

Ultimate Tic Tac Toe

Aayush Bajaj

July 10, 2026

Contents

1	the game	1
1.1	rules	1
1.2	branching factor and depth	2
2	game-tree search	2
2.1	minimax	2
2.2	negamax	3
2.3	alpha-beta pruning	3
2.4	how much does pruning buy?	3
3	the agent	4
3.1	talking to the referee	4
3.2	the search	5
3.3	the heuristic, as designed	6
3.4	the heuristic, as executed	6
4	results	7
5	the c and java agents	8
6	see also	8

ordinary tic-tac-toe is a solved bore — both players draw with a lookup table. glue nine boards together and add one rule about *where* you are allowed to move, and suddenly the game tree is deep enough that you need actual search theory: minimax, the negamax reformulation, alpha-beta pruning, and a heuristic to stand in for the leaves you cannot reach. this page documents the agent my partner and i wrote for unsw comp3411 (assignment 3, “nine-board tic-tac-toe”), warts and all — and the warts are instructive.

*code: agent.py in the
cs3411-9ttt repo, plus alan
blair’s referee scaffolding
servt.c / game.c and starter
agents in c and java*

1 the game

1.1 rules

nine standard 3×3 boards are arranged in a 3×3 grid; boards and cells are both numbered 1–9 in row-major order. play alternates as usual, but with one twist that generates all the depth:

- **the sending rule:** if you play in cell k of the current board, your opponent must make their next move somewhere in board k .

- **the win condition** (comp3411 variant): the first player to complete three-in-a-row in *any single* subboard wins the whole game. if all reachable cells fill up, it is a draw.

the sending rule couples the boards: a locally brilliant move may hand the opponent a free win on another board. this is what kills greedy play and rewards lookahead.

the variant usually called ‘ultimate tic-tac-toe’ adds a meta-game: winning a subboard claims it as one big cell, and you must win three subboards in a row. the sending rule is identical; only the terminal test differs.

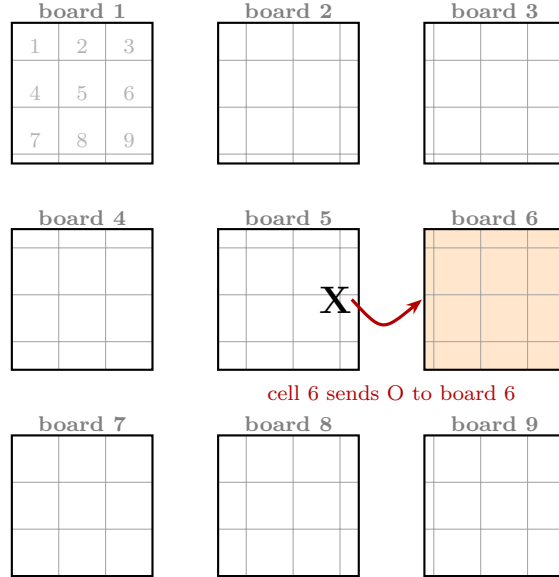


Figure 1: the nine-board layout. X plays cell 6 of board 5 (centre board), so O is sent to board 6 — O may only reply there. cell numbering shown faintly in board 1.

1.2 branching factor and depth

because the sending rule confines each reply to a single subboard, the branching factor is at most 9 (the free cells of the destination board), decaying as boards fill. a full game runs at most 81 plies; in practice games end well before that under the any-subboard win rule. a complete search is still hopeless from the opening — 9^d nodes at depth d means a naive depth-12 search already touches $\sim 2.8 \times 10^{11}$ positions — so the agent searches to a fixed depth and evaluates the frontier with a heuristic.

