ---+---+---+---+---
 4 {} | {} | {} | {} | {}

random 5x5 board:
   0   1   2   3   4
 0 X |   | X | X | X
  ---+---+---+---+---
 1 X | X | O | X |
  ---+---+---+---+---
 2   | X | X | O | O
  ---+---+---+---+---
 3 X |   |   |   |
  ---+---+---+---+---
 4 O | X | O | X | X
```

Listing 7: Output of `formatting.py`

Listing 8 deals with some of the really dull stuff, like getting user input.

```python
"""
Handling and verifying user input
"""

from base import Win
from formatting import print_board, syms
from checking import is_run

# boolean-indexed array to get a string name for player
name_from_bool = ["noughts", "crosses"]

# dictionary to translate symbols to internal representation of tile states
state_from_string = {"_": None, 'x': True, 'o': False}

def isqrt(n):
    """
    Calculate the integer square root of a number using the "bit-shift"
    algorithm.
    """
    if n < 2:
        return n
    else:
        small = isqrt(n >> 2) << 1
        large = small + 1
        if large ** 2 > n:
            return small
        else:
            return large

def SquareInt(s):
    """
    Acts as a "parser" for perfect square integers for argparse
    """
    n = int(s)
    if isqrt(n) ** 2 != n:
        raise ValueError("{!r} is not a square number".format(s))
    return n

def SquareBoard(board):
    """
    Acts as a "parser" for strings representing square boards, similar to
    SquareInt. Ignores all non-interesting characters and demands squareness.
    """
    b = [state_from_string[c] for c in board if c in state_from_string]
    if isqrt(len(b)) ** 2 != len(b):
        raise ValueError('The board must be square')
    return b

def get_pos(s, n):
    """
    Get position in 1d list from 2d coordinate reference
```

```
    """
    x, y = map(int, s.split())
    if not all(0 <= c < n for c in (x, y)):
        raise ValueError("Not in range [0,{})".format(n))

    return y * n + x

def get_input(board, is_crosses, n):
    """
    Get user input of where to play on a board.
    """
    print("You are playing as {}".format(name_from_bool[is_crosses]))
    while True:
        try:
            mov = get_pos(input("Enter the position you want to play in > "), n)
            if board[mov] is not None:
                raise ValueError("This position is already taken")
        except ValueError as ve:
            print(ve)
            continue
        return mov

def do_player_move(board, is_crosses, n):
    """
    Execute player move - assumes board is valid at start of turn.
    """
    print_board(board, n)
    try:
        pos = get_input(board, is_crosses, n)
    except (KeyboardInterrupt, EOFError):
        raise Win("\n{} wins because {} is a coward"
                  .format(syms[not is_crosses], syms[is_crosses]))
    board[pos] = is_crosses
    if is_run(board, pos, n):
        raise Win("{} wins".format(syms[is_crosses]))
```

Listing 8: `interface.py`: Dealing with user input

Listing 9 ties it all together, providing a pretty sophisticated interface through command-line arguments (see the `get_args` function). It allows you to play against yourself, the computer, or even just for you to watch the computer instantly force itself into a draw, if that's your thing.

The only other application of the `--battle` mode is to change the size of the board to anything more than 3, and watch your CPU melt. This is due to the size of the search tree growing with $O(2^{(n^2)})$, which, however, nicely implemented your tree search is, is a bit of a party-pooper.

A sample of play is provided in listing 10

```
"""
Play noughts and crosses. Incorporates both human and computer players.
"""

```

```
from argparse import ArgumentParser
from itertools import cycle, repeat

from base import GameFinish, Draw
from interface import do_player_move
from computer import do_computer_move as _do_computer_move

def get_args():
    """
    Get configuration for the program. See the help text for details.
    """
    parser = ArgumentParser(description=__doc__)
    mode = parser.add_mutually_exclusive_group()
    mode.add_argument("-c", "--computer", action="store_true",
                    help="play against computer opponent")
    mode.add_argument("-b", "--battle", action="store_true",
                    help="computer plays against itself")
    parser.add_argument("--headstart", action="store_true",
                    help="start first when playing against computer")
    parser.add_argument("--noughts-start", action="store_true",
                    help="noughts to start instead of crosses")
    parser.add_argument("-s", "--size", type=int, default=3,
                    help="size of board to play on")
    parser.add_argument("-v", "--verbose", action="store_true",
                    help="Show minmax thought process")
    return parser.parse_args()

def play(board, players, noughts_start, n):
    """
    Play a game of noughts and crosses until a finishing condition or a draw,
    given an infinite iterable of players.
    """
    is_crosses = not noughts_start
    try:
        for player in players:
            player(board, is_crosses, n)
            is_crosses = not is_crosses
            if board.count(None) == 0:
                raise Draw("Nobody wins!")
    except GameFinish as gf:
        print("{}: {}".format(type(gf).__name__, gf))

if __name__ == "__main__":
    args = get_args()
    # Correctly initialise the infinite iterable of players according to
    # arguments
    vb = args.verbose
    do_computer_move = lambda *args: _do_computer_move(*args, verbose=vb)
    if args.computer:
        if args.headstart:
            players = cycle([do_player_move, do_computer_move])
```

```
        else:
            players = cycle([do_computer_move, do_player_move])
    elif args.battle:
        players = repeat(do_computer_move)
    else:
        players = repeat(do_player_move)
    play([None] * args.size ** 2, players, args.noughts_start, args.size)
```

Listing 9: `play.py`: Bringing it all together to play the game

```
Computer plays at (0, 0)
   0   1   2
 0 X |   |
  ---+---+---
 1   |   |
  ---+---+---
 2   |   |

   0   1   2
 0 X |   |
  ---+---+---
 1   |   |
  ---+---+---
 2   |   |

You are playing as noughts
Enter the position you want to play in > 1 0
Computer plays at (0, 1)
   0   1   2
 0 X | O |
  ---+---+---
 1 X |   |
  ---+---+---
 2   |   |

   0   1   2
 0 X | O |
  ---+---+---
 1 X |   |
  ---+---+---
 2   |   |

You are playing as noughts
Enter the position you want to play in > 0 2
Computer plays at (1, 1)
   0   1   2
 0 X | O |
  ---+---+---
 1 X | X |
  ---+---+---
 2 O |   |
```

```

   0   1   2
 0 X | O |
  ---+---+---
 1 X | X |
  ---+---+---
 2 O |   |

You are playing as noughts
Enter the position you want to play in > 2 2
Computer plays at (2, 1)
   0   1   2
 0 X | O |
  ---+---+---
 1 X | X | X
  ---+---+---
 2 O |   | O

Win: I'm sorry, Dave. I'm afraid I can't do that.
```

Listing 10: Output of `play.py`

3 Source

The full project in its directory structure, including this document (as a full-colour PDF and TeX file), can be found at `https://github.com/elterminad0r/noughtsandcrosses`.