CMPUT 396 – Tic-Tac-Toe Game

Recall minimax:

- For a game tree, we find the root minimax from leaf values
- With minimax we can always determine the score and can use a bottom-up approach

Why use minimax? It gives the best worst-case score (that is, the best score against all possible opponent strategies).

Why use minimax for 2-player search? 1) they are zero sum games (if something good happens to me it's bad for you) 2) we can maximize or minimize this score

Alpha-beta search is a minimax search with the obvious shortcuts taken.

Let's pick a game and use minimax to solve arbitrary states, or rather find the minimax value. We need a game with a relatively small state space, like tic-tac-toe.

Tic Tac Toe

Theoretical best of both players in tic-tac-toe?

- It should be a tie
- Can you prove it?

Tic-tac-toe dates back to ~2 CE with Ovid's (poet) book *Ars Amatoria III, lines 365-369*:

> There is another game divided into as many parts
> as there are months in the year;
> A small board has three pieces on either side,
> the winner must get all the pieces in a straight line.

Estimate for number of nodes in Tic-Tac-Toe search space: absolute maximum is 9!, but in reality it will be much lower because when you get to a win you don't have to keep looking (at minimum, a game will be five moves).

Why use minimax? Because it gives the best worst-case answers. In theory, it is bottom-up while taking the obvious cutoffs (alpha-beta).

Negamax

Negamax is a variation of minimax search that relies on the zero-sum property of a two-player game.

Good coding practice is to avoid code duplication. Negamax reduces code duplication.

One area where minimax suffers is that it has two cases: one where P1 moves and one where P2 moves (we maximize one and minimize the other). However, we don't really need two cases. In negamax, we compute the minimax value for the *player-to-move* so we only have one case: $score\_for\_p2(node) = neg(score\_for\_p1(node))$. At each node, we select the move that maximizes the negation of the score of the children: $neg(score(child))$.

This algorithm relies on the fact that $\max(a, b) = -\min(-a, -b)$ to simplify the implementation of the minimax algorithm. The value of a position to player A in such a game is the negation of the value to player B. Thus, the player-to-move looks for a move that maximizes the negation of the value resulting from the move: this successor position must by definition have been valued by the opponent. The reasoning of the previous sentence works regardless of whether A or B is on move. This means that a single procedure can be used to value both positions. This is a coding simplification over minimax, which requires that A selects the move with the maximum-valued successor while B selects the move with the minimum-valued successor. *So with negamax we don't care who is A and who is B, the algorithm works the same regardless.*

Warning: When using negamax, make sure that the leaf scores are for the **player-to-move** (minimax assumes all node scores are for the first player). To convert leaf scores in a minimax tree to equivalent leaf scores in a negamax tree: negate leaf scores whose distance-to-root is odd. The leaf scores whose distance-to-root is even do not need to be changed because their player-to-move is max.

Base case:

- If we win, return 1
- If there are no legal moves, return 0
- The best score so far: could be infinity, but we say -1 because the worst possible outcome is a loss and I *know* I can achieve a loss so we set it as -1 but obviously hope to do better

```
def eval(s):
  for player who would move next, return score of s

def negamax(s):
  if terminal(s):
    return eval(s)
  else:
    return max(for all children c of s, -negamax(c))
```

Note that with negamax we *always* maximize the score at every level ($max(-a, -b)$).

simple/ttt/ttt_classic.py → ab_neg(), pseudocode

```
for cell in legal_moves:
     Set value of cell for the player-to-move
     Call negamax recursively from position we have just updated
          Two arguments: position, player-to-move
     Gives us a negamax score
     Current best we can do is so_far
          Is -negamax score better than this
     Erase cell (to reset for for loop)
Return best score so far
```