2 game-tree search

2.1 minimax

model the game as a tree: nodes are positions, edges are moves, and the two players alternate levels. assign every terminal node a value from the maximising player’s perspective (win > 0 , loss < 0 , draw 0). the **minimax value** of an internal node is

$$V(n) = \begin{cases} U(n) & n \text{ terminal} \\ \max_{c \in \text{succ}(n)} V(c) & n \text{ a max node} \\ \min_{c \in \text{succ}(n)} V(c) & n \text{ a min node} \end{cases} \quad (1)$$

computed by depth-first traversal (Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford, 2009). playing the child of maximal value

is optimal against an optimal opponent; against a weaker opponent it never does worse than the guaranteed value.

*standard treatment: russell
norvig, /artificial intelligence:
a modern approach/, ch. 5 —
aima.cs.berkeley.edu*

2.2 negamax

carrying separate max/min cases through an implementation invites sign bugs. the **negamax** trick exploits the zero-sum symmetry $\min(a, b) = -\max(-a, -b)$: store every node's value *from the perspective of the player to move*, and then a single rule suffices,

$$V(n) = \max_{c \in \text{succ}(n)} (-V(c)). \quad (2)$$

one recursive function, one negation at the call site. this is exactly the shape of the agent's search (and of alan blair's supplied single-board `ttt.py/~ttt.c/~ttt.java` examples, which the assignment agent generalises to nine boards).

2.3 alpha-beta pruning

alpha-beta threads two bounds through the depth-first traversal:

- α : the best value the maximiser can already guarantee on the current path (a lower bound on the outcome),
- β : the best the minimiser can already guarantee (an upper bound).

whenever $\alpha \geq \beta$ at a node, the node's remaining children are **pruned**.

soundness argument. suppose at a min node n we have examined some children and driven the running value v down to $v \leq \alpha$. the final value of n can only decrease as more children are examined, so $V(n) \leq \alpha$. but the maximiser already has a path elsewhere guaranteeing α ; it will therefore never route play through n , whatever the unexamined children hold. cutting them cannot change the minimax value at the root — pruning is *lossless*. the symmetric argument covers max nodes with $v \geq \beta$. in negamax form both cases collapse into one: search child c with the window $(-\beta, -\alpha)$, and cut when the negated result reaches β .

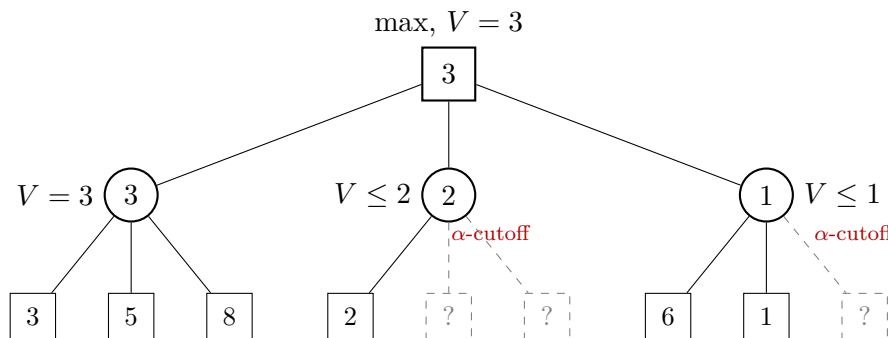


Figure 2: alpha-beta in action. after the left min-node settles at 3, the middle min-node's first leaf returns 2: its value is now $\leq 2 < 3 = \alpha$, so its remaining leaves (dashed) are pruned — nothing there can matter at the root.

2.4 how much does pruning buy?

everything depends on move ordering. if at every node the *best* move happens to be searched first, alpha-beta examines only

$$b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1 \quad (3)$$

leaves instead of b^d — asymptotically $O(b^{d/2})$, i.e. the same work buys *double the search depth*.¹ with poor ordering the exponent degrades back towards d . moral: a cheap heuristic that sorts probably-good moves first (tactical moves, centre squares, killer moves) pays for itself many times over. our agent, as we shall see, mostly declined this free lunch.

3 the agent

3.1 talking to the referee

the assignment ships a c referee (`servt.c`) that owns the real game state, enforces legality and the clock (30 s initially plus 2 s per move, enforced with `select()` on the agent's socket), and speaks a tiny prolog-flavoured line protocol over tcp:

referee -> agent	agent -> referee
-----	-----
<code>init.</code>	
<code>start(x).</code>	
<code>second_move(6,1).</code>	5 # opp opened board 6 cell 1; we reply
<code>third_move(6,1,5).</code>	9 # our 1st move + opp's reply; we reply
<code>next_move(7).</code>	3 # opp played cell 7; reply with a cell
<code>last_move(4).</code>	
<code>win(triple). / loss(timeout). / draw(full_board).</code>	
<code>end.</code>	

the python agent's event loop is a straight transliteration: connect, read lines, `parse()`, answer with a bare cell number. note where the state burden sits — the referee sends only the opponent's *cell*, and the agent must remember which board the sending rule points at.

```

1  # from cs341l-9ttt/src/agent.py (trimmed)
2  def main():
3      s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
4      port = int(sys.argv[2])          # usage: ./agent.py -p <port>
5      s.connect(('localhost', port))
6      while True:
7          text = s.recv(1024).decode()
8          if not text:
9              continue
10         for line in text.split("\n"):
11             response = parse(line)    # dispatch on
12             ↪ second_move/third_move/next_move/...
13             if response == -1:        # win./loss. -> hang up
14                 s.close()
15                 return
16             elif response > 0:
17                 s.sendall((str(response) + "\n").encode())

```

¹knuth & moore (1975), *an analysis of alpha-beta pruning*, artificial intelligence 6(4):293–326. they prove alpha-beta returns the same value as full minimax (formalising the soundness sketch above) and derive the $b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$ best-case leaf count.

3.2 the search

state is a 10×10 numpy `int8` array (index 0 unused, a c habit inherited from the scaffold): `boards[b][c]` is 0/1/2 for empty/us/them, and a global `curr` tracks which board the sending rule points at. the search is negamax with alpha-beta, in the same shape as the course's single-board example, plus two hand-rolled shortcut rules:

```
1  # from cs341l-9ttt/src/agent.py (trimmed; debug prints elided)
2  def alphabeta(player, m, board, subboard, alpha, beta, best_move, depth_limit):
3      best_eval = MIN_EVAL
4      if m >= depth_limit:
5          return evaluate_board(board, player)
6
7      # if the opponent threatens a triple in the current subboard, block it
8      if win_op(board[subboard], 2-player) != 0:
9          if boards[subboard][win_op(board[subboard], 2-player)] == 0:
10             best_move[m] = win_op(board[subboard], 1-player+1)
11             return 2000 + m
12
13     # if we can complete a triple right here, take it
14     if win_op(board[subboard], 1+player) != 0:
15         if boards[subboard][win_op(board[subboard], 1+player)] == 0:
16             best_move[m] = win_op(board[subboard], 1+player)
17             return 2000 + m
18
19     if game_won(2-player, board):
20         return -1000 + m          # better to win faster (or lose slower)
21
22     this_move = 0
23     for r in range(1, 10):
24         if board[subboard][r] == EMPTY:          # move is legal
25             this_move = r
26             board[subboard][this_move] = player  # make move
27             this_eval = -alphabeta(1-player, m+1, board, this_move,
28                                   -beta, -alpha, best_move, depth_limit)
29             board[subboard][this_move] = EMPTY   # undo move
30             if this_eval > best_eval:
31                 best_move[m] = this_move
32                 best_eval = this_eval
33                 if best_eval > alpha:
34                     alpha = best_eval
35                     if alpha >= beta:              # cutoff
36                         return alpha
37
38     return 0 if this_move == 0 else alpha
```

points worth noting, for and against:

- **the negamax recursion is textbook-correct:** child searched with the window $(-\beta, -\alpha)$, value negated, move made and unmade in place — no board copying, which matters in python.
- **the win_op shortcuts** scan the destination subboard for a two-in-a-row-plus-gap pattern (rows, columns, both diagonals — 24 pattern checks) and return immediately with score $2000 + m$, without expanding any children. this is simultaneously the agent's biggest strength and its biggest theoretical sin: it collapses enormous subtrees (see the node counts below), but it also *stops the search* at any node where a tactical move exists, assuming the tactic is best. a forced block that loses the game one board over is invisible.
- **depth term** m nudges the engine to win sooner and lose later — standard and

correct.