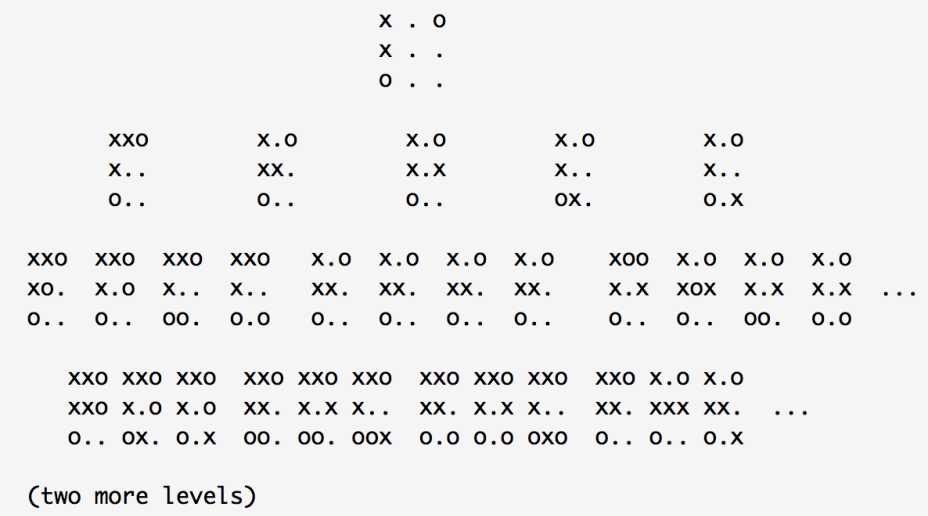
Minimax example, negamax format (negative scores of minimax):

```
at each node, score is for *** player-to-move ***

p1 to move            *
                   /  |  \
                 /    |    \
p2 to move     *      *      *
             / | \   / \   / | \
p1 to move * * * * * * * * * *
p1 score   2 3 -1 1 6 -4 2 9


p1 score              1
                   /  |  \
                 /    |    \
p2 score       1     -1      4
             / | \   / \   / | \
p1 score   * * * * * * * * * *
           2 3 -1 1 6 -4 2 9
```

Tic-Tac-Toe Example Trees

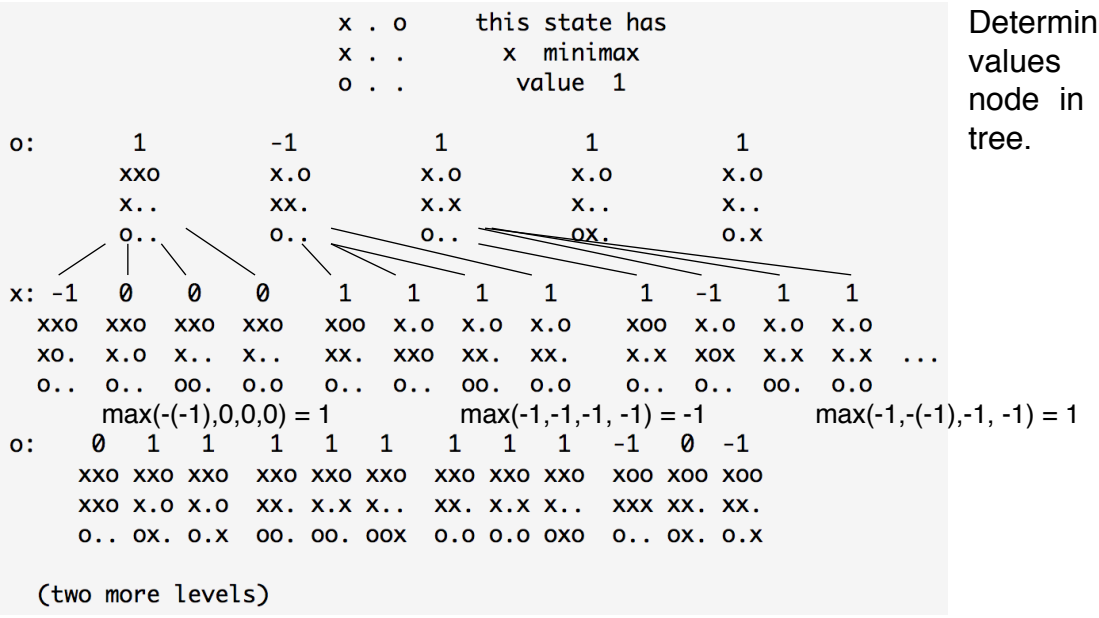
```
for x to move,              x . o
draw (part of) the          x . .
game tree                   o . .

then draw (part of)
the minimax tree

then draw
the proof tree, also called solution tree
(show best move winner)
```

Part of game tree:



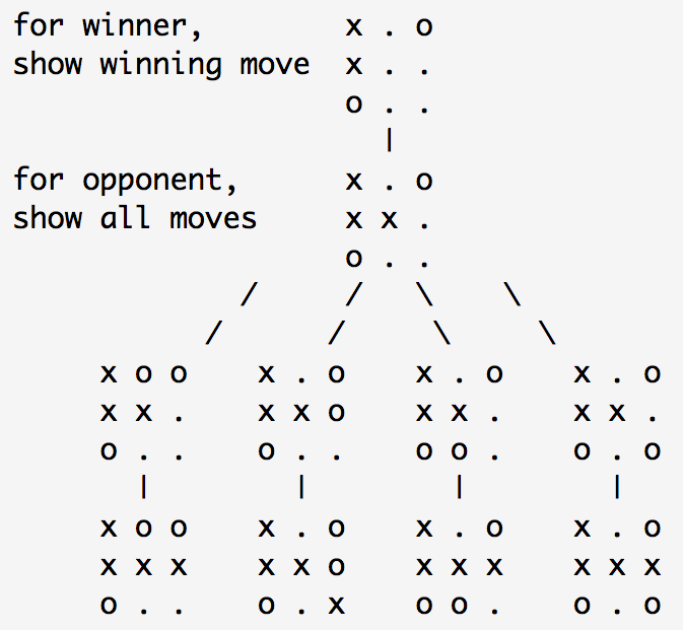
Start with the current position, make all the possible next moves, and do that over and over until there is a winner or a draw for that board.

Part of minimax tree:



Determine minimax values for each node in the game tree.

Proof tree:



Prove that a certain move for a certain position is a winning move by working through the tree.

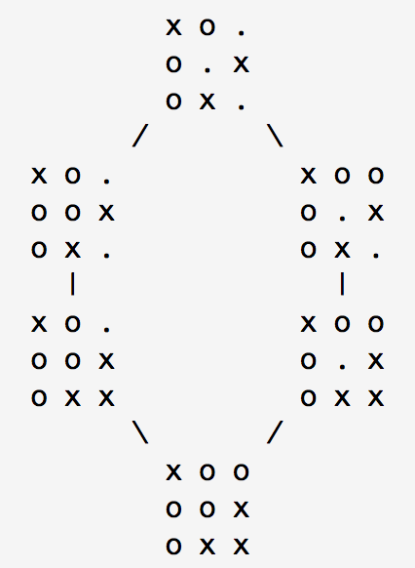
These can also give you an indication of how hard it will be to solve.

Simple Tic-Tac-Toe negamax code:

```
def negamax(psn, ptm): # position, player-to-move
  if psn.has_win(ptm):
    return 1  # previous move created win
  L = psn.legal_moves()
  if len(L) == 0:
    return 0  # board full, so draw
  so_far = -1  # best score so far
  for cell in L:
    psn.brd[cell] = ptm
    nmx = negamax(psn, opponent(ptm))
    so_far = max(so_far,-nmx)
    psn.brd[cell] = Cell.e   # reset brd by erasing cell
  return so_far

# solves 3x3 ttt with 740 170 calls
#                9! = 362 880
# nodes at most       986 410 = 9!/8! + 9!/7! + 9!/6! + ··· + 9!/1!
# can we do better ?
```

Transpositions

The board can represent the same position if it is rotated or reflected. In our negamax implementation (without checking for isomorphic moves), we hit all of these transpositions. We need to check: have we seen this board position before? We need a table with all positions we've seen before to take symmetry into account. As in the right, we should collapse all positions that were the same into one node.



- The Tic-Tac-Toe search space is not a tree because it has undirected cycles
- The Tic-Tac-Toe search space is a directed-acyclic graph (no directed cycles because it always moves top to bottom)
- Different move sequences that yield that same position are called **transpositions**

Pruning Tic-Tac-Toe Game Trees

How many Tic-Tac-Toe states need to be examined to find the minimax value of the corner opening move? 8! Or far fewer than 8!

Far fewer than 8! Because nodes can be pruned.

Win detection: in child minimax for loop, abort if winning child found (prune remaining siblings).

- Solve Tic-Tac-Toe with 129 988 nodes (when we check for early wins)
- If we don't check for early wins, more like 740 170.

We could also check for a forced move. Does the opponent have any move where they have two in a row and we can block? If yes, we shouldn't consider other positions.

We can add some additional conditions to our base case:

- Did someone win?
- Can I win on this move? Go there
- Can my opponent win on the next move? Block there
- Can I conclude that no one can win (every possible 3-in-a-row is blocked)? You can draw early

With a transposition table we should check:

1) Have we seen this exact position before (used different move sequence to get there)?
2) Have we seen a reflected or rotated position before?

Save the minimax values for positions we've seen before so we don't need to recalculate.

Tic-Tac-Toe Search Space Size

How big is the tic-tac-toe search space?