- **move ordering**: the child loop runs cells 1..9 in row index order. no centre-first, no killer moves — none of the $O(b^{d/2})$ best case is being courted here. a static ordering as trivial as corners-centre-edges (the fallback path in `play()` actually contains exactly such an ordering: 1,3,7,9,5,2,4,6,8) would have been free.
- **“iterative deepening”**: `play()` loops `depth_limit` from 1 to 6 and keeps only the final iteration’s move. real iterative deepening uses the previous iteration’s best move to order the next search, and a clock to decide when to stop; this version spends the time and collects neither benefit.
- **a print statement inside the node expansion**. every `alphabeta()` call printed the full nine-board position to stdout. at thousands of nodes per move this is a material constant-factor tax — and it was still there in the submitted code.

3.3 the heuristic, as designed

at the depth frontier `evaluate_board()` runs. the intent, per the header comment: rate each subboard by counting x’s and o’s along its 8 lines, sum the line ratings via `get_rating()`, and prefer moves that send the opponent to *bad* boards. the per-line scoring table (`evaluate_rating()`), taking `count_x`, `count_o` and total occupancy `count` of a 3-cell line:

line contents	rating
full line (<code>count == 3</code>)	-1
two of a kind in the line	-5
two mixed stones	-1
exactly one x	+3
one o with another stone present	+1

lower total = a board you would rather *send the opponent to*; the eight line ratings are summed per board, the nine sums go into a numpy vector, and `np.argmax` picks a winner.

3.4 the heuristic, as executed

reading the submitted code closely turns up a small museum of python gotchas, and they change what the function actually computes. the tests are written with bitwise `&~/~|` instead of `and~/~or` — but in python `&` binds *tighter than* `==`, so `a == 1 & b == 1` parses as the chained comparison `a == (1 & b) == 1`. executed consequences (real run):

```
1  b = [0, 2, 2, 0]                # a row: two opponent stones, one gap
2  print(b[1] == 2 & b[2] == 2)    # intended (b[1]==2) and (b[2]==2)
3
4  full = [1, 1, 2, 1]             # a fully occupied line
5  print(full[1] != 0 & full[2] != 0 & full[3] != 0) # the "full line" guard
6
7  count_x, count_o = 0, 2         # two opponent stones on a line
8  print(count_x == 2 | count_o == 2) # intended count_x==2 or count_o==2
```

True
False
False

- the equality chains like `b[1] == 2 & b[2] == 2` are *accidentally correct* for stone values in $\{0, 1, 2\}$: `2 & v` equals 2 exactly when `v` is 2, so the chain degenerates to the intended conjunction. luck, not design.
- the full-line guards `board[j] != 0 & ... != 0` are *always false* (the chain contains a literal `0 != 0`). perversely, this bug is load-bearing: the guarded `continue` statements sit in `while` loops whose counters increment *after* the guard, so if the guard ever fired the loop would spin forever. one bug shields another.
- `count_x == 2 | count_o == 2` misses the “two opponent stones” case entirely (in context it reduces to `count_x == 2`), so the -5 penalty meant to make the opponent’s near-wins unattractive never applies to *their* pairs — only to ours.
- in `evaluate_board()` the loop first adds win-opportunity bonuses to `board_ratings[num]` and then immediately *overwrites* the slot with `get_rating(...)` — the bonus logic is dead code. the loop also runs `range(1, 9)`, never rating board 9.
- the function returns `np.argmax(board_ratings)` — a *board index* in 0..9, not a score on the ± 1000 scale of the terminal values. every frontier node therefore evaluates to a small near-constant value, and the search’s actual playing strength comes almost entirely from the $2000 + m$ win/block shortcuts propagated up the tree.

i keep this section because it is the truest lesson of the assignment: an agent can look sophisticated, pass the marker, and beat a random opponent convincingly while its evaluation function is essentially returning noise. the tactics (make triples, block triples) were doing all the work.

the one genuine crash — a fallback path in `play()` that read a variable before assignment — is now fixed in the archived `agent.py`; everything else is preserved as submitted.

4 results

all runs below are real, on the referee compiled from the course’s `servt.c~/~game.c` with `gcc -Wall -O2` on this machine. against the supplied random agent `randt`, the agent (X) wins by triple in board 3:

```
$ ./servt -p 31411 & python3 agent.py -p 31411 & ./randt -p 31411
Connecting to port 31411
```

```
. . 0 | . . 0 | X X x
0 . 0 | . 0 . | . . .
. 0 . | . . . | . . .
-----+-----+-----
X . . | X . . | . . 0
. . . | . . . | . . X
. . . | . . . | . . .
-----+-----+-----
. . . | X X . | . . .
. . . | . . . | . . .
X 0 . | . . . | . . .
```

Player X wins (triple)

Yay!! We win!! :)

against the course’s reference lookahead player `lookt` (shipped as a compiled bi-

nary), the agent loses — `lookt` (O) completes the left column of board 1:

```
$ ./servt -p 31412 & python3 agent.py -p 31412 & ./lookt.mac -p 31412
Connecting to port 31412
```

```
0 X . | X . . | X . .
0 . . | . 0 . | . . .
0 . . | . . 0 | . . .
-----+-----+-----
. . 0 | . . . | . 0 .
. . . | X . X | . . .
. . . | . . . | . . .
-----+-----+-----
. X . | . . . | X . .
. . . | . . . | . . .
. . . | . . . | . . .
```

Player 0 wins (triple)

We lost :(

instrumentation from the winning game: the agent played 8 moves, and counting board printouts in the debug log gives **7850 node expansions total** across all six deepening passes of all eight moves — under a thousand nodes per move for a nominal depth-6 search over branching factor ≤ 9 . the `win_op` shortcuts are amputating almost the entire tree (a full depth-6 negamax would touch up to $9^6 \approx 5.3 \times 10^5$ nodes *per move*). that explains both results above: enough tactics to punish random play, not enough actual search to survive a real lookahead opponent. the debug log for one game weighed 5.6 mb.

5 the c and java agents

the repo also carries `agent.c` and `Agent.java`. neither contains the search: both are the course's starter scaffolds that answer every `next_move` with a *random* legal cell (the c one via `random() % 9` retries, the java one via `rand.nextInt(9)`). the interesting difference is architectural. the c agent implements only callbacks — `agent_second_move()`, `agent_next_move()` — because the course's `client.c` owns the socket, parses the protocol with `sscanf` patterns, and calls into you (Kernighan, Brian W. and Ritchie, Dennis M., 1988). the python and java agents each re-implement that transport layer by hand (string splitting on "(" and ","). same protocol, two philosophies: framework-calls-you versus you-parse-everything. the single-board `ttt.c~/~ttt.py~/~ttt.java` at the repo root are alan blair's reference alpha-beta implementations — structurally the exact skeleton `agent.py`'s search grew from, down to the `best_move[]` array threaded through the recursion.

6 see also

- [adversarial searching](#) — parent section
- [chess bot](#) — the same search machinery with move ordering, pv tables and iterative deepening done properly
- [connect 4](#) — another adversarial toy game

- **classical algorithms** — depth-first search, the substrate under all of this

references

Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford (2009). *Introduction to Algorithms*, MIT Press.

Kernighan, Brian W. and Ritchie, Dennis M. (1988). *The C Programming Language*, Prentice Hall.