Root, 9 children, 9*8 grandchildren, 9*8*7 great-grandchildren

Total number of nodes < 1 + 9 + 9*8 + 9*8*7 + … + 9*8*7*6*5*4*3*2*1 = 986 410

Level 0	0 nodes	1 (starting position)
Level 1	9 nodes	9!/8! = 9
Level 2	9*8 nodes	9!/7! = 9*8
Level 3	9*8*7 nodes	9!/6! = 9*8*7
…		
Level 9	9! nodes at leaf level	9!/1! = 9*8*7*6*5*4*3*2*1

Why should it be less than that? Because some terminal nodes may not be at max depth, because a win is a leaf node and because of transpositions.

If we treat the search space as a tree, ignoring transpositions so we allow a position in multiple nodes, then the above number is a reasonable estimate.

How much of a space reduction do we get if we allow each position to appear in at most one node?

- Number of possible positions:
  - 9 cells, 3 possibilities for each (corner, edge, center) = 3^9 = 19 683
  - Each of the board positions can be empty – X – O, which we represent as 0 – 1 – 2 (so we can represent each board position by a 9-digit base 3 number)

Good news:

- Many of these positions are not reachable
- Many of these states are isomorphic (can be transformed to be the same)
  - From root state: only 3 non-isomorphic moves, so expect 6500 nodes
  - Alpha-beta search from root examines 3025 non-isomorphic states

Tic-Tac-Toe Board Representation Example

```
eg. this position       represented by this number:

   x . o
   x o .           102 120 010 (base 3) = 8 427 (decimal)
   . x .

an unreachable ttt position:

   x . o
   x . o
   x . o
```

Minimax pruning example (min/max format):

```
recall min/max format:
  p1 max, p2 min, show p1 values

         a
       /   \
      b     c
     / \   /|\
    d   e  f g h
    7   5  3 ? ?

assume dfs with children explored left-to-right
         *
       /   \
      5    <=3
     / \   /|\
    7   5  3 ? ?

after examining node     we know ...
  d                      val(b) <= 7
  e                      val(b) = min(7,5) = 5
  c                      val(c) <= 3

now, no need to learn more about val(c)
... val(c) <=3   <   val(b)
... p1  prefers   b   to   c
... so prune remaining children of c from our search
```

Alpha-beta search (negamax format)

```
recall negamax format:
  value(t) is for *** player to move from t ***
  p1_score( t ) = - p2_score( t )

def abnega(v, alpha, beta):
  if terminal(v):
    return eval(v)

  so_far = neg_infinity  # best score so far
  for each child c of v:
    so_far = max(so_far, - abnega( c, - beta, - alpha) )
    alpha = max( alpha, so_far )
    if alpha >= beta:
      break
   return so_far

abnega(root, neg_infinity, infinity)
```

Alpha-beta Trace Example:

```
         A
       /   \
      B     C
     / \   /|\
    D   E  F G H
    7   5  3 1 9

directory /games-puzzles-algorithms/simple/alphabeta
python3 alphabeta.py < t1.in

A -999 999 -999
  B -999 999 -999
    D -999 999 -999
    D leaf value 7
    D now best child of B
    D improved alpha( B ) to -7
    E -999 7 -999
    E leaf value 5
    E now best child of B
    E improved alpha( B ) to -5
  B -5 999 -5
  B now best child of A
  B improved alpha( A ) to 5
  C -999 -5 -999
    F 5 999 -999
    F leaf value 3
    F now best child of C
    F improved alpha( C ) to -3
    alpha >= beta, prune remaining children of C
  C -3 -5 -3
A 5 999 5
```

Search Enhancements

When you want to solve a game, start with minimax. Then make improvements like cutoffs with alpha-beta and minimizing code similarity with negamax.

Always start with the vanilla version of the algorithm, it won't be super fast at first. Ask yourself, how can we make this better?

- Move ordering: to maximize pruning, consider children in order from strongest to weakest (child strength is not known a priori but we can guess with a heuristic)
- Threat-search: check for wins or lose-threats (forced moves) first
    - Win-threat: if player has a winning move, make it
    - Lose-threat: if opponent has next-move-win, player must block at that cell, all other moves lose and can be pruned
- Alpha-beta revisions for Tic-Tac-Toe
    - Initialize with -1 rather than -infinity (because a loss is the worst case already, don't need to make it super big)

- Search stops when win/loss found
- Before starting search, order children so that win-threats precede lose-threats precede the rest of the moves
    - If I can win, I want to do that. If not, I should make sure I won't lose on the opponent's next turn by blocking them with my move. Then I can consider other moves.

Heuristics with limited depth alpha-beta search is the basis of many strong chess programs